The Instruction Semantics Generation for GCC Register Transfer Language

Timothy Daly, Daniel Ferullo, Stacy Prowell

Sept. 2007

Abstract

Contents

1 GCC machine description for IA-32 and x86-64

```
;;
;; The original PO technology requires these to be ordered by speed,
;; so that assigner will pick the fastest.
;;
;; See file "rtl.def" for documentation on define_insn, match_*, et. al.
;;
;; Macro REG_CLASS_FROM_LETTER in file i386.h defines the register
;; constraint letters.
;;
;; The special asm out single letter directives following a '%' are:
;; 'z' mov%z1 would be movl, movw, or movb depending on the mode of
;;     operands[1].
;; 'L' Print the opcode suffix for a 32-bit integer opcode.
;; 'W' Print the opcode suffix for a 16-bit integer opcode.
;; 'B' Print the opcode suffix for an 8-bit integer opcode.
;; 'Q' Print the opcode suffix for a 64-bit float opcode.
;; 'S' Print the opcode suffix for a 32-bit float opcode.
;; 'T' Print the opcode suffix for an 80-bit extended real XFmode float opcode.
;; 'J' Print the appropriate jump operand.
;;
;; 'b' Print the QImode name of the register for the indicated operand.
;;     %b0 would print %al if operands[0] is reg 0.
;; 'w' Likewise, print the HImode name of the register.
;; 'k' Likewise, print the SImode name of the register.
;; 'h' Print the QImode name for a "high" register, either ah, bh, ch or dh.
;; 'y' Print "st(0)" instead of "st" as a register.
```

1.1 UNSPEC usage

```
(define_constants
  [; Relocation specifiers
   (UNSPEC_GOT 0)
   (UNSPEC_GOTOFF 1)
   (UNSPEC_GOTPCREL 2)
   (UNSPEC_GOTTPOFF 3)
   (UNSPEC_TPOFF 4)
   (UNSPEC_NTPOFF 5)
   (UNSPEC_DTPOFF 6)
   (UNSPEC_GOTNTPOFF 7)
   (UNSPEC_INDNTPOFF 8)

   ; Prologue support
   (UNSPEC_STACK_ALLOC 11)
   (UNSPEC_SET_GOT 12)
   (UNSPEC_SSE_PROLOGUE_SAVE 13)
```

```
; TLS support
(UNSPEC_TP 15)
(UNSPEC_TLS_GD 16)
(UNSPEC_TLS_LD_BASE 17)

; Other random patterns
(UNSPEC_SCAS 20)
(UNSPEC_FNSTSW 21)
(UNSPEC_SAHF 22)
(UNSPEC_FSTCW 23)
(UNSPEC_ADD_CARRY 24)
(UNSPEC_FLDCW 25)
(UNSPEC_REP 26)
(UNSPEC_EH_RETURN 27)

; For SSE/MMX support:
(UNSPEC_FIX_NOTRUNC 30)
(UNSPEC_MASKMOV 31)
(UNSPEC_MOVMSK 32)
(UNSPEC_MOVNT 33)
(UNSPEC_MOVU 34)
(UNSPEC_RCP 35)
(UNSPEC_RSQRT 36)
(UNSPEC_SFENCE 37)
(UNSPEC_NOP 38) ; prevents combiner cleverness
(UNSPEC_PFRCP 39)
(UNSPEC_PFRCPIT1 40)
(UNSPEC_PFRCPIT2 41)
(UNSPEC_PFRSQRT 42)
(UNSPEC_PFRSQIT1 43)
(UNSPEC_MFENCE 44)
(UNSPEC_LFENCE 45)
(UNSPEC_PSADBW 46)
(UNSPEC_LDQQU 47)

; Generic math support
(UNSPEC_COPYSIGN 50)
(UNSPEC_IEEE_MIN 51) ; not commutative
(UNSPEC_IEEE_MAX 52) ; not commutative

; x87 Floating point
(UNSPEC_SIN 60)
(UNSPEC_COS 61)
(UNSPEC_FPATAN 62)
(UNSPEC_FYL2X 63)
(UNSPEC_FYL2XP1 64)
(UNSPEC_FRNDINT 65)
(UNSPEC_FIST 66)
(UNSPEC_F2XM1 67)
```

```
   ; x87 Rounding
   (UNSPEC_FRNDINT_FLOOR 70)
   (UNSPEC_FRNDINT_CEIL  71)
   (UNSPEC_FRNDINT_TRUNC 72)
   (UNSPEC_FRNDINT_MASK_PM 73)
   (UNSPEC_FIST_FLOOR 74)
   (UNSPEC_FIST_CEIL  75)

   ; x87 Double output FP
   (UNSPEC_SINCOS_COS 80)
   (UNSPEC_SINCOS_SIN 81)
   (UNSPEC_TAN_ONE 82)
   (UNSPEC_TAN_TAN 83)
   (UNSPEC_XTRACT_FRACT 84)
   (UNSPEC_XTRACT_EXP 85)
   (UNSPEC_FSCALE_FRACT 86)
   (UNSPEC_FSCALE_EXP 87)
   (UNSPEC_FPREM_F 88)
   (UNSPEC_FPREM_U 89)
   (UNSPEC_FPREM1_F 90)
   (UNSPEC_FPREM1_U 91)

   ; SSP patterns
   (UNSPEC_SP_SET 100)
   (UNSPEC_SP_TEST 101)
   (UNSPEC_SP_TLS_SET 102)
   (UNSPEC_SP_TLS_TEST 103)
  ])

(define_constants
  [(UNSPECV_BLOCKAGE 0)
   (UNSPECV_STACK_PROBE 1)
   (UNSPECV_EMMS 2)
   (UNSPECV_LDMXCSR 3)
   (UNSPECV_STMXCSR 4)
   (UNSPECV_FEMMS 5)
   (UNSPECV_CLFLUSH 6)
   (UNSPECV_ALIGN 7)
   (UNSPECV_MONITOR 8)
   (UNSPECV_MWAIT 9)
   (UNSPECV_CMPXCHG_1 10)
   (UNSPECV_CMPXCHG_2 11)
   (UNSPECV_XCHG 12)
   (UNSPECV_LOCK 13)
  ])

;; Registers by name.
(define_constants
  [(BP_REG  6)
   (SP_REG  7)
```

```
   (FLAGS_REG 17)
   (FPSR_REG 18)
   (DIRFLAG_REG 19)
  ])

;; Insns whose names begin with "x86_" are emitted by gen_FOO calls
;; from i386.c.

;; In C guard expressions, put expressions which may be compile-time
;; constants first.  This allows for better optimization.  For
;; example, write "TARGET_64BIT && reload_completed", not
;; "reload_completed && TARGET_64BIT".


;; Processor type.  This attribute must exactly match the processor_type
;; enumeration in i386.h.
(define_attr "cpu" "i386,i486,pentium,pentiumpro,k6,athlon,pentium4,k8,nocona"
  (const (symbol_ref "ix86_tune")))

;; A basic instruction type.  Refinements due to arguments to be
;; provided in other attributes.
(define_attr "type"
  "other,multi,
   alu,alu1,negnot,imov,imovx,lea,
   incdec,ishift,ishift1,rotate,rotate1,imul,idiv,
   icmp,test,ibr,setcc,icmov,
   push,pop,call,callv,leave,
   str,cld,
   fmov,fop,fsgn,fmul,fdiv,fpspc,fcmov,fcmp,fxch,fistp,fisttp,frndint,
   sselog,sselog1,sseiadd,sseishft,sseimul,
   sse,ssemov,sseadd,ssemul,ssecmp,ssecomi,ssecvt,sseicvt,ssediv,
   mmx,mmxmov,mmxadd,mmxmul,mmxcmp,mmxcvt,mmxshft"
  (const_string "other"))

;; Main data type used by the insn
(define_attr "mode"
  "unknown,none,QI,HI,SI,DI,SF,DF,XF,TI,V4SF,V2DF,V2SF,V1DF"
  (const_string "unknown"))

;; The CPU unit operations uses.
(define_attr "unit" "integer,i387,sse,mmx,unknown"
  (cond [(eq_attr "type" "fmov,fop,fsgn,fmul,fdiv,fpspc,fcmov,fcmp,fxch,fistp,fisttp,frndint")
   (const_string "i387")
 (eq_attr "type" "sselog,sselog1,sseiadd,sseishft,sseimul,
  sse,ssemov,sseadd,ssemul,ssecmp,ssecomi,ssecvt,sseicvt,ssediv")
   (const_string "sse")
 (eq_attr "type" "mmx,mmxmov,mmxadd,mmxmul,mmxcmp,mmxcvt,mmxshft")
   (const_string "mmx")
 (eq_attr "type" "other")
   (const_string "unknown")]
```

```
 (const_string "integer")))

;; The (bounding maximum) length of an instruction immediate.
(define_attr "length_immediate" ""
  (cond [(eq_attr "type" "incdec,setcc,icmov,str,cld,lea,other,multi,idiv,leave")
   (const_int 0)
 (eq_attr "unit" "i387,sse,mmx")
   (const_int 0)
 (eq_attr "type" "alu,alu1,negnot,imovx,ishift,rotate,ishift1,rotate1,
  imul,icmp,push,pop")
   (symbol_ref "ix86_attr_length_immediate_default(insn,1)")
 (eq_attr "type" "imov,test")
   (symbol_ref "ix86_attr_length_immediate_default(insn,0)")
 (eq_attr "type" "call")
   (if_then_else (match_operand 0 "constant_call_address_operand" "")
     (const_int 4)
     (const_int 0))
 (eq_attr "type" "callv")
   (if_then_else (match_operand 1 "constant_call_address_operand" "")
     (const_int 4)
     (const_int 0))
 ;; We don't know the size before shorten_branches.  Expect
 ;; the instruction to fit for better scheduling.
 (eq_attr "type" "ibr")
   (const_int 1)
 ]
 (symbol_ref "/* Update immediate_length and other attributes! */
      gcc_unreachable (),1")))

;; The (bounding maximum) length of an instruction address.
(define_attr "length_address" ""
  (cond [(eq_attr "type" "str,cld,other,multi,fxch")
   (const_int 0)
 (and (eq_attr "type" "call")
      (match_operand 0 "constant_call_address_operand" ""))
     (const_int 0)
 (and (eq_attr "type" "callv")
      (match_operand 1 "constant_call_address_operand" ""))
     (const_int 0)
 ]
 (symbol_ref "ix86_attr_length_address_default (insn)")))

;; Set when length prefix is used.
(define_attr "prefix_data16" ""
  (if_then_else (ior (eq_attr "mode" "HI")
     (and (eq_attr "unit" "sse") (eq_attr "mode" "V2DF")))
    (const_int 1)
    (const_int 0)))

;; Set when string REP prefix is used.
```

```
(define_attr "prefix_rep" ""
  (if_then_else (and (eq_attr "unit" "sse") (eq_attr "mode" "SF,DF"))
    (const_int 1)
    (const_int 0)))

;; Set when 0f opcode prefix is used.
(define_attr "prefix_0f" ""
  (if_then_else
    (ior (eq_attr "type" "imovx,setcc,icmov")
 (eq_attr "unit" "sse,mmx"))
    (const_int 1)
    (const_int 0)))

;; Set when REX opcode prefix is used.
(define_attr "prefix_rex" ""
  (cond [(and (eq_attr "mode" "DI")
         (eq_attr "type" "!push,pop,call,callv,leave,ibr"))
   (const_int 1)
 (and (eq_attr "mode" "QI")
      (ne (symbol_ref "x86_extended_QIreg_mentioned_p (insn)")
  (const_int 0)))
   (const_int 1)
 (ne (symbol_ref "x86_extended_reg_mentioned_p (insn)")
     (const_int 0))
   (const_int 1)
]
(const_int 0)))

;; Set when modrm byte is used.
(define_attr "modrm" ""
  (cond [(eq_attr "type" "str,cld,leave")
   (const_int 0)
 (eq_attr "unit" "i387")
   (const_int 0)
         (and (eq_attr "type" "incdec")
      (ior (match_operand:SI 1 "register_operand" "")
   (match_operand:HI 1 "register_operand" "")))
   (const_int 0)
 (and (eq_attr "type" "push")
      (not (match_operand 1 "memory_operand" "")))
   (const_int 0)
 (and (eq_attr "type" "pop")
      (not (match_operand 0 "memory_operand" "")))
   (const_int 0)
 (and (eq_attr "type" "imov")
      (and (match_operand 0 "register_operand" "")
           (match_operand 1 "immediate_operand" "")))
   (const_int 0)
 (and (eq_attr "type" "call")
      (match_operand 0 "constant_call_address_operand" ""))
```

```
    (const_int 0)
 (and (eq_attr "type" "callv")
      (match_operand 1 "constant_call_address_operand" ""))
    (const_int 0)
 ]
 (const_int 1)))

;; The (bounding maximum) length of an instruction in bytes.
;; ??? fistp and frndint are in fact fldcw/{fistp,frndint}/fldcw sequences.
;; Later we may want to split them and compute proper length as for
;; other insns.
(define_attr "length" ""
  (cond [(eq_attr "type" "other,multi,fistp,frndint")
   (const_int 16)
 (eq_attr "type" "fcmp")
   (const_int 4)
 (eq_attr "unit" "i387")
   (plus (const_int 2)
 (plus (attr "prefix_data16")
       (attr "length_address")))]
 (plus (plus (attr "modrm")
     (plus (attr "prefix_0f")
   (plus (attr "prefix_rex")
 (const_int 1))))
       (plus (attr "prefix_rep")
     (plus (attr "prefix_data16")
   (plus (attr "length_immediate")
 (attr "length_address")))))))

;; The ‘memory’ attribute is ‘none’ if no memory is referenced, ‘load’ or
;; ‘store’ if there is a simple memory reference therein, or ‘unknown’
;; if the instruction is complex.

(define_attr "memory" "none,load,store,both,unknown"
  (cond [(eq_attr "type" "other,multi,str")
   (const_string "unknown")
 (eq_attr "type" "lea,fcmov,fpspc,cld")
   (const_string "none")
 (eq_attr "type" "fistp,leave")
   (const_string "both")
 (eq_attr "type" "frndint")
   (const_string "load")
 (eq_attr "type" "push")
   (if_then_else (match_operand 1 "memory_operand" "")
     (const_string "both")
     (const_string "store"))
 (eq_attr "type" "pop")
   (if_then_else (match_operand 0 "memory_operand" "")
     (const_string "both")
     (const_string "load"))
```

```
 (eq_attr "type" "setcc")
   (if_then_else (match_operand 0 "memory_operand" "")
     (const_string "store")
     (const_string "none"))
 (eq_attr "type" "icmp,test,ssecmp,ssecomi,mmxcmp,fcmp")
   (if_then_else (ior (match_operand 0 "memory_operand" "")
      (match_operand 1 "memory_operand" ""))
     (const_string "load")
     (const_string "none"))
 (eq_attr "type" "ibr")
   (if_then_else (match_operand 0 "memory_operand" "")
     (const_string "load")
     (const_string "none"))
 (eq_attr "type" "call")
   (if_then_else (match_operand 0 "constant_call_address_operand" "")
     (const_string "none")
     (const_string "load"))
 (eq_attr "type" "callv")
   (if_then_else (match_operand 1 "constant_call_address_operand" "")
     (const_string "none")
     (const_string "load"))
 (and (eq_attr "type" "alu1,negnot,ishift1,sselog1")
      (match_operand 1 "memory_operand" ""))
   (const_string "both")
 (and (match_operand 0 "memory_operand" "")
      (match_operand 1 "memory_operand" ""))
   (const_string "both")
 (match_operand 0 "memory_operand" "")
   (const_string "store")
 (match_operand 1 "memory_operand" "")
   (const_string "load")
 (and (eq_attr "type"
 "!alu1,negnot,ishift1,
   imov,imovx,icmp,test,
   fmov,fcmp,fsgn,
   sse,ssemov,ssecmp,ssecomi,ssecvt,sseicvt,sselog1,
   mmx,mmxmov,mmxcmp,mmxcvt")
      (match_operand 2 "memory_operand" ""))
   (const_string "load")
 (and (eq_attr "type" "icmov")
      (match_operand 3 "memory_operand" ""))
   (const_string "load")
]
(const_string "none")))

;; Indicates if an instruction has both an immediate and a displacement.

(define_attr "imm_disp" "false,true,unknown"
  (cond [(eq_attr "type" "other,multi")
   (const_string "unknown")
```

```
 (and (eq_attr "type" "icmp,test,imov,alu1,ishift1,rotate1")
      (and (match_operand 0 "memory_displacement_operand" "")
   (match_operand 1 "immediate_operand" "")))
   (const_string "true")
 (and (eq_attr "type" "alu,ishift,rotate,imul,idiv")
      (and (match_operand 0 "memory_displacement_operand" "")
   (match_operand 2 "immediate_operand" "")))
   (const_string "true")
]
(const_string "false")))

;; Indicates if an FP operation has an integer source.

(define_attr "fp_int_src" "false,true"
  (const_string "false"))

;; Defines rounding mode of an FP operation.

(define_attr "i387_cw" "trunc,floor,ceil,mask_pm,uninitialized,any"
  (const_string "any"))

;; Describe a user's asm statement.
(define_asm_attributes
  [(set_attr "length" "128")
   (set_attr "type" "multi")])

;; All x87 floating point modes
(define_mode_macro X87MODEF [SF DF XF])

;; All integer modes handled by x87 fisttp operator.
(define_mode_macro X87MODEI [HI SI DI])

;; All integer modes handled by integer x87 operators.
(define_mode_macro X87MODEI12 [HI SI])

;; All SSE floating point modes
(define_mode_macro SSEMODEF [SF DF])

;; All integer modes handled by SSE cvtts?2si* operators.
(define_mode_macro SSEMODEI24 [SI DI])


;; Scheduling descriptions
```

1.2 Pentium Scheduling

;;(include "pentium.md")

```
;; The Pentium is an in-order core with two integer pipelines.
```

```
;; True for insns that behave like prefixed insns on the Pentium.
(define_attr "pent_prefix" "false,true"
  (if_then_else (ior (eq_attr "prefix_0f" "1")
        (ior (eq_attr "prefix_data16" "1")
  (eq_attr "prefix_rep" "1")))
    (const_string "true")
    (const_string "false")))

;; Categorize how an instruction slots.

;; The non-MMX Pentium slots an instruction with prefixes on U pipe only,
;; while MMX Pentium can slot it on either U or V.  Model non-MMX Pentium
;; rules, because it results in noticeably better code on non-MMX Pentium
;; and doesn't hurt much on MMX.  (Prefixed instructions are not very
;; common, so the scheduler usually has a non-prefixed insn to pair).

(define_attr "pent_pair" "uv,pu,pv,np"
  (cond [(eq_attr "imm_disp" "true")
   (const_string "np")
 (ior (eq_attr "type" "alu1,alu,imov,icmp,test,lea,incdec")
      (and (eq_attr "type" "pop,push")
   (eq_attr "memory" "!both")))
   (if_then_else (eq_attr "pent_prefix" "true")
     (const_string "pu")
     (const_string "uv"))
 (eq_attr "type" "ibr")
   (const_string "pv")
 (and (eq_attr "type" "ishift")
      (match_operand 2 "const_int_operand" ""))
   (const_string "pu")
 (and (eq_attr "type" "rotate")
      (match_operand 2 "const1_operand" ""))
   (const_string "pu")
 (and (eq_attr "type" "ishift1")
      (match_operand 1 "const_int_operand" ""))
   (const_string "pu")
 (and (eq_attr "type" "rotate1")
      (match_operand 1 "const1_operand" ""))
   (const_string "pu")
 (and (eq_attr "type" "call")
      (match_operand 0 "constant_call_address_operand" ""))
   (const_string "pv")
 (and (eq_attr "type" "callv")
      (match_operand 1 "constant_call_address_operand" ""))
   (const_string "pv")
]
(const_string "np")))

(define_automaton "pentium,pentium_fpu")
```

```
;; Pentium do have U and V pipes.  Instruction to both pipes
;; are always issued together, much like on VLIW.
;;
;;                    predecode
;;                   /         \
;;               decodeu     decodev
;;             /    |           |
;;           fpu executeu    executev
;;            |     |           |
;;           fpu  retire     retire
;;            |
;;           fpu
;; We add dummy "port" pipes allocated only first cycle of
;; instruction to specify this behavior.

(define_cpu_unit "pentium-portu,pentium-portv" "pentium")
(define_cpu_unit "pentium-u,pentium-v" "pentium")
(absence_set "pentium-portu" "pentium-u,pentium-v")
(presence_set "pentium-portv" "pentium-portu")

;; Floating point instructions can overlap with new issue of integer
;; instructions.  We model only first cycle of FP pipeline, as it is
;; fully pipelined.
(define_cpu_unit "pentium-fp" "pentium_fpu")

;; There is non-pipelined multiplier unit used for complex operations.
(define_cpu_unit "pentium-fmul" "pentium_fpu")

;; Pentium preserves memory ordering, so when load-execute-store
;; instruction is executed together with other instruction loading
;; data, the execution of the other instruction is delayed to very
;; last cycle of first instruction, when data are bypassed.
;; We model this by allocating "memory" unit when store is pending
;; and using conflicting load units together.

(define_cpu_unit "pentium-memory" "pentium")
(define_cpu_unit "pentium-load0" "pentium")
(define_cpu_unit "pentium-load1" "pentium")
(absence_set "pentium-load0,pentium-load1" "pentium-memory")

(define_reservation "pentium-load" "(pentium-load0 | pentium-load1)")
(define_reservation "pentium-np" "(pentium-u + pentium-v)")
(define_reservation "pentium-uv" "(pentium-u | pentium-v)")
(define_reservation "pentium-portuv" "(pentium-portu | pentium-portv)")
(define_reservation "pentium-firstu" "(pentium-u + pentium-portu)")
(define_reservation "pentium-firstv" "(pentium-v + pentium-portuv)")
(define_reservation "pentium-firstuv" "(pentium-uv + pentium-portuv)")
(define_reservation "pentium-firstuload" "(pentium-load + pentium-firstu)")
(define_reservation "pentium-firstvload" "(pentium-load + pentium-firstv)")
```

```
(define_reservation "pentium-firstuvload" "(pentium-load + pentium-firstuv)
   | (pentium-firstv,pentium-v,
      (pentium-load+pentium-firstv))")
(define_reservation "pentium-firstuboth" "(pentium-load + pentium-firstu
   + pentium-memory)")
(define_reservation "pentium-firstvboth" "(pentium-load + pentium-firstv
   + pentium-memory)")
(define_reservation "pentium-firstuvboth" "(pentium-load + pentium-firstuv
    + pentium-memory)
   | (pentium-firstv,pentium-v,
      (pentium-load+pentium-firstv))")

;; Few common long latency instructions
(define_insn_reservation "pent_mul" 11
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "imul"))
  "pentium-np*11")

(define_insn_reservation "pent_str" 12
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "str"))
  "pentium-np*12")

;; Integer division and some other long latency instruction block all
;; units, including the FP pipe.  There is no value in modeling the
;; latency of these instructions and not modeling the latency
;; decreases the size of the DFA.
(define_insn_reservation "pent_block" 1
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "idiv"))
  "pentium-np+pentium-fp")

(define_insn_reservation "pent_cld" 2
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "cld"))
  "pentium-np*2")

;;  Moves usually have one cycle penalty, but there are exceptions.
(define_insn_reservation "pent_fmov" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "type" "fmov")
    (eq_attr "memory" "none,load")))
  "(pentium-fp+pentium-np)")

(define_insn_reservation "pent_fpmovxf" 3
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "type" "fmov")
    (and (eq_attr "memory" "load,store")
 (eq_attr "mode" "XF"))))
  "(pentium-fp+pentium-np)*3")
```

```
(define_insn_reservation "pent_fpstore" 2
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "type" "fmov")
    (ior (match_operand 1 "immediate_operand" "")
 (eq_attr "memory" "store"))))
  "(pentium-fp+pentium-np)*2")

(define_insn_reservation "pent_imov" 1
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "imov"))
  "pentium-firstuv")

;; Push and pop instructions have 1 cycle latency and special
;; hardware bypass allows them to be paired with other push,pop
;; and call instructions.
(define_bypass 0 "pent_push,pent_pop" "pent_push,pent_pop,pent_call")
(define_insn_reservation "pent_push" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "type" "push")
    (eq_attr "memory" "store")))
  "pentium-firstuv")

(define_insn_reservation "pent_pop" 1
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "pop,leave"))
  "pentium-firstuv")

;; Call and branch instruction can execute in either pipe, but
;; they are only pairable when in the v pipe.
(define_insn_reservation "pent_call" 10
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "call,callv"))
  "pentium-firstv,pentium-v*9")

(define_insn_reservation "pent_branch" 1
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "ibr"))
  "pentium-firstv")

;; Floating point instruction dispatch in U pipe, but continue
;; in FP pipeline allowing other instructions to be executed.
(define_insn_reservation "pent_fp" 3
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "fop,fistp"))
  "(pentium-firstu+pentium-fp),nothing,nothing")

;; First two cycles of fmul are not pipelined.
(define_insn_reservation "pent_fmul" 3
  (and (eq_attr "cpu" "pentium")
```

```
       (eq_attr "type" "fmul"))
  "(pentium-firstuv+pentium-fp+pentium-fmul),pentium-fmul,nothing")

;; Long latency FP instructions overlap with integer instructions,
;; but only last 2 cycles with FP ones.
(define_insn_reservation "pent_fdiv" 39
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "fdiv"))
  "(pentium-np+pentium-fp+pentium-fmul),
   (pentium-fp+pentium-fmul)*36,pentium-fmul*2")

(define_insn_reservation "pent_fpspc" 70
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "fpspc"))
  "(pentium-np+pentium-fp+pentium-fmul),
   (pentium-fp+pentium-fmul)*67,pentium-fmul*2")

;; Integer instructions.  Load/execute/store takes 3 cycles,
;; load/execute 2 cycles and execute only one cycle.
(define_insn_reservation "pent_uv_both" 3
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "uv")
    (eq_attr "memory" "both")))
  "pentium-firstuvboth,pentium-uv+pentium-memory,pentium-uv")

(define_insn_reservation "pent_u_both" 3
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pu")
    (eq_attr "memory" "both")))
  "pentium-firstuboth,pentium-u+pentium-memory,pentium-u")

(define_insn_reservation "pent_v_both" 3
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pv")
    (eq_attr "memory" "both")))
  "pentium-firstvboth,pentium-v+pentium-memory,pentium-v")

(define_insn_reservation "pent_np_both" 3
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "np")
    (eq_attr "memory" "both")))
  "pentium-np,pentium-np,pentium-np")

(define_insn_reservation "pent_uv_load" 2
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "uv")
    (eq_attr "memory" "load")))
  "pentium-firstuvload,pentium-uv")

(define_insn_reservation "pent_u_load" 2
```

```
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pu")
    (eq_attr "memory" "load")))
  "pentium-firstuload,pentium-u")

(define_insn_reservation "pent_v_load" 2
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pv")
    (eq_attr "memory" "load")))
  "pentium-firstvload,pentium-v")

(define_insn_reservation "pent_np_load" 2
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "np")
    (eq_attr "memory" "load")))
  "pentium-np,pentium-np")

(define_insn_reservation "pent_uv" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "uv")
    (eq_attr "memory" "none")))
  "pentium-firstuv")

(define_insn_reservation "pent_u" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pu")
    (eq_attr "memory" "none")))
  "pentium-firstu")

(define_insn_reservation "pent_v" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "pv")
    (eq_attr "memory" "none")))
  "pentium-firstv")

(define_insn_reservation "pent_np" 1
  (and (eq_attr "cpu" "pentium")
       (and (eq_attr "pent_pair" "np")
    (eq_attr "memory" "none")))
  "pentium-np")
```

1.3 Scheduling for the Intel P6 family of processors

;;(include "ppro.md")

```
;; The P6 family includes the Pentium Pro, Pentium II, Pentium III, Celeron
;; and Xeon lines of CPUs.  The DFA scheduler description in this file is
;; based on information that can be found in the following three documents:
;;
```

```
;;    "P6 Family of Processors Hardware Developer's Manual",
;;    Intel, September 1999.
;;
;;    "Intel Architecture Optimization Manual",
;;    Intel, 1999 (Order Number: 245127-001).
;;
;;    "How to optimize for the Pentium family of microprocessors",
;;    by Agner Fog, PhD.
;;
;; The P6 pipeline has three major components:
;;   1) the FETCH/DECODE unit, an in-order issue front-end
;;   2) the DISPATCH/EXECUTE unit, which is the out-of-order core
;;   3) the RETIRE unit, an in-order retirement unit
;;
;; So, the P6 CPUs have out-of-order cores, but the instruction decoder and
;; retirement unit are naturally in-order.
;;
;;                       BUS INTERFACE UNIT
;;                      /                  \
;;                L1 ICACHE             L1 DCACHE
;;              /     |     \              |     \
;;       DECODER0  DECODER1  DECODER2  DISP/EXEC  RETIRE
;;              \     |     /              |        |
;;            INSTRUCTION POOL   __________|_______/
;;          (inc. reorder buffer)
;;
;; Since the P6 CPUs execute instructions out-of-order, the most important
;; consideration in performance tuning is making sure enough micro-ops are
;; ready for execution in the out-of-order core, while not stalling the
;; decoder.
;;
;; TODO:
;; - Find a less crude way to model complex instructions, in
;;   particular how many cycles they take to be decoded.
;; - Include decoder latencies in the total reservation latencies.
;;   This isn't necessary right now because we assume for every
;;   instruction that it never blocks a decoder.
;; - Figure out where the p0 and p1 reservations come from.  These
;;   appear not to be in the manual (e.g. why is cld "(p0+p1)*2"
;;   better than "(p0|p1)*4" ???)
;; - Lots more because I'm sure this is still far from optimal :-)

;; The ppro_idiv and ppro_fdiv automata are used to model issue
;; latencies of idiv and fdiv type insns.
(define_automaton "ppro_decoder,ppro_core,ppro_idiv,ppro_fdiv,ppro_load,ppro_store")

;; Simple instructions of the register-register form have only one uop.
;; Load instructions are also only one uop.  Store instructions decode to
;; two uops, and simple read-modify instructions also take two uops.
;; Simple instructions of the register-memory form have two to three uops.
```

```
;; Simple read-modify-write instructions have four uops.  The rules for
;; the decoder are simple:
;;  - an instruction with 1 uop can be decoded by any of the three
;;    decoders in one cycle.
;;  - an instruction with 1 to 4 uops can be decoded only by decoder 0
;;    but still in only one cycle.
;;  - a complex (microcode) instruction can also only be decoded by
;;    decoder 0, and this takes an unspecified number of cycles.
;;
;; The goal is to schedule such that we have a few-one-one uops sequence
;; in each cycle, to decode as many instructions per cycle as possible.
(define_cpu_unit "decoder0" "ppro_decoder")
(define_cpu_unit "decoder1" "ppro_decoder")
(define_cpu_unit "decoder2" "ppro_decoder")

;; We first wish to find an instruction for decoder0, so exclude
;; decoder1 and decoder2 from being reserved until decoder 0 is
;; reserved.
(presence_set "decoder1" "decoder0")
(presence_set "decoder2" "decoder0")

;; Most instructions can be decoded on any of the three decoders.
(define_reservation "decodern" "(decoder0|decoder1|decoder2)")

;; The out-of-order core has five pipelines.  During each cycle, the core
;; may dispatch zero or one uop on the port of any of the five pipelines
;; so the maximum number of dispatched uops per cycle is 5.  In practicer,
;; 3 uops per cycle is more realistic.
;;
;; Two of the five pipelines contain several execution units:
;;
;; Port 0 Port 1 Port 2 Port 3 Port 4
;; ALU ALU LOAD SAC SDA
;; FPU JUE
;; AGU MMX
;; MMX P3FPU
;; P3FPU
;;
;; (SAC=Store Address Calculation, SDA=Store Data Unit, P3FPU = SSE unit,
;;  JUE = Jump Execution Unit, AGU = Address Generation Unit)
;;
(define_cpu_unit "p0,p1" "ppro_core")
(define_cpu_unit "p2" "ppro_load")
(define_cpu_unit "p3,p4" "ppro_store")
(define_cpu_unit "idiv" "ppro_idiv")
(define_cpu_unit "fdiv" "ppro_fdiv")

;; Only the irregular instructions have to be modeled here.  A load
;; increases the latency by 2 or 3, or by nothing if the manual gives
;; a latency already.  Store latencies are not accounted for.
```

```
;;
;; The simple instructions follow a very regular pattern of 1 uop per
;; reg-reg operation, 1 uop per load on port 2. and 2 uops per store
;; on port 4 and port 3.  These instructions are modelled at the bottom
;; of this file.
;;
;; For microcoded instructions we don't know how many uops are produced.
;; These instructions are the "complex" ones in the Intel manuals.  All
;; we _do_ know is that they typically produce four or more uops, so
;; they can only be decoded on decoder0.  Modelling their latencies
;; doesn't make sense because we don't know how these instructions are
;; executed in the core.  So we just model that they can only be decoded
;; on decoder 0, and say that it takes a little while before the result
;; is available.
(define_insn_reservation "ppro_complex_insn" 6
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "other,multi,call,callv,str"))
 "decoder0")

;; imov with memory operands does not use the integer units.
(define_insn_reservation "ppro_imov" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "imov")))
 "decodern,(p0|p1)")

(define_insn_reservation "ppro_imov_load" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "imov")))
 "decodern,p2")

(define_insn_reservation "ppro_imov_store" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
   (eq_attr "type" "imov")))
 "decoder0,p4+p3")

;; imovx always decodes to one uop, and also doesn't use the integer
;; units if it has memory operands.
(define_insn_reservation "ppro_imovx" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "imovx")))
 "decodern,(p0|p1)")

(define_insn_reservation "ppro_imovx_load" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "imovx")))
```

```
 "decodern,p2")

;; lea executes on port 0 with latency one and throughput 1.
(define_insn_reservation "ppro_lea" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "lea")))
 "decodern,p0")

;; Shift and rotate execute on port 0 with latency and throughput 1.
;; The load and store units need to be reserved when memory operands
;; are involved.
(define_insn_reservation "ppro_shift_rotate" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "ishift,ishift1,rotate,rotate1")))
 "decodern,p0")

(define_insn_reservation "ppro_shift_rotate_mem" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "!none")
   (eq_attr "type" "ishift,ishift1,rotate,rotate1")))
 "decoder0,p2+p0,p4+p3")

(define_insn_reservation "ppro_cld" 2
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "cld"))
 "decoder0,(p0+p1)*2")

;; The P6 has a sophisticated branch prediction mechanism to minimize
;; latencies due to branching.  In particular, it has a fast way to
;; execute branches that are taken multiple times (such as in loops).
;; Branches not taken suffer no penalty, and correctly predicted
;; branches cost only one fetch cycle.  Mispredicted branches are very
;; costly: typically 15 cycles and possibly as many as 26 cycles.
;;
;; Unfortunately all this makes it quite difficult to properly model
;; the latencies for the compiler.  Here I've made the choice to be
;; optimistic and assume branches are often predicted correctly, so
;; they have latency 1, and the decoders are not blocked.
;;
;; In addition, the model assumes a branch always decodes to only 1 uop,
;; which is not exactly true because there are a few instructions that
;; decode to 2 uops or microcode.  But this probably gives the best
;; results because we can assume these instructions can decode on all
;; decoders.
(define_insn_reservation "ppro_branch" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "ibr")))
```

```
 "decodern,p1")

;; ??? Indirect branches probably have worse latency than this.
(define_insn_reservation "ppro_indirect_branch" 6
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "!none")
   (eq_attr "type" "ibr")))
 "decoder0,p2+p1")

(define_insn_reservation "ppro_leave" 4
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "leave"))
 "decoder0,p2+(p0|p1),(p0|p1)")

;; imul has throughput one, but latency 4, and can only execute on port 0.
(define_insn_reservation "ppro_imul" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "imul")))
 "decodern,p0")

(define_insn_reservation "ppro_imul_mem" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "!none")
   (eq_attr "type" "imul")))
 "decoder0,p2+p0")

;; div and idiv are very similar, so we model them the same.
;; QI, HI, and SI have issue latency 12, 21, and 37, respectively.
;; These issue latencies are modelled via the ppro_div automaton.
(define_insn_reservation "ppro_idiv_QI" 19
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "QI")
(eq_attr "type" "idiv"))))
 "decoder0,(p0+idiv)*2,(p0|p1)+idiv,idiv*9")

(define_insn_reservation "ppro_idiv_QI_load" 19
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "QI")
(eq_attr "type" "idiv"))))
 "decoder0,p2+p0+idiv,p0+idiv,(p0|p1)+idiv,idiv*9")

(define_insn_reservation "ppro_idiv_HI" 23
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "HI")
(eq_attr "type" "idiv"))))
 "decoder0,(p0+idiv)*3,(p0|p1)+idiv,idiv*17")
```

```
(define_insn_reservation "ppro_idiv_HI_load" 23
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "HI")
(eq_attr "type" "idiv"))))
 "decoder0,p2+p0+idiv,p0+idiv,(p0|p1)+idiv,idiv*18")

(define_insn_reservation "ppro_idiv_SI" 39
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SI")
(eq_attr "type" "idiv"))))
 "decoder0,(p0+idiv)*3,(p0|p1)+idiv,idiv*33")

(define_insn_reservation "ppro_idiv_SI_load" 39
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SI")
(eq_attr "type" "idiv"))))
 "decoder0,p2+p0+idiv,p0+idiv,(p0|p1)+idiv,idiv*34")

;; Floating point operations always execute on port 0.
;; ??? where do these latencies come from? fadd has latency 3 and
;;     has throughput "1/cycle (align with FADD)".  What do they
;;     mean and how can we model that?
(define_insn_reservation "ppro_fop" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none,unknown")
   (eq_attr "type" "fop")))
 "decodern,p0")

(define_insn_reservation "ppro_fop_load" 5
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "fop")))
 "decoder0,p2+p0,p0")

(define_insn_reservation "ppro_fop_store" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
   (eq_attr "type" "fop")))
 "decoder0,p0,p0,p0+p4+p3")

(define_insn_reservation "ppro_fop_both" 5
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "both")
   (eq_attr "type" "fop")))
 "decoder0,p2+p0,p0+p4+p3")
```

```
(define_insn_reservation "ppro_fsgn" 1
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "fsgn"))
 "decodern,p0")

(define_insn_reservation "ppro_fistp" 5
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "fistp"))
 "decoder0,p0*2,p4+p3")

(define_insn_reservation "ppro_fcmov" 2
 (and (eq_attr "cpu" "pentiumpro")
      (eq_attr "type" "fcmov"))
 "decoder0,p0*2")

(define_insn_reservation "ppro_fcmp" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "fcmp")))
 "decodern,p0")

(define_insn_reservation "ppro_fcmp_load" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "fcmp")))
 "decoder0,p2+p0")

(define_insn_reservation "ppro_fmov" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "fmov")))
 "decodern,p0")

(define_insn_reservation "ppro_fmov_load" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "!XF")
(eq_attr "type" "fmov"))))
 "decodern,p2")

(define_insn_reservation "ppro_fmov_XF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "XF")
(eq_attr "type" "fmov"))))
 "decoder0,(p2+p0)*2")

(define_insn_reservation "ppro_fmov_store" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
```

```
   (and (eq_attr "mode" "!XF")
(eq_attr "type" "fmov"))))
 "decodern,p0")

(define_insn_reservation "ppro_fmov_XF_store" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
   (and (eq_attr "mode" "XF")
(eq_attr "type" "fmov"))))
 "decoder0,(p0+p4),(p0+p3)")

;; fmul executes on port 0 with latency 5.  It has issue latency 2,
;; but we don't model this.
(define_insn_reservation "ppro_fmul" 5
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "fmul")))
 "decoder0,p0*2")

(define_insn_reservation "ppro_fmul_load" 6
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "fmul")))
 "decoder0,p2+p0,p0")

;; fdiv latencies depend on the mode of the operands.  XFmode gives
;; a latency of 38 cycles, DFmode gives 32, and SFmode gives latency 18.
;; Division by a power of 2 takes only 9 cycles, but we cannot model
;; that.  Throughput is equal to latency - 1, which we model using the
;; ppro_div automaton.
(define_insn_reservation "ppro_fdiv_SF" 18
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "fdiv,fpspc"))))
 "decodern,p0+fdiv,fdiv*16")

(define_insn_reservation "ppro_fdiv_SF_load" 19
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "fdiv,fpspc"))))
 "decoder0,p2+p0+fdiv,fdiv*16")

(define_insn_reservation "ppro_fdiv_DF" 32
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "DF")
(eq_attr "type" "fdiv,fpspc"))))
 "decodern,p0+fdiv,fdiv*30")
```

```
(define_insn_reservation "ppro_fdiv_DF_load" 33
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "DF")
(eq_attr "type" "fdiv,fpspc"))))
 "decoder0,p2+p0+fdiv,fdiv*30")

(define_insn_reservation "ppro_fdiv_XF" 38
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "XF")
(eq_attr "type" "fdiv,fpspc"))))
 "decodern,p0+fdiv,fdiv*36")

(define_insn_reservation "ppro_fdiv_XF_load" 39
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "XF")
(eq_attr "type" "fdiv,fpspc"))))
 "decoder0,p2+p0+fdiv,fdiv*36")

;; MMX instructions can execute on either port 0 or port 1 with a
;; throughput of 1/cycle.
;;   on port 0: - ALU (latency 1)
;; - Multiplier Unit (latency 3)
;;   on port 1: - ALU (latency 1)
;; - Shift Unit (latency 1)
;;
;; MMX instructions are either of the type reg-reg, or read-modify, and
;; except for mmxshft and mmxmul they can execute on port 0 or port 1,
;; so they behave as "simple" instructions that need no special modelling.
;; We only have to model mmxshft and mmxmul.
(define_insn_reservation "ppro_mmx_shft" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "mmxshft")))
 "decodern,p1")

(define_insn_reservation "ppro_mmx_shft_load" 2
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "mmxshft")))
 "decoder0,p2+p1")

(define_insn_reservation "ppro_mmx_mul" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "mmxmul")))
 "decodern,p0")
```

```
(define_insn_reservation "ppro_mmx_mul_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (eq_attr "type" "mmxmul")))
 "decoder0,p2+p0")

(define_insn_reservation "ppro_sse_mmxcvt" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "mode" "DI")
   (eq_attr "type" "mmxcvt")))
 "decodern,p1")

;; FIXME: These are Pentium III only, but we cannot tell here if
;; we're generating code for PentiumPro/Pentium II or Pentium III
;; (define_insn_reservation "ppro_sse_mmxshft" 2
;;  (and (eq_attr "cpu" "pentiumpro")
;;       (and (eq_attr "mode" "DI")
;;    (eq_attr "type" "mmxshft")))
;;  "decodern,p0")

;; SSE is very complicated, and takes a bit more effort.
;; ??? I assumed that all SSE instructions decode on decoder0,
;;     but is this correct?

;; The sfence instruction.
(define_insn_reservation "ppro_sse_sfence" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "unknown")
   (eq_attr "type" "sse")))
 "decoder0,p4+p3")

;; FIXME: This reservation is all wrong when we're scheduling sqrtss.
(define_insn_reservation "ppro_sse_SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "mode" "SF")
   (eq_attr "type" "sse")))
 "decodern,p0")

(define_insn_reservation "ppro_sse_add_SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "sseadd"))))
 "decodern,p1")

(define_insn_reservation "ppro_sse_add_SF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
```

```
(eq_attr "type" "sseadd"))))
 "decoder0,p2+p1")

(define_insn_reservation "ppro_sse_cmp_SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssecmp"))))
 "decoder0,p1")

(define_insn_reservation "ppro_sse_cmp_SF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssecmp"))))
 "decoder0,p2+p1")

(define_insn_reservation "ppro_sse_comi_SF" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssecomi"))))
 "decodern,p0")

(define_insn_reservation "ppro_sse_comi_SF_load" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssecomi"))))
 "decoder0,p2+p0")

(define_insn_reservation "ppro_sse_mul_SF" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssemul"))))
"decodern,p0")

(define_insn_reservation "ppro_sse_mul_SF_load" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssemul"))))
"decoder0,p2+p0")

;; FIXME: ssediv doesn't close p0 for 17 cycles, surely???
(define_insn_reservation "ppro_sse_div_SF" 18
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
```

```
(eq_attr "type" "ssediv"))))
 "decoder0,p0*17")

(define_insn_reservation "ppro_sse_div_SF_load" 18
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssediv"))))
 "decoder0,(p2+p0),p0*16")

(define_insn_reservation "ppro_sse_icvt_SF" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "mode" "SF")
   (eq_attr "type" "sseicvt")))
 "decoder0,(p2+p1)*2")

(define_insn_reservation "ppro_sse_icvt_SI" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "mode" "SI")
   (eq_attr "type" "sseicvt")))
 "decoder0,(p2+p1)")

(define_insn_reservation "ppro_sse_mov_SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssemov"))))
 "decoder0,(p0|p1)")

(define_insn_reservation "ppro_sse_mov_SF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssemov"))))
 "decoder0,p2+(p0|p1)")

(define_insn_reservation "ppro_sse_mov_SF_store" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
   (and (eq_attr "mode" "SF")
(eq_attr "type" "ssemov"))))
 "decoder0,p4+p3")

(define_insn_reservation "ppro_sse_V4SF" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "mode" "V4SF")
   (eq_attr "type" "sse")))
 "decoder0,p1*2")

(define_insn_reservation "ppro_sse_add_V4SF" 3
```

```
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "sseadd"))))
 "decoder0,p1*2")

(define_insn_reservation "ppro_sse_add_V4SF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "sseadd"))))
 "decoder0,(p2+p1)*2")

(define_insn_reservation "ppro_sse_cmp_V4SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssecmp"))))
 "decoder0,p1*2")

(define_insn_reservation "ppro_sse_cmp_V4SF_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssecmp"))))
 "decoder0,(p2+p1)*2")

(define_insn_reservation "ppro_sse_cvt_V4SF" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none,unknown")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssecvt"))))
 "decoder0,p1*2")

(define_insn_reservation "ppro_sse_cvt_V4SF_other" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "!none,unknown")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssecmp"))))
 "decoder0,p1,p4+p3")

(define_insn_reservation "ppro_sse_mul_V4SF" 5
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssemul"))))
"decoder0,p0*2")

(define_insn_reservation "ppro_sse_mul_V4SF_load" 5
 (and (eq_attr "cpu" "pentiumpro")
```

```
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssemul"))))
"decoder0,(p2+p0)*2")

;; FIXME: p0 really closed this long???
(define_insn_reservation "ppro_sse_div_V4SF" 48
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssediv"))))
 "decoder0,p0*34")

(define_insn_reservation "ppro_sse_div_V4SF_load" 48
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssediv"))))
 "decoder0,(p2+p0)*2,p0*32")

(define_insn_reservation "ppro_sse_log_V4SF" 2
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "sselog,sselog1"))))
 "decodern,p1")

(define_insn_reservation "ppro_sse_log_V4SF_load" 2
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "sselog,sselog1"))))
 "decoder0,(p2+p1)")

(define_insn_reservation "ppro_sse_mov_V4SF" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssemov"))))
 "decoder0,(p0|p1)*2")

(define_insn_reservation "ppro_sse_mov_V4SF_load" 2
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssemov"))))
 "decoder0,p2*2")

(define_insn_reservation "ppro_sse_mov_V4SF_store" 3
 (and (eq_attr "cpu" "pentiumpro")
```

```
      (and (eq_attr "memory" "store")
   (and (eq_attr "mode" "V4SF")
(eq_attr "type" "ssemov"))))
 "decoder0,(p4+p3)*2")

;; All other instructions are modelled as simple instructions.
;; We have already modelled all i387 floating point instructions, so all
;; other instructions execute on either port 0 or port 1.  This includes
;; the ALU units, and the MMX units.
;;
;; reg-reg instructions produce 1 uop so they can be decoded on any of
;; the three decoders.
(define_insn_reservation "ppro_insn" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "none,unknown")
   (eq_attr "type" "alu,alu1,negnot,incdec,icmp,test,setcc,icmov,push,pop,fxch,sseiadd,sseishft,sseimu
 "decodern,(p0|p1)")

;; read-modify and register-memory instructions have 2 or three uops,
;; so they have to be decoded on decoder0.
(define_insn_reservation "ppro_insn_load" 3
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "load")
   (eq_attr "type" "alu,alu1,negnot,incdec,icmp,test,setcc,icmov,push,pop,fxch,sseiadd,sseishft,sseimu
 "decoder0,p2+(p0|p1)")

(define_insn_reservation "ppro_insn_store" 1
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "store")
   (eq_attr "type" "alu,alu1,negnot,incdec,icmp,test,setcc,icmov,push,pop,fxch,sseiadd,sseishft,sseimu
 "decoder0,(p0|p1),p4+p3")

;; read-modify-store instructions produce 4 uops so they have to be
;; decoded on decoder0 as well.
(define_insn_reservation "ppro_insn_both" 4
 (and (eq_attr "cpu" "pentiumpro")
      (and (eq_attr "memory" "both")
   (eq_attr "type" "alu,alu1,negnot,incdec,icmp,test,setcc,icmov,push,pop,fxch,sseiadd,sseishft,sseimu
 "decoder0,p2+(p0|p1),p4+p3")
```

1.4 AMD K6/K6-2 Scheduling

;;(include "k6.md")

```
;; The K6 architecture is quite similar to PPro.  Important difference is
;; that there are only two decoders and they seems to be much slower than
;; any of the execution units.  So we have to pay much more attention to
;; proper scheduling for the decoders.
;; FIXME: We don't do that right now.  A good start would be to sort the
```

```
;;        instructions based on length.
;;
;; This description is based on data from the following documents:
;;
;;    "AMD-K6 Processor Data Sheet (Preliminary information)"
;;    Advanced Micro Devices, Inc., 1998.
;;
;;    "AMD-K6 Processor Code Optimization Application Note"
;;    Advanced Micro Devices, Inc., 2000.
;;
;; CPU execution units of the K6:
;;
;; store describes the Store unit.  This unit is not modelled
;; completely and it is only used to model lea operation.
;; Otherwise it lies outside of any critical path.
;; load describes the Load unit
;; alux describes the Integer X unit
;; mm describes the Multimedia unit, which shares a pipe
;; with the Integer X unit.  This unit is used for MMX,
;; which is not implemented for K6.
;; aluy describes the Integer Y unit
;; fpu describes the FPU unit
;; branch describes the Branch unit
;;
;; The fp unit is not pipelined, and it can only do one operation per two
;; cycles, including fxcg.
;;
;; Generally this is a very poor description, but at least no worse than
;; the old description, and a lot easier to extend to something more
;; reasonable if anyone still cares enough about this architecture in 2004.
;;
;; ??? fxch isn't handled; not an issue until sched3 after reg-stack is real.

(define_automaton "k6_decoder,k6_load_unit,k6_store_unit,k6_integer_units,k6_fpu_unit,k6_branch_unit")

;; The K6 instruction decoding begins before the on-chip instruction cache is
;; filled.  Depending on the length of the instruction, two simple instructions
;; can be decoded in two parallel short decoders, or one complex instruction can
;; be decoded in either the long or the vector decoder.  For all practical
;; purposes, the long and vector decoder can be modelled as one decoder.
(define_cpu_unit "k6_decode_short0" "k6_decoder")
(define_cpu_unit "k6_decode_short1" "k6_decoder")
(define_cpu_unit "k6_decode_long" "k6_decoder")
(exclusion_set "k6_decode_long" "k6_decode_short0,k6_decode_short1")
(define_reservation "k6_decode_short" "k6_decode_short0|k6_decode_short1")
(define_reservation "k6_decode_vector" "k6_decode_long")

(define_cpu_unit "k6_store" "k6_store_unit")
(define_cpu_unit "k6_load" "k6_load_unit")
(define_cpu_unit "k6_alux,k6_aluy" "k6_integer_units")
```

```
(define_cpu_unit "k6_fpu" "k6_fpu_unit")
(define_cpu_unit "k6_branch" "k6_branch_unit")

;; Shift instructions and certain arithmetic are issued only on Integer X.
(define_insn_reservation "k6_alux_only" 1
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "ishift,ishift1,rotate,rotate1,alu1,negnot,cld")
   (eq_attr "memory" "none")))
 "k6_decode_short,k6_alux")

(define_insn_reservation "k6_alux_only_load" 3
 (and (eq_attr "cpu" "k6")
       (and (eq_attr "type" "ishift,ishift1,rotate,rotate1,alu1,negnot,cld")
    (eq_attr "memory" "load")))
 "k6_decode_short,k6_load,k6_alux")

(define_insn_reservation "k6_alux_only_store" 3
 (and (eq_attr "cpu" "k6")
       (and (eq_attr "type" "ishift,ishift1,rotate,rotate1,alu1,negnot,cld")
    (eq_attr "memory" "store,both,unknown")))
 "k6_decode_long,k6_load,k6_alux,k6_store")

;; Integer divide and multiply can only be issued on Integer X, too.
(define_insn_reservation "k6_alu_imul" 2
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "imul"))
 "k6_decode_vector,k6_alux*3")

(define_insn_reservation "k6_alu_imul_load" 4
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "load")))
 "k6_decode_vector,k6_load,k6_alux*3")

(define_insn_reservation "k6_alu_imul_store" 4
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "store,both,unknown")))
 "k6_decode_vector,k6_load,k6_alux*3,k6_store")

;; ??? Guessed latencies based on the old pipeline description.
(define_insn_reservation "k6_alu_idiv" 17
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "idiv")
   (eq_attr "memory" "none")))
 "k6_decode_vector,k6_alux*17")

(define_insn_reservation "k6_alu_idiv_mem" 19
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "idiv")
```

```
   (eq_attr "memory" "!none")))
 "k6_decode_vector,k6_load,k6_alux*17")

;; Basic word and doubleword ALU ops can be issued on both Integer units.
(define_insn_reservation "k6_alu" 1
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "alu,alu1,negnot,icmp,test,imovx,incdec,setcc")
   (eq_attr "memory" "none")))
 "k6_decode_short,k6_alux|k6_aluy")

(define_insn_reservation "k6_alu_load" 3
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "alu,alu1,negnot,icmp,test,imovx,incdec,setcc")
   (eq_attr "memory" "load")))
 "k6_decode_short,k6_load,k6_alux|k6_aluy")

(define_insn_reservation "k6_alu_store" 3
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "alu,alu1,negnot,icmp,test,imovx,incdec,setcc")
   (eq_attr "memory" "store,both,unknown")))
 "k6_decode_long,k6_load,k6_alux|k6_aluy,k6_store")

;; A "load immediate" operation does not require execution at all,
;; it is available immediately after decoding.  Special-case this.
(define_insn_reservation "k6_alu_imov" 1
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imov")
   (and (eq_attr "memory" "none")
(match_operand 1 "nonimmediate_operand"))))
 "k6_decode_short,k6_alux|k6_aluy")

(define_insn_reservation "k6_alu_imov_imm" 0
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imov")
   (and (eq_attr "memory" "none")
(match_operand 1 "immediate_operand"))))
 "k6_decode_short")

(define_insn_reservation "k6_alu_imov_load" 2
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imov")
   (eq_attr "memory" "load")))
 "k6_decode_short,k6_load")

(define_insn_reservation "k6_alu_imov_store" 1
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imov")
   (eq_attr "memory" "store")))
 "k6_decode_short,k6_store")
```

```
(define_insn_reservation "k6_alu_imov_both" 2
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "imov")
   (eq_attr "memory" "both,unknown")))
 "k6_decode_long,k6_load,k6_alux|k6_aluy")

;; The branch unit.
(define_insn_reservation "k6_branch_call" 1
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "call,callv"))
 "k6_decode_vector,k6_branch")

(define_insn_reservation "k6_branch_branch" 1
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "ibr"))
 "k6_decode_short,k6_branch")

;; The load and units have two pipeline stages.  The load latency is
;; two cycles.
(define_insn_reservation "k6_load_pop" 3
 (and (eq_attr "cpu" "k6")
      (ior (eq_attr "type" "pop")
   (eq_attr "memory" "load,both")))
 "k6_decode_short,k6_load")

(define_insn_reservation "k6_load_leave" 5
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "leave"))
 "k6_decode_long,k6_load,(k6_alux|k6_aluy)*2")

;; ??? From the old pipeline description.  Egad!
;; ??? Apparently we take care of this reservation in adjust_cost.
(define_insn_reservation "k6_load_str" 10
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "str")
   (eq_attr "memory" "load,both")))
 "k6_decode_vector,k6_load*10")

;; The store unit handles lea and push.  It is otherwise unmodelled.
(define_insn_reservation "k6_store_lea" 2
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "lea"))
 "k6_decode_short,k6_store,k6_alux|k6_aluy")

(define_insn_reservation "k6_store_push" 2
 (and (eq_attr "cpu" "k6")
      (ior (eq_attr "type" "push")
   (eq_attr "memory" "store,both")))
 "k6_decode_short,k6_store")
```

```
(define_insn_reservation "k6_store_str" 10
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "str"))
 "k6_store*10")

;; Most FPU instructions have latency 2 and throughput 2.
(define_insn_reservation "k6_fpu" 2
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "fop,fmov,fcmp,fistp")
   (eq_attr "memory" "none")))
 "k6_decode_vector,k6_fpu*2")

(define_insn_reservation "k6_fpu_load" 6
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "fop,fmov,fcmp,fistp")
   (eq_attr "memory" "load,both")))
 "k6_decode_short,k6_load,k6_fpu*2")

(define_insn_reservation "k6_fpu_store" 6
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "fop,fmov,fcmp,fistp")
   (eq_attr "memory" "store")))
 "k6_decode_short,k6_store,k6_fpu*2")

(define_insn_reservation "k6_fpu_fmul" 2
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "fmul")
   (eq_attr "memory" "none")))
 "k6_decode_short,k6_fpu*2")

(define_insn_reservation "k6_fpu_fmul_load" 2
 (and (eq_attr "cpu" "k6")
      (and (eq_attr "type" "fmul")
   (eq_attr "memory" "load,both")))
 "k6_decode_short,k6_load,k6_fpu*2")

;; ??? Guessed latencies from the old pipeline description.
(define_insn_reservation "k6_fpu_expensive" 56
 (and (eq_attr "cpu" "k6")
      (eq_attr "type" "fdiv,fpspc"))
 "k6_decode_short,k6_fpu*56")
```

1.5 AMD Athlon Scheduling

;;(include "athlon.md")

```
;;
;;
;; The Athlon does contain three pipelined FP units, three integer units and
```

```
;; three address generation units.
;;
;; The predecode logic is determining boundaries of instructions in the 64
;; byte cache line. So the cache line straddling problem of K6 might be issue
;; here as well, but it is not noted in the documentation.
;;
;; Three DirectPath instructions decoders and only one VectorPath decoder
;; is available. They can decode three DirectPath instructions or one VectorPath
;; instruction per cycle.
;; Decoded macro instructions are then passed to 72 entry instruction control
;; unit, that passes
;; it to the specialized integer (18 entry) and fp (36 entry) schedulers.
;;
;; The load/store queue unit is not attached to the schedulers but
;; communicates with all the execution units separately instead.

(define_attr "athlon_decode" "direct,vector,double"
  (cond [(eq_attr "type" "call,imul,idiv,other,multi,fcmov,fpspc,str,pop,cld,leave")
   (const_string "vector")
         (and (eq_attr "type" "push")
              (match_operand 1 "memory_operand" ""))
   (const_string "vector")
         (and (eq_attr "type" "fmov")
      (and (eq_attr "memory" "load,store")
   (eq_attr "mode" "XF")))
   (const_string "vector")]
(const_string "direct")))

;;
;;           decode0 decode1 decode2
;;                 \    |   /
;;    instruction control unit (72 entry scheduler)
;;                |                        |
;;      integer scheduler (18)         stack map
;;     /  |    |    |    |   \        stack rename
;;  ieu0 agu0 ieu1 agu1 ieu2 agu2      scheduler
;;    |  agu0  |   agu1      agu2    register file
;;    |      \ |    |       /         |     |     |
;;     \      /\    |     /         fadd  fmul  fstore
;;       \  /    \  |   /           fadd  fmul  fstore
;;       imul  load/store (2x)      fadd  fmul  fstore

(define_automaton "athlon,athlon_load,athlon_mult,athlon_fp")
(define_cpu_unit "athlon-decode0" "athlon")
(define_cpu_unit "athlon-decode1" "athlon")
(define_cpu_unit "athlon-decode2" "athlon")
(define_cpu_unit "athlon-decodev" "athlon")
;; Model the fact that double decoded instruction may take 2 cycles
;; to decode when decoder2 and decoder0 in next cycle
;; is used (this is needed to allow troughput of 1.5 double decoded
```

```
;; instructions per cycle).
;;
;; In order to avoid dependence between reservation of decoder
;; and other units, we model decoder as two stage fully pipelined unit
;; and only double decoded instruction may occupy unit in the first cycle.
;; With this scheme however two double instructions can be issued cycle0.
;;
;; Avoid this by using presence set requiring decoder0 to be allocated
;; too. Vector decoded instructions then can't be issued when
;; modeled as consuming decoder0+decoder1+decoder2.
;; We solve that by specialized vector decoder unit and exclusion set.
(presence_set "athlon-decode2" "athlon-decode0")
(exclusion_set "athlon-decodev" "athlon-decode0,athlon-decode1,athlon-decode2")
(define_reservation "athlon-vector" "nothing,athlon-decodev")
(define_reservation "athlon-direct0" "nothing,athlon-decode0")
(define_reservation "athlon-direct" "nothing,
     (athlon-decode0 | athlon-decode1
     | athlon-decode2)")
;; Double instructions behaves like two direct instructions.
(define_reservation "athlon-double" "((athlon-decode2, athlon-decode0)
     | (nothing,(athlon-decode0 + athlon-decode1))
     | (nothing,(athlon-decode1 + athlon-decode2)))")

;; Agu and ieu unit results in extremely large automatons and
;; in our approximation they are hardly filled in.  Only ieu
;; unit can, as issue rate is 3 and agu unit is always used
;; first in the insn reservations.  Skip the models.

;(define_cpu_unit "athlon-ieu0" "athlon_ieu")
;(define_cpu_unit "athlon-ieu1" "athlon_ieu")
;(define_cpu_unit "athlon-ieu2" "athlon_ieu")
;(define_reservation "athlon-ieu" "(athlon-ieu0 | athlon-ieu1 | athlon-ieu2)")
(define_reservation "athlon-ieu" "nothing")
(define_cpu_unit "athlon-ieu0" "athlon")
;(define_cpu_unit "athlon-agu0" "athlon_agu")
;(define_cpu_unit "athlon-agu1" "athlon_agu")
;(define_cpu_unit "athlon-agu2" "athlon_agu")
;(define_reservation "athlon-agu" "(athlon-agu0 | athlon-agu1 | athlon-agu2)")
(define_reservation "athlon-agu" "nothing")

(define_cpu_unit "athlon-mult" "athlon_mult")

(define_cpu_unit "athlon-load0" "athlon_load")
(define_cpu_unit "athlon-load1" "athlon_load")
(define_reservation "athlon-load" "athlon-agu,
   (athlon-load0 | athlon-load1),nothing")
;; 128bit SSE instructions issue two loads at once
(define_reservation "athlon-load2" "athlon-agu,
   (athlon-load0 + athlon-load1),nothing")
```

```
(define_reservation "athlon-store" "(athlon-load0 | athlon-load1)")
;; 128bit SSE instructions issue two stores at once
(define_reservation "athlon-store2" "(athlon-load0 + athlon-load1)")


;; The FP operations start to execute at stage 12 in the pipeline, while
;; integer operations start to execute at stage 9 for Athlon and 11 for K8
;; Compensate the difference for Athlon because it results in significantly
;; smaller automata.
(define_reservation "athlon-fpsched" "nothing,nothing,nothing")
;; The floating point loads.
(define_reservation "athlon-fpload" "(athlon-fpsched + athlon-load)")
(define_reservation "athlon-fpload2" "(athlon-fpsched + athlon-load2)")
(define_reservation "athlon-fploadk8" "(athlon-fpsched + athlon-load)")
(define_reservation "athlon-fpload2k8" "(athlon-fpsched + athlon-load2)")


;; The three fp units are fully pipelined with latency of 3
(define_cpu_unit "athlon-fadd" "athlon_fp")
(define_cpu_unit "athlon-fmul" "athlon_fp")
(define_cpu_unit "athlon-fstore" "athlon_fp")
(define_reservation "athlon-fany" "(athlon-fstore | athlon-fmul | athlon-fadd)")
(define_reservation "athlon-faddmul" "(athlon-fmul | athlon-fadd)")

;; Vector operations usually consume many of pipes.
(define_reservation "athlon-fvector" "(athlon-fadd + athlon-fmul + athlon-fstore)")


;; Jump instructions are executed in the branch unit completely transparent to us
(define_insn_reservation "athlon_branch" 0
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "ibr"))
 "athlon-direct,athlon-ieu")
(define_insn_reservation "athlon_call" 0
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "call,callv"))
 "athlon-vector,athlon-ieu")

;; Latency of push operation is 3 cycles, but ESP value is available
;; earlier
(define_insn_reservation "athlon_push" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "push"))
 "athlon-direct,athlon-agu,athlon-store")
(define_insn_reservation "athlon_pop" 4
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "pop"))
 "athlon-vector,athlon-load,athlon-ieu")
(define_insn_reservation "athlon_pop_k8" 3
 (and (eq_attr "cpu" "k8")
```

```
      (eq_attr "type" "pop"))
 "athlon-double,(athlon-ieu+athlon-load)")
(define_insn_reservation "athlon_leave" 3
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "leave"))
 "athlon-vector,(athlon-ieu+athlon-load)")
(define_insn_reservation "athlon_leave_k8" 3
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "leave"))
 "athlon-double,(athlon-ieu+athlon-load)")

;; Lea executes in AGU unit with 2 cycles latency.
(define_insn_reservation "athlon_lea" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "lea"))
 "athlon-direct,athlon-agu,nothing")

;; Mul executes in special multiplier unit attached to IEU0
(define_insn_reservation "athlon_imul" 5
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "none,unknown")))
 "athlon-vector,athlon-ieu0,athlon-mult,nothing,nothing,athlon-ieu0")
;; ??? Widening multiply is vector or double.
(define_insn_reservation "athlon_imul_k8_DI" 4
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "imul")
   (and (eq_attr "mode" "DI")
(eq_attr "memory" "none,unknown"))))
 "athlon-direct0,athlon-ieu0,athlon-mult,nothing,athlon-ieu0")
(define_insn_reservation "athlon_imul_k8" 3
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "none,unknown")))
 "athlon-direct0,athlon-ieu0,athlon-mult,athlon-ieu0")
(define_insn_reservation "athlon_imul_mem" 8
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "load,both")))
 "athlon-vector,athlon-load,athlon-ieu,athlon-mult,nothing,nothing,athlon-ieu")
(define_insn_reservation "athlon_imul_mem_k8_DI" 7
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "imul")
   (and (eq_attr "mode" "DI")
(eq_attr "memory" "load,both"))))
 "athlon-vector,athlon-load,athlon-ieu,athlon-mult,nothing,athlon-ieu")
(define_insn_reservation "athlon_imul_mem_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "imul")
   (eq_attr "memory" "load,both")))
```

```
 "athlon-vector,athlon-load,athlon-ieu,athlon-mult,athlon-ieu")

;; Idiv cannot execute in parallel with other instructions.  Dealing with it
;; as with short latency vector instruction is good approximation avoiding
;; scheduler from trying too hard to can hide it's latency by overlap with
;; other instructions.
;; ??? Experiments show that the idiv can overlap with roughly 6 cycles
;; of the other code

(define_insn_reservation "athlon_idiv" 6
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "idiv")
   (eq_attr "memory" "none,unknown")))
 "athlon-vector,(athlon-ieu0*6+(athlon-fpsched,athlon-fvector))")
(define_insn_reservation "athlon_idiv_mem" 9
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "idiv")
   (eq_attr "memory" "load,both")))
 "athlon-vector,((athlon-load,athlon-ieu0*6)+(athlon-fpsched,athlon-fvector))")
;; The parallelism of string instructions is not documented.  Model it same way
;; as idiv to create smaller automata.  This probably does not matter much.
(define_insn_reservation "athlon_str" 6
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "str")
   (eq_attr "memory" "load,both,store")))
 "athlon-vector,athlon-load,athlon-ieu0*6")

(define_insn_reservation "athlon_idirect" 1
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "direct")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "none,unknown"))))
 "athlon-direct,athlon-ieu")
(define_insn_reservation "athlon_ivector" 2
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "vector")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "none,unknown"))))
 "athlon-vector,athlon-ieu,athlon-ieu")
(define_insn_reservation "athlon_idirect_loadmov" 3
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "imov")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-load")
(define_insn_reservation "athlon_idirect_load" 4
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "direct")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-load,athlon-ieu")
```

```
(define_insn_reservation "athlon_ivector_load" 6
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "vector")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "load"))))
 "athlon-vector,athlon-load,athlon-ieu,athlon-ieu")
(define_insn_reservation "athlon_idirect_movstore" 1
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "imov")
   (eq_attr "memory" "store")))
 "athlon-direct,athlon-agu,athlon-store")
(define_insn_reservation "athlon_idirect_both" 4
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "direct")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "both"))))
 "athlon-direct,athlon-load,
  athlon-ieu,athlon-store,
  athlon-store")
(define_insn_reservation "athlon_ivector_both" 6
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "vector")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "both"))))
 "athlon-vector,athlon-load,
  athlon-ieu,
  athlon-ieu,
  athlon-store")
(define_insn_reservation "athlon_idirect_store" 1
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "direct")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "store"))))
 "athlon-direct,(athlon-ieu+athlon-agu),
  athlon-store")
(define_insn_reservation "athlon_ivector_store" 2
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "vector")
   (and (eq_attr "unit" "integer,unknown")
(eq_attr "memory" "store"))))
 "athlon-vector,(athlon-ieu+athlon-agu),athlon-ieu,
  athlon-store")

;; Athlon floatin point unit
(define_insn_reservation "athlon_fldxf" 12
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fmov")
   (and (eq_attr "memory" "load")
(eq_attr "mode" "XF"))))
 "athlon-vector,athlon-fpload2,athlon-fvector*9")
```

```
(define_insn_reservation "athlon_fldxf_k8" 13
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fmov")
   (and (eq_attr "memory" "load")
(eq_attr "mode" "XF"))))
 "athlon-vector,athlon-fpload2k8,athlon-fvector*9")
;; Assume superforwarding to take place so effective latency of fany op is 0.
(define_insn_reservation "athlon_fld" 0
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fmov")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fany")
(define_insn_reservation "athlon_fld_k8" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fmov")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fstore")

(define_insn_reservation "athlon_fstxf" 10
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fmov")
   (and (eq_attr "memory" "store,both")
(eq_attr "mode" "XF"))))
 "athlon-vector,(athlon-fpsched+athlon-agu),(athlon-store2+(athlon-fvector*7))")
(define_insn_reservation "athlon_fstxf_k8" 8
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fmov")
   (and (eq_attr "memory" "store,both")
(eq_attr "mode" "XF"))))
 "athlon-vector,(athlon-fpsched+athlon-agu),(athlon-store2+(athlon-fvector*6))")
(define_insn_reservation "athlon_fst" 4
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fmov")
   (eq_attr "memory" "store,both")))
 "athlon-direct,(athlon-fpsched+athlon-agu),(athlon-fstore+athlon-store)")
(define_insn_reservation "athlon_fst_k8" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fmov")
   (eq_attr "memory" "store,both")))
 "athlon-direct,(athlon-fpsched+athlon-agu),(athlon-fstore+athlon-store)")
(define_insn_reservation "athlon_fist" 4
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fistp"))
 "athlon-direct,(athlon-fpsched+athlon-agu),(athlon-fstore+athlon-store)")
(define_insn_reservation "athlon_fmov" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fmov"))
 "athlon-direct,athlon-fpsched,athlon-faddmul")
(define_insn_reservation "athlon_fadd_load" 4
 (and (eq_attr "cpu" "athlon")
```

```
      (and (eq_attr "type" "fop")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_fadd_load_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fop")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_fadd" 4
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fop"))
 "athlon-direct,athlon-fpsched,athlon-fadd")
(define_insn_reservation "athlon_fmul_load" 4
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fmul")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fmul")
(define_insn_reservation "athlon_fmul_load_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fmul")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fmul")
(define_insn_reservation "athlon_fmul" 4
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fmul"))
 "athlon-direct,athlon-fpsched,athlon-fmul")
(define_insn_reservation "athlon_fsgn" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fsgn"))
 "athlon-direct,athlon-fpsched,athlon-fmul")
(define_insn_reservation "athlon_fdiv_load" 24
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fdiv")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fmul")
(define_insn_reservation "athlon_fdiv_load_k8" 13
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fdiv")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fmul")
(define_insn_reservation "athlon_fdiv" 24
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "fdiv"))
 "athlon-direct,athlon-fpsched,athlon-fmul")
(define_insn_reservation "athlon_fdiv_k8" 11
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "fdiv"))
 "athlon-direct,athlon-fpsched,athlon-fmul")
(define_insn_reservation "athlon_fpspc_load" 103
 (and (eq_attr "cpu" "athlon,k8")
```

```
      (and (eq_attr "type" "fpspc")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload,athlon-fvector")
(define_insn_reservation "athlon_fpspc" 100
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fpspc"))
 "athlon-vector,athlon-fpsched,athlon-fvector")
(define_insn_reservation "athlon_fcmov_load" 7
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fcmov")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload,athlon-fvector")
(define_insn_reservation "athlon_fcmov" 7
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "fcmov"))
 "athlon-vector,athlon-fpsched,athlon-fvector")
(define_insn_reservation "athlon_fcmov_load_k8" 17
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fcmov")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fploadk8,athlon-fvector")
(define_insn_reservation "athlon_fcmov_k8" 15
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "fcmov"))
 "athlon-vector,athlon-fpsched,athlon-fvector")
;; fcomi is vector decoded by uses only one pipe.
(define_insn_reservation "athlon_fcomi_load" 3
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fcmp")
   (and (eq_attr "athlon_decode" "vector")
        (eq_attr "memory" "load"))))
 "athlon-vector,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_fcomi_load_k8" 5
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "fcmp")
   (and (eq_attr "athlon_decode" "vector")
        (eq_attr "memory" "load"))))
 "athlon-vector,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_fcomi" 3
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "athlon_decode" "vector")
   (eq_attr "type" "fcmp")))
 "athlon-vector,athlon-fpsched,athlon-fadd")
(define_insn_reservation "athlon_fcom_load" 2
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "fcmp")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_fcom_load_k8" 4
 (and (eq_attr "cpu" "k8")
```

```
      (and (eq_attr "type" "fcmp")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_fcom" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "fcmp"))
 "athlon-direct,athlon-fpsched,athlon-fadd")
;; Never seen by the scheduler because we still don't do post reg-stack
;; scheduling.
;(define_insn_reservation "athlon_fxch" 2
;  (and (eq_attr "cpu" "athlon,k8")
;       (eq_attr "type" "fxch"))
;  "athlon-direct,athlon-fpsched,athlon-fany")

;; Athlon handle MMX operations in the FPU unit with shorter latencies

(define_insn_reservation "athlon_movlpd_load" 0
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemov")
   (match_operand:DF 1 "memory_operand" "")))
 "athlon-direct,athlon-fpload,athlon-fany")
(define_insn_reservation "athlon_movlpd_load_k8" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemov")
   (match_operand:DF 1 "memory_operand" "")))
 "athlon-direct,athlon-fploadk8,athlon-fstore")
(define_insn_reservation "athlon_movaps_load_k8" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemov")
   (and (eq_attr "mode" "V4SF,V2DF,TI")
(eq_attr "memory" "load"))))
 "athlon-double,athlon-fpload2k8,athlon-fstore,athlon-fstore")
(define_insn_reservation "athlon_movaps_load" 0
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemov")
   (and (eq_attr "mode" "V4SF,V2DF,TI")
(eq_attr "memory" "load"))))
 "athlon-vector,athlon-fpload2,(athlon-fany+athlon-fany)")
(define_insn_reservation "athlon_movss_load" 1
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemov")
   (and (eq_attr "mode" "SF,DI")
(eq_attr "memory" "load"))))
 "athlon-vector,athlon-fpload,(athlon-fany*2)")
(define_insn_reservation "athlon_movss_load_k8" 1
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemov")
   (and (eq_attr "mode" "SF,DI")
(eq_attr "memory" "load"))))
 "athlon-double,athlon-fploadk8,(athlon-fstore+athlon-fany)")
```

```
(define_insn_reservation "athlon_mmxsseld" 0
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "mmxmov,ssemov")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fany")
(define_insn_reservation "athlon_mmxsseld_k8" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "mmxmov,ssemov")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fploadk8,athlon-fstore")
(define_insn_reservation "athlon_mmxssest" 3
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "mmxmov,ssemov")
   (and (eq_attr "mode" "V4SF,V2DF,TI")
(eq_attr "memory" "store,both"))))
 "athlon-vector,(athlon-fpsched+athlon-agu),((athlon-fstore+athlon-store2)*2)")
(define_insn_reservation "athlon_mmxssest_k8" 3
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "mmxmov,ssemov")
   (and (eq_attr "mode" "V4SF,V2DF,TI")
(eq_attr "memory" "store,both"))))
 "athlon-double,(athlon-fpsched+athlon-agu),((athlon-fstore+athlon-store2)*2)")
(define_insn_reservation "athlon_mmxssest_short" 2
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "mmxmov,ssemov")
   (eq_attr "memory" "store,both")))
 "athlon-direct,(athlon-fpsched+athlon-agu),(athlon-fstore+athlon-store)")
(define_insn_reservation "athlon_movaps" 2
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemov")
   (eq_attr "mode" "V4SF,V2DF,TI")))
 "athlon-double,athlon-fpsched,(athlon-faddmul+athlon-faddmul)")
(define_insn_reservation "athlon_movaps_k8" 2
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemov")
   (eq_attr "mode" "V4SF,V2DF,TI")))
 "athlon-vector,athlon-fpsched,(athlon-faddmul+athlon-faddmul)")
(define_insn_reservation "athlon_mmxssemov" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "mmxmov,ssemov"))
 "athlon-direct,athlon-fpsched,athlon-faddmul")
(define_insn_reservation "athlon_mmxmul_load" 4
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "mmxmul")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-fmul")
(define_insn_reservation "athlon_mmxmul" 3
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "mmxmul"))
 "athlon-direct,athlon-fpsched,athlon-fmul")
```

```
(define_insn_reservation "athlon_mmx_load" 3
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "unit" "mmx")
   (eq_attr "memory" "load")))
 "athlon-direct,athlon-fpload,athlon-faddmul")
(define_insn_reservation "athlon_mmx" 2
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "unit" "mmx"))
 "athlon-direct,athlon-fpsched,athlon-faddmul")
;; SSE operations are handled by the i387 unit as well.  The latency
;; is same as for i387 operations for scalar operations

(define_insn_reservation "athlon_sselog_load" 3
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "sselog,sselog1")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload2,(athlon-fmul*2)")
(define_insn_reservation "athlon_sselog_load_k8" 5
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "sselog,sselog1")
   (eq_attr "memory" "load")))
 "athlon-double,athlon-fpload2k8,(athlon-fmul*2)")
(define_insn_reservation "athlon_sselog" 3
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "sselog,sselog1"))
 "athlon-vector,athlon-fpsched,athlon-fmul*2")
(define_insn_reservation "athlon_sselog_k8" 3
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "sselog,sselog1"))
 "athlon-double,athlon-fpsched,athlon-fmul")
;; ??? pcmp executes in addmul, probably not worthwhile to bother about that.
(define_insn_reservation "athlon_ssecmp_load" 2
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssecmp")
   (and (eq_attr "mode" "SF,DF,DI")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_ssecmp_load_k8" 4
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssecmp")
   (and (eq_attr "mode" "SF,DF,DI,TI")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_ssecmp" 2
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssecmp")
   (eq_attr "mode" "SF,DF,DI,TI")))
 "athlon-direct,athlon-fpsched,athlon-fadd")
(define_insn_reservation "athlon_ssecmpvector_load" 3
 (and (eq_attr "cpu" "athlon")
```

```
      (and (eq_attr "type" "ssecmp")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload2,(athlon-fadd*2)")
(define_insn_reservation "athlon_ssecmpvector_load_k8" 5
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssecmp")
   (eq_attr "memory" "load")))
 "athlon-double,athlon-fpload2k8,(athlon-fadd*2)")
(define_insn_reservation "athlon_ssecmpvector" 3
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "ssecmp"))
 "athlon-vector,athlon-fpsched,(athlon-fadd*2)")
(define_insn_reservation "athlon_ssecmpvector_k8" 3
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "ssecmp"))
 "athlon-double,athlon-fpsched,(athlon-fadd*2)")
(define_insn_reservation "athlon_ssecomi_load" 4
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssecomi")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_ssecomi_load_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssecomi")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_ssecomi" 4
 (and (eq_attr "cpu" "athlon,k8")
      (eq_attr "type" "ssecmp"))
 "athlon-vector,athlon-fpsched,athlon-fadd")
(define_insn_reservation "athlon_sseadd_load" 4
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "sseadd")
   (and (eq_attr "mode" "SF,DF,DI")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fpload,athlon-fadd")
(define_insn_reservation "athlon_sseadd_load_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "sseadd")
   (and (eq_attr "mode" "SF,DF,DI")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fploadk8,athlon-fadd")
(define_insn_reservation "athlon_sseadd" 4
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "sseadd")
   (eq_attr "mode" "SF,DF,DI")))
 "athlon-direct,athlon-fpsched,athlon-fadd")
(define_insn_reservation "athlon_sseaddvector_load" 5
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "sseadd")
```

```
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload2,(athlon-fadd*2)")
(define_insn_reservation "athlon_sseaddvector_load_k8" 7
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "sseadd")
   (eq_attr "memory" "load")))
 "athlon-double,athlon-fpload2k8,(athlon-fadd*2)")
(define_insn_reservation "athlon_sseaddvector" 5
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "sseadd"))
 "athlon-vector,athlon-fpsched,(athlon-fadd*2)")
(define_insn_reservation "athlon_sseaddvector_k8" 5
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "sseadd"))
 "athlon-double,athlon-fpsched,(athlon-fadd*2)")

;; Conversions behaves very irregularly and the scheduling is critical here.
;; Take each instruction separately.  Assume that the mode is always set to the
;; destination one and athlon_decode is set to the K8 versions.

;; cvtss2sd
(define_insn_reservation "athlon_ssecvt_cvtss2sd_load_k8" 4
 (and (eq_attr "cpu" "k8,athlon")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "direct")
(and (eq_attr "mode" "DF")
     (eq_attr "memory" "load")))))
 "athlon-direct,athlon-fploadk8,athlon-fstore")
(define_insn_reservation "athlon_ssecvt_cvtss2sd" 2
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "direct")
(eq_attr "mode" "DF"))))
 "athlon-direct,athlon-fpsched,athlon-fstore")
;; cvtps2pd.  Model same way the other double decoded FP conversions.
(define_insn_reservation "athlon_ssecvt_cvtps2pd_load_k8" 5
 (and (eq_attr "cpu" "k8,athlon")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "V2DF,V4SF,TI")
     (eq_attr "memory" "load")))))
 "athlon-double,athlon-fpload2k8,(athlon-fstore*2)")
(define_insn_reservation "athlon_ssecvt_cvtps2pd_k8" 3
 (and (eq_attr "cpu" "k8,athlon")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "double")
(eq_attr "mode" "V2DF,V4SF,TI"))))
 "athlon-double,athlon-fpsched,athlon-fstore,athlon-fstore")
;; cvtsi2sd mem,reg is directpath path  (cvtsi2sd reg,reg is doublepath)
;; cvtsi2sd has troughput 1 and is executed in store unit with latency of 6
```

```
(define_insn_reservation "athlon_sseicvt_cvtsi2sd_load" 6
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "direct")
(and (eq_attr "mode" "SF,DF")
     (eq_attr "memory" "load")))))
 "athlon-direct,athlon-fploadk8,athlon-fstore")
;; cvtsi2ss mem, reg is doublepath
(define_insn_reservation "athlon_sseicvt_cvtsi2ss_load" 9
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SF,DF")
     (eq_attr "memory" "load")))))
 "athlon-vector,athlon-fpload,(athlon-fstore*2)")
(define_insn_reservation "athlon_sseicvt_cvtsi2ss_load_k8" 9
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SF,DF")
     (eq_attr "memory" "load")))))
 "athlon-double,athlon-fploadk8,(athlon-fstore*2)")
;; cvtsi2sd reg,reg is double decoded (vector on Athlon)
(define_insn_reservation "athlon_sseicvt_cvtsi2sd_k8" 11
 (and (eq_attr "cpu" "k8,athlon")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SF,DF")
     (eq_attr "memory" "none")))))
 "athlon-double,athlon-fploadk8,athlon-fstore")
;; cvtsi2ss reg, reg is doublepath
(define_insn_reservation "athlon_sseicvt_cvtsi2ss" 14
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "vector")
(and (eq_attr "mode" "SF,DF")
     (eq_attr "memory" "none")))))
 "athlon-vector,athlon-fploadk8,(athlon-fvector*2)")
;; cvtsd2ss mem,reg is doublepath, troughput unknown, latency 9
(define_insn_reservation "athlon_ssecvt_cvtsd2ss_load_k8" 9
 (and (eq_attr "cpu" "k8,athlon")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SF")
     (eq_attr "memory" "load")))))
 "athlon-double,athlon-fploadk8,(athlon-fstore*3)")
;; cvtsd2ss reg,reg is vectorpath, troughput unknown, latency 12
(define_insn_reservation "athlon_ssecvt_cvtsd2ss" 12
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssecvt")
```

```
   (and (eq_attr "athlon_decode" "vector")
(and (eq_attr "mode" "SF")
     (eq_attr "memory" "none")))))
 "athlon-vector,athlon-fpsched,(athlon-fvector*3)")
(define_insn_reservation "athlon_ssecvt_cvtpd2ps_load_k8" 8
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "vector")
(and (eq_attr "mode" "V4SF,V2DF,TI")
     (eq_attr "memory" "load")))))
 "athlon-double,athlon-fpload2k8,(athlon-fstore*3)")
;; cvtpd2ps mem,reg is vectorpath, troughput unknown, latency 10
;; ??? Why it is fater than cvtsd2ss?
(define_insn_reservation "athlon_ssecvt_cvtpd2ps" 8
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssecvt")
   (and (eq_attr "athlon_decode" "vector")
(and (eq_attr "mode" "V4SF,V2DF,TI")
     (eq_attr "memory" "none")))))
 "athlon-vector,athlon-fpsched,athlon-fvector*2")
;; cvtsd2si mem,reg is doublepath, troughput 1, latency 9
(define_insn_reservation "athlon_secvt_cvtsX2si_load" 9
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "vector")
(and (eq_attr "mode" "SI,DI")
     (eq_attr "memory" "load")))))
 "athlon-vector,athlon-fploadk8,athlon-fvector")
;; cvtsd2si reg,reg is doublepath, troughput 1, latency 9
(define_insn_reservation "athlon_ssecvt_cvtsX2si" 9
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SI,DI")
     (eq_attr "memory" "none")))))
 "athlon-vector,athlon-fpsched,athlon-fvector")
(define_insn_reservation "athlon_ssecvt_cvtsX2si_k8" 9
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "sseicvt")
   (and (eq_attr "athlon_decode" "double")
(and (eq_attr "mode" "SI,DI")
     (eq_attr "memory" "none")))))
 "athlon-double,athlon-fpsched,athlon-fstore")


(define_insn_reservation "athlon_ssemul_load" 4
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemul")
   (and (eq_attr "mode" "SF,DF")
(eq_attr "memory" "load"))))
```

```
 "athlon-direct,athlon-fpload,athlon-fmul")
(define_insn_reservation "athlon_ssemul_load_k8" 6
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemul")
   (and (eq_attr "mode" "SF,DF")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fploadk8,athlon-fmul")
(define_insn_reservation "athlon_ssemul" 4
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssemul")
   (eq_attr "mode" "SF,DF")))
 "athlon-direct,athlon-fpsched,athlon-fmul")
(define_insn_reservation "athlon_ssemulvector_load" 5
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssemul")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload2,(athlon-fmul*2)")
(define_insn_reservation "athlon_ssemulvector_load_k8" 7
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssemul")
   (eq_attr "memory" "load")))
 "athlon-double,athlon-fpload2k8,(athlon-fmul*2)")
(define_insn_reservation "athlon_ssemulvector" 5
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "ssemul"))
 "athlon-vector,athlon-fpsched,(athlon-fmul*2)")
(define_insn_reservation "athlon_ssemulvector_k8" 5
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "ssemul"))
 "athlon-double,athlon-fpsched,(athlon-fmul*2)")
;; divsd timings.  divss is faster
(define_insn_reservation "athlon_ssediv_load" 20
 (and (eq_attr "cpu" "athlon")
      (and (eq_attr "type" "ssediv")
   (and (eq_attr "mode" "SF,DF")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fpload,athlon-fmul*17")
(define_insn_reservation "athlon_ssediv_load_k8" 22
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssediv")
   (and (eq_attr "mode" "SF,DF")
(eq_attr "memory" "load"))))
 "athlon-direct,athlon-fploadk8,athlon-fmul*17")
(define_insn_reservation "athlon_ssediv" 20
 (and (eq_attr "cpu" "athlon,k8")
      (and (eq_attr "type" "ssediv")
   (eq_attr "mode" "SF,DF")))
 "athlon-direct,athlon-fpsched,athlon-fmul*17")
(define_insn_reservation "athlon_ssedivvector_load" 39
 (and (eq_attr "cpu" "athlon")
```

```
      (and (eq_attr "type" "ssediv")
   (eq_attr "memory" "load")))
 "athlon-vector,athlon-fpload2,athlon-fmul*34")
(define_insn_reservation "athlon_ssedivvector_load_k8" 35
 (and (eq_attr "cpu" "k8")
      (and (eq_attr "type" "ssediv")
   (eq_attr "memory" "load")))
 "athlon-double,athlon-fpload2k8,athlon-fmul*34")
(define_insn_reservation "athlon_ssedivvector" 39
 (and (eq_attr "cpu" "athlon")
      (eq_attr "type" "ssediv"))
 "athlon-vector,athlon-fmul*34")
(define_insn_reservation "athlon_ssedivvector_k8" 39
 (and (eq_attr "cpu" "k8")
      (eq_attr "type" "ssediv"))
 "athlon-double,athlon-fmul*34")
```

1.6 Operand and operator predicates

```
\subsubsection{Predicate definitions for IA-32 and x86-64.}
;;(include "predicates.md")
\begin{verbatim}

;; Return nonzero if OP is either a i387 or SSE fp register.
(define_predicate "any_fp_register_operand"
  (and (match_code "reg")
       (match_test "ANY_FP_REGNO_P (REGNO (op))")))

;; Return nonzero if OP is an i387 fp register.
(define_predicate "fp_register_operand"
  (and (match_code "reg")
       (match_test "FP_REGNO_P (REGNO (op))")))

;; Return nonzero if OP is a non-fp register_operand.
(define_predicate "register_and_not_any_fp_reg_operand"
  (and (match_code "reg")
       (not (match_test "ANY_FP_REGNO_P (REGNO (op))"))))

;; Return nonzero if OP is a register operand other than an i387 fp register.
(define_predicate "register_and_not_fp_reg_operand"
  (and (match_code "reg")
       (not (match_test "FP_REGNO_P (REGNO (op))"))))

;; True if the operand is an MMX register.
(define_predicate "mmx_reg_operand"
  (and (match_code "reg")
       (match_test "MMX_REGNO_P (REGNO (op))")))

;; True if the operand is a Q_REGS class register.
```

```
(define_predicate "q_regs_operand"
  (match_operand 0 "register_operand")
{
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);
  return ANY_QI_REG_P (op);
})

;; Return true if op is a NON_Q_REGS class register.
(define_predicate "non_q_regs_operand"
  (match_operand 0 "register_operand")
{
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);
  return NON_QI_REG_P (op);
})

;; Match an SI or HImode register for a zero_extract.
(define_special_predicate "ext_register_operand"
  (match_operand 0 "register_operand")
{
  if ((!TARGET_64BIT || GET_MODE (op) != DImode)
      && GET_MODE (op) != SImode && GET_MODE (op) != HImode)
    return 0;
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);

  /* Be careful to accept only registers having upper parts.  */
  return REGNO (op) > LAST_VIRTUAL_REGISTER || REGNO (op) < 4;
})

;; Return true if op is the flags register.
(define_predicate "flags_reg_operand"
  (and (match_code "reg")
       (match_test "REGNO (op) == FLAGS_REG")))

;; Return 1 if VALUE can be stored in a sign extended immediate field.
(define_predicate "x86_64_immediate_operand"
  (match_code "const_int,symbol_ref,label_ref,const")
{
  if (!TARGET_64BIT)
    return immediate_operand (op, mode);

  switch (GET_CODE (op))
    {
    case CONST_INT:
      /* CONST_DOUBLES never match, since HOST_BITS_PER_WIDE_INT is known
         to be at least 32 and this all acceptable constants are
 represented as CONST_INT.  */
      if (HOST_BITS_PER_WIDE_INT == 32)
```

```
return 1;
      else
{
  HOST_WIDE_INT val = trunc_int_for_mode (INTVAL (op), DImode);
  return trunc_int_for_mode (val, SImode) == val;
}
      break;

    case SYMBOL_REF:
      /* For certain code models, the symbolic references are known to fit.
 in CM_SMALL_PIC model we know it fits if it is local to the shared
 library.  Don't count TLS SYMBOL_REFs here, since they should fit
 only if inside of UNSPEC handled below.  */
      /* TLS symbols are not constant.  */
      if (tls_symbolic_operand (op, Pmode))
return false;
      return (ix86_cmodel == CM_SMALL || ix86_cmodel == CM_KERNEL
      || (ix86_cmodel == CM_MEDIUM && !SYMBOL_REF_FAR_ADDR_P (op)));

    case LABEL_REF:
      /* For certain code models, the code is near as well.  */
      return (ix86_cmodel == CM_SMALL || ix86_cmodel == CM_MEDIUM
      || ix86_cmodel == CM_KERNEL);

    case CONST:
      /* We also may accept the offsetted memory references in certain
 special cases.  */
      if (GET_CODE (XEXP (op, 0)) == UNSPEC)
switch (XINT (XEXP (op, 0), 1))
  {
  case UNSPEC_GOTPCREL:
  case UNSPEC_DTPOFF:
  case UNSPEC_GOTNTPOFF:
  case UNSPEC_NTPOFF:
    return 1;
  default:
    break;
  }

      if (GET_CODE (XEXP (op, 0)) == PLUS)
{
  rtx op1 = XEXP (XEXP (op, 0), 0);
  rtx op2 = XEXP (XEXP (op, 0), 1);
  HOST_WIDE_INT offset;

  if (ix86_cmodel == CM_LARGE)
    return 0;
  if (GET_CODE (op2) != CONST_INT)
    return 0;
  offset = trunc_int_for_mode (INTVAL (op2), DImode);
```

```
  switch (GET_CODE (op1))
    {
    case SYMBOL_REF:
      /* For CM_SMALL assume that latest object is 16MB before
 end of 31bits boundary.  We may also accept pretty
 large negative constants knowing that all objects are
 in the positive half of address space.  */
      if ((ix86_cmodel == CM_SMALL
   || (ix86_cmodel == CM_MEDIUM
       && !SYMBOL_REF_FAR_ADDR_P (op1)))
  && offset < 16*1024*1024
  && trunc_int_for_mode (offset, SImode) == offset)
return 1;
      /* For CM_KERNEL we know that all object resist in the
 negative half of 32bits address space.  We may not
 accept negative offsets, since they may be just off
 and we may accept pretty large positive ones.  */
      if (ix86_cmodel == CM_KERNEL
  && offset > 0
  && trunc_int_for_mode (offset, SImode) == offset)
return 1;
      break;

    case LABEL_REF:
      /* These conditions are similar to SYMBOL_REF ones, just the
 constraints for code models differ.  */
      if ((ix86_cmodel == CM_SMALL || ix86_cmodel == CM_MEDIUM)
  && offset < 16*1024*1024
  && trunc_int_for_mode (offset, SImode) == offset)
return 1;
      if (ix86_cmodel == CM_KERNEL
  && offset > 0
  && trunc_int_for_mode (offset, SImode) == offset)
return 1;
      break;

    case UNSPEC:
      switch (XINT (op1, 1))
{
case UNSPEC_DTPOFF:
case UNSPEC_NTPOFF:
  if (offset > 0
      && trunc_int_for_mode (offset, SImode) == offset)
    return 1;
}
      break;

    default:
      break;
    }
```

```
}
      break;

      default:
gcc_unreachable ();
    }

  return 0;
})

;; Return 1 if VALUE can be stored in the zero extended immediate field.
(define_predicate "x86_64_zext_immediate_operand"
  (match_code "const_double,const_int,symbol_ref,label_ref,const")
{
  switch (GET_CODE (op))
    {
    case CONST_DOUBLE:
      if (HOST_BITS_PER_WIDE_INT == 32)
return (GET_MODE (op) == VOIDmode && !CONST_DOUBLE_HIGH (op));
      else
return 0;

    case CONST_INT:
      if (HOST_BITS_PER_WIDE_INT == 32)
return INTVAL (op) >= 0;
      else
return !(INTVAL (op) & ~(HOST_WIDE_INT) 0xffffffff);

    case SYMBOL_REF:
      /* For certain code models, the symbolic references are known to fit.  */
      /* TLS symbols are not constant.  */
      if (tls_symbolic_operand (op, Pmode))
return false;
      return (ix86_cmodel == CM_SMALL
      || (ix86_cmodel == CM_MEDIUM
  && !SYMBOL_REF_FAR_ADDR_P (op)));

    case LABEL_REF:
      /* For certain code models, the code is near as well.  */
      return ix86_cmodel == CM_SMALL || ix86_cmodel == CM_MEDIUM;

    case CONST:
      /* We also may accept the offsetted memory references in certain
 special cases.  */
      if (GET_CODE (XEXP (op, 0)) == PLUS)
{
  rtx op1 = XEXP (XEXP (op, 0), 0);
  rtx op2 = XEXP (XEXP (op, 0), 1);

  if (ix86_cmodel == CM_LARGE)
```

```
    return 0;
  switch (GET_CODE (op1))
    {
    case SYMBOL_REF:
      /* For small code model we may accept pretty large positive
 offsets, since one bit is available for free.  Negative
 offsets are limited by the size of NULL pointer area
 specified by the ABI.  */
      if ((ix86_cmodel == CM_SMALL
   || (ix86_cmodel == CM_MEDIUM
       && !SYMBOL_REF_FAR_ADDR_P (op1)))
  && GET_CODE (op2) == CONST_INT
  && trunc_int_for_mode (INTVAL (op2), DImode) > -0x10000
  && trunc_int_for_mode (INTVAL (op2), SImode) == INTVAL (op2))
return 1;
      /* ??? For the kernel, we may accept adjustment of
 -0x10000000, since we know that it will just convert
 negative address space to positive, but perhaps this
 is not worthwhile.  */
      break;

    case LABEL_REF:
      /* These conditions are similar to SYMBOL_REF ones, just the
 constraints for code models differ.  */
      if ((ix86_cmodel == CM_SMALL || ix86_cmodel == CM_MEDIUM)
  && GET_CODE (op2) == CONST_INT
  && trunc_int_for_mode (INTVAL (op2), DImode) > -0x10000
  && trunc_int_for_mode (INTVAL (op2), SImode) == INTVAL (op2))
return 1;
      break;

    default:
      return 0;
    }
}
      break;

    default:
      gcc_unreachable ();
    }
  return 0;
})

;; Return nonzero if OP is general operand representable on x86_64.
(define_predicate "x86_64_general_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (ior (match_operand 0 "nonimmediate_operand")
 (match_operand 0 "x86_64_immediate_operand"))
    (match_operand 0 "general_operand")))
```

```
;; Return nonzero if OP is general operand representable on x86_64
;; as either sign extended or zero extended constant.
(define_predicate "x86_64_szext_general_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (ior (match_operand 0 "nonimmediate_operand")
 (ior (match_operand 0 "x86_64_immediate_operand")
      (match_operand 0 "x86_64_zext_immediate_operand")))
    (match_operand 0 "general_operand")))

;; Return nonzero if OP is nonmemory operand representable on x86_64.
(define_predicate "x86_64_nonmemory_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (ior (match_operand 0 "register_operand")
 (match_operand 0 "x86_64_immediate_operand"))
    (match_operand 0 "nonmemory_operand")))

;; Return nonzero if OP is nonmemory operand representable on x86_64.
(define_predicate "x86_64_szext_nonmemory_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (ior (match_operand 0 "register_operand")
 (ior (match_operand 0 "x86_64_immediate_operand")
      (match_operand 0 "x86_64_zext_immediate_operand")))
    (match_operand 0 "nonmemory_operand")))

;; Return true when operand is PIC expression that can be computed by lea
;; operation.
(define_predicate "pic_32bit_operand"
  (match_code "const,symbol_ref,label_ref")
{
  if (!flag_pic)
    return 0;
  /* Rule out relocations that translate into 64bit constants.  */
  if (TARGET_64BIT && GET_CODE (op) == CONST)
    {
      op = XEXP (op, 0);
      if (GET_CODE (op) == PLUS && GET_CODE (XEXP (op, 1)) == CONST_INT)
op = XEXP (op, 0);
      if (GET_CODE (op) == UNSPEC
  && (XINT (op, 1) == UNSPEC_GOTOFF
      || XINT (op, 1) == UNSPEC_GOT))
return 0;
    }
  return symbolic_operand (op, mode);
})


;; Return nonzero if OP is nonmemory operand acceptable by movabs patterns.
(define_predicate "x86_64_movabs_operand"
  (if_then_else (match_test "!TARGET_64BIT || !flag_pic")
    (match_operand 0 "nonmemory_operand")
```

```
    (ior (match_operand 0 "register_operand")
 (and (match_operand 0 "const_double_operand")
      (match_test "GET_MODE_SIZE (mode) <= 8"))))

;; Returns nonzero if OP is either a symbol reference or a sum of a symbol
;; reference and a constant.
(define_predicate "symbolic_operand"
  (match_code "symbol_ref,label_ref,const")
{
  switch (GET_CODE (op))
    {
    case SYMBOL_REF:
    case LABEL_REF:
      return 1;

    case CONST:
      op = XEXP (op, 0);
      if (GET_CODE (op) == SYMBOL_REF
  || GET_CODE (op) == LABEL_REF
  || (GET_CODE (op) == UNSPEC
      && (XINT (op, 1) == UNSPEC_GOT
  || XINT (op, 1) == UNSPEC_GOTOFF
  || XINT (op, 1) == UNSPEC_GOTPCREL)))
return 1;
      if (GET_CODE (op) != PLUS
  || GET_CODE (XEXP (op, 1)) != CONST_INT)
return 0;

      op = XEXP (op, 0);
      if (GET_CODE (op) == SYMBOL_REF
  || GET_CODE (op) == LABEL_REF)
return 1;
      /* Only @GOTOFF gets offsets.  */
      if (GET_CODE (op) != UNSPEC
  || XINT (op, 1) != UNSPEC_GOTOFF)
return 0;

      op = XVECEXP (op, 0, 0);
      if (GET_CODE (op) == SYMBOL_REF
  || GET_CODE (op) == LABEL_REF)
return 1;
      return 0;

    default:
      gcc_unreachable ();
    }
})

;; Return true if the operand contains a @GOT or @GOTOFF reference.
(define_predicate "pic_symbolic_operand"
```

```
  (match_code "const")
{
  op = XEXP (op, 0);
  if (TARGET_64BIT)
    {
      if (GET_CODE (op) == UNSPEC
  && XINT (op, 1) == UNSPEC_GOTPCREL)
return 1;
      if (GET_CODE (op) == PLUS
  && GET_CODE (XEXP (op, 0)) == UNSPEC
  && XINT (XEXP (op, 0), 1) == UNSPEC_GOTPCREL)
return 1;
    }
  else
    {
      if (GET_CODE (op) == UNSPEC)
return 1;
      if (GET_CODE (op) != PLUS
  || GET_CODE (XEXP (op, 1)) != CONST_INT)
return 0;
      op = XEXP (op, 0);
      if (GET_CODE (op) == UNSPEC)
return 1;
    }
  return 0;
})

;; Return true if OP is a symbolic operand that resolves locally.
(define_predicate "local_symbolic_operand"
  (match_code "const,label_ref,symbol_ref")
{
  if (GET_CODE (op) == CONST
      && GET_CODE (XEXP (op, 0)) == PLUS
      && GET_CODE (XEXP (XEXP (op, 0), 1)) == CONST_INT)
    op = XEXP (XEXP (op, 0), 0);

  if (GET_CODE (op) == LABEL_REF)
    return 1;

  if (GET_CODE (op) != SYMBOL_REF)
    return 0;

  if (SYMBOL_REF_LOCAL_P (op))
    return 1;

  /* There is, however, a not insubstantial body of code in the rest of
     the compiler that assumes it can just stick the results of
     ASM_GENERATE_INTERNAL_LABEL in a symbol_ref and have done.  */
  /* ??? This is a hack.  Should update the body of the compiler to
     always create a DECL an invoke targetm.encode_section_info.  */
```

```
  if (strncmp (XSTR (op, 0), internal_label_prefix,
       internal_label_prefix_len) == 0)
    return 1;

  return 0;
})

;; Test for various thread-local symbols.
(define_predicate "tls_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_TLS_MODEL (op) != 0")))

(define_predicate "global_dynamic_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_TLS_MODEL (op) == TLS_MODEL_GLOBAL_DYNAMIC")))

(define_predicate "local_dynamic_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_TLS_MODEL (op) == TLS_MODEL_LOCAL_DYNAMIC")))

(define_predicate "initial_exec_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_TLS_MODEL (op) == TLS_MODEL_INITIAL_EXEC")))

(define_predicate "local_exec_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_TLS_MODEL (op) == TLS_MODEL_LOCAL_EXEC")))

;; Test for a pc-relative call operand
(define_predicate "constant_call_address_operand"
  (ior (match_code "symbol_ref")
       (match_operand 0 "local_symbolic_operand")))

;; True for any non-virtual or eliminable register.  Used in places where
;; instantiation of such a register may cause the pattern to not be recognized.
(define_predicate "register_no_elim_operand"
  (match_operand 0 "register_operand")
{
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);
  return !(op == arg_pointer_rtx
   || op == frame_pointer_rtx
   || (REGNO (op) >= FIRST_PSEUDO_REGISTER
       && REGNO (op) <= LAST_VIRTUAL_REGISTER));
})

;; Similarly, but include the stack pointer.  This is used to prevent esp
;; from being used as an index reg.
(define_predicate "index_register_operand"
  (match_operand 0 "register_operand")
```

```
{
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);
  if (reload_in_progress || reload_completed)
    return REG_OK_FOR_INDEX_STRICT_P (op);
  else
    return REG_OK_FOR_INDEX_NONSTRICT_P (op);
})

;; Return false if this is any eliminable register.  Otherwise general_operand.
(define_predicate "general_no_elim_operand"
  (if_then_else (match_code "reg,subreg")
    (match_operand 0 "register_no_elim_operand")
    (match_operand 0 "general_operand")))

;; Return false if this is any eliminable register.  Otherwise
;; register_operand or a constant.
(define_predicate "nonmemory_no_elim_operand"
  (ior (match_operand 0 "register_no_elim_operand")
       (match_operand 0 "immediate_operand")))

;; Test for a valid operand for a call instruction.
(define_predicate "call_insn_operand"
  (ior (match_operand 0 "constant_call_address_operand")
       (ior (match_operand 0 "register_no_elim_operand")
    (match_operand 0 "memory_operand"))))

;; Similarly, but for tail calls, in which we cannot allow memory references.
(define_predicate "sibcall_insn_operand"
  (ior (match_operand 0 "constant_call_address_operand")
       (match_operand 0 "register_no_elim_operand")))

;; Match exactly zero.
(define_predicate "const0_operand"
  (match_code "const_int,const_double,const_vector")
{
  if (mode == VOIDmode)
    mode = GET_MODE (op);
  return op == CONST0_RTX (mode);
})

;; Match exactly one.
(define_predicate "const1_operand"
  (and (match_code "const_int")
       (match_test "op == const1_rtx")))

;; Match 2, 4, or 8.  Used for leal multiplicands.
(define_predicate "const248_operand"
  (match_code "const_int")
{
```

```
  HOST_WIDE_INT i = INTVAL (op);
  return i == 2 || i == 4 || i == 8;
})

;; Match 0 or 1.
(define_predicate "const_0_to_1_operand"
  (and (match_code "const_int")
       (match_test "op == const0_rtx || op == const1_rtx")))

;; Match 0 to 3.
(define_predicate "const_0_to_3_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 0 && INTVAL (op) <= 3")))

;; Match 0 to 7.
(define_predicate "const_0_to_7_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 0 && INTVAL (op) <= 7")))

;; Match 0 to 15.
(define_predicate "const_0_to_15_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 0 && INTVAL (op) <= 15")))

;; Match 0 to 63.
(define_predicate "const_0_to_63_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 0 && INTVAL (op) <= 63")))

;; Match 0 to 255.
(define_predicate "const_0_to_255_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 0 && INTVAL (op) <= 255")))

;; Match (0 to 255) * 8
(define_predicate "const_0_to_255_mul_8_operand"
  (match_code "const_int")
{
  unsigned HOST_WIDE_INT val = INTVAL (op);
  return val <= 255*8 && val % 8 == 0;
})

;; Return nonzero if OP is CONST_INT >= 1 and <= 31 (a valid operand
;; for shift & compare patterns, as shifting by 0 does not change flags).
(define_predicate "const_1_to_31_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 1 && INTVAL (op) <= 31")))

;; Match 2 or 3.
(define_predicate "const_2_to_3_operand"
```

```
  (and (match_code "const_int")
       (match_test "INTVAL (op) == 2 || INTVAL (op) == 3")))

;; Match 4 to 7.
(define_predicate "const_4_to_7_operand"
  (and (match_code "const_int")
       (match_test "INTVAL (op) >= 4 && INTVAL (op) <= 7")))

;; Match exactly one bit in 4-bit mask.
(define_predicate "const_pow2_1_to_8_operand"
  (match_code "const_int")
{
  unsigned int log = exact_log2 (INTVAL (op));
  return log <= 3;
})

;; Match exactly one bit in 8-bit mask.
(define_predicate "const_pow2_1_to_128_operand"
  (match_code "const_int")
{
  unsigned int log = exact_log2 (INTVAL (op));
  return log <= 7;
})

;; True if this is a constant appropriate for an increment or decrement.
(define_predicate "incdec_operand"
  (match_code "const_int")
{
  /* On Pentium4, the inc and dec operations causes extra dependency on flag
     registers, since carry flag is not set.  */
  if ((TARGET_PENTIUM4 || TARGET_NOCONA) && !optimize_size)
    return 0;
  return op == const1_rtx || op == constm1_rtx;
})

;; True for registers, or 1 or -1.  Used to optimize double-word shifts.
(define_predicate "reg_or_pm1_operand"
  (ior (match_operand 0 "register_operand")
       (and (match_code "const_int")
    (match_test "op == const1_rtx || op == constm1_rtx"))))

;; True if OP is acceptable as operand of DImode shift expander.
(define_predicate "shiftdi_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (match_operand 0 "nonimmediate_operand")
    (match_operand 0 "register_operand")))

(define_predicate "ashldi_input_operand"
  (if_then_else (match_test "TARGET_64BIT")
    (match_operand 0 "nonimmediate_operand")
```

```
    (match_operand 0 "reg_or_pm1_operand")))

;; Return true if OP is a vector load from the constant pool with just
;; the first element nonzero.
(define_predicate "zero_extended_scalar_load_operand"
  (match_code "mem")
{
  unsigned n_elts;
  op = maybe_get_pool_constant (op);
  if (!op)
    return 0;
  if (GET_CODE (op) != CONST_VECTOR)
    return 0;
  n_elts =
    (GET_MODE_SIZE (GET_MODE (op)) /
     GET_MODE_SIZE (GET_MODE_INNER (GET_MODE (op))));
  for (n_elts--; n_elts > 0; n_elts--)
    {
      rtx elt = CONST_VECTOR_ELT (op, n_elts);
      if (elt != CONST0_RTX (GET_MODE_INNER (GET_MODE (op))))
return 0;
    }
  return 1;
})

;; Return 1 when OP is operand acceptable for standard SSE move.
(define_predicate "vector_move_operand"
  (ior (match_operand 0 "nonimmediate_operand")
       (match_operand 0 "const0_operand")))

;; Return true if OP is a register or a zero.
(define_predicate "reg_or_0_operand"
  (ior (match_operand 0 "register_operand")
       (match_operand 0 "const0_operand")))

;; Return true if op if a valid address, and does not contain
;; a segment override.
(define_special_predicate "no_seg_address_operand"
  (match_operand 0 "address_operand")
{
  struct ix86_address parts;
  int ok;

  ok = ix86_decompose_address (op, &parts);
  gcc_assert (ok);
  return parts.seg == SEG_DEFAULT;
})

;; Return nonzero if the rtx is known aligned.
(define_predicate "aligned_operand"
```

```
  (match_operand 0 "general_operand")
{
  struct ix86_address parts;
  int ok;

  /* Registers and immediate operands are always "aligned".  */
  if (GET_CODE (op) != MEM)
    return 1;

  /* Don't even try to do any aligned optimizations with volatiles.  */
  if (MEM_VOLATILE_P (op))
    return 0;
  op = XEXP (op, 0);

  /* Pushes and pops are only valid on the stack pointer.  */
  if (GET_CODE (op) == PRE_DEC
      || GET_CODE (op) == POST_INC)
    return 1;

  /* Decode the address.  */
  ok = ix86_decompose_address (op, &parts);
  gcc_assert (ok);

  /* Look for some component that isn't known to be aligned.  */
  if (parts.index)
    {
      if (REGNO_POINTER_ALIGN (REGNO (parts.index)) * parts.scale < 32)
return 0;
    }
  if (parts.base)
    {
      if (REGNO_POINTER_ALIGN (REGNO (parts.base)) < 32)
return 0;
    }
  if (parts.disp)
    {
      if (GET_CODE (parts.disp) != CONST_INT
  || (INTVAL (parts.disp) & 3) != 0)
return 0;
    }

  /* Didn't find one -- this must be an aligned address.  */
  return 1;
})

;; Returns 1 if OP is memory operand with a displacement.
(define_predicate "memory_displacement_operand"
  (match_operand 0 "memory_operand")
{
  struct ix86_address parts;
```

```
  int ok;

  ok = ix86_decompose_address (XEXP (op, 0), &parts);
  gcc_assert (ok);
  return parts.disp != NULL_RTX;
})

;; Returns 1 if OP is memory operand that cannot be represented
;; by the modRM array.
(define_predicate "long_memory_operand"
  (and (match_operand 0 "memory_operand")
       (match_test "memory_address_length (op) != 0")))

;; Return 1 if OP is a comparison operator that can be issued by fcmov.
(define_predicate "fcmov_comparison_operator"
  (match_operand 0 "comparison_operator")
{
  enum machine_mode inmode = GET_MODE (XEXP (op, 0));
  enum rtx_code code = GET_CODE (op);

  if (inmode == CCFPmode || inmode == CCFPUmode)
    {
      enum rtx_code second_code, bypass_code;
      ix86_fp_comparison_codes (code, &bypass_code, &code, &second_code);
      if (bypass_code != UNKNOWN || second_code != UNKNOWN)
return 0;
      code = ix86_fp_compare_code_to_integer (code);
    }
  /* i387 supports just limited amount of conditional codes.  */
  switch (code)
    {
    case LTU: case GTU: case LEU: case GEU:
      if (inmode == CCmode || inmode == CCFPmode || inmode == CCFPUmode)
return 1;
      return 0;
    case ORDERED: case UNORDERED:
    case EQ: case NE:
      return 1;
    default:
      return 0;
    }
})

;; Return 1 if OP is a comparison that can be used in the CMPSS/CMPPS insns.
;; The first set are supported directly; the second set can't be done with
;; full IEEE support, i.e. NaNs.
;;
;; ??? It would seem that we have a lot of uses of this predicate that pass
;; it the wrong mode.  We got away with this because the old function didn't
;; check the mode at all.  Mirror that for now by calling this a special
```

```
;; predicate.

(define_special_predicate "sse_comparison_operator"
  (match_code "eq,lt,le,unordered,ne,unge,ungt,ordered"))

;; Return 1 if OP is a valid comparison operator in valid mode.
(define_predicate "ix86_comparison_operator"
  (match_operand 0 "comparison_operator")
{
  enum machine_mode inmode = GET_MODE (XEXP (op, 0));
  enum rtx_code code = GET_CODE (op);

  if (inmode == CCFPmode || inmode == CCFPUmode)
    {
      enum rtx_code second_code, bypass_code;
      ix86_fp_comparison_codes (code, &bypass_code, &code, &second_code);
      return (bypass_code == UNKNOWN && second_code == UNKNOWN);
    }
  switch (code)
    {
    case EQ: case NE:
      return 1;
    case LT: case GE:
      if (inmode == CCmode || inmode == CCGCmode
  || inmode == CCGOCmode || inmode == CCNOmode)
return 1;
      return 0;
    case LTU: case GTU: case LEU: case ORDERED: case UNORDERED: case GEU:
      if (inmode == CCmode)
return 1;
      return 0;
    case GT: case LE:
      if (inmode == CCmode || inmode == CCGCmode || inmode == CCNOmode)
return 1;
      return 0;
    default:
      return 0;
    }
})

;; Return 1 if OP is a valid comparison operator testing carry flag to be set.
(define_predicate "ix86_carry_flag_operator"
  (match_code "ltu,lt,unlt,gt,ungt,le,unle,ge,unge,ltgt,uneq")
{
  enum machine_mode inmode = GET_MODE (XEXP (op, 0));
  enum rtx_code code = GET_CODE (op);

  if (GET_CODE (XEXP (op, 0)) != REG
      || REGNO (XEXP (op, 0)) != FLAGS_REG
      || XEXP (op, 1) != const0_rtx)
```

```
    return 0;

  if (inmode == CCFPmode || inmode == CCFPUmode)
    {
      enum rtx_code second_code, bypass_code;
      ix86_fp_comparison_codes (code, &bypass_code, &code, &second_code);
      if (bypass_code != UNKNOWN || second_code != UNKNOWN)
return 0;
      code = ix86_fp_compare_code_to_integer (code);
    }
  else if (inmode != CCmode)
    return 0;

  return code == LTU;
})

;; Nearly general operand, but accept any const_double, since we wish
;; to be able to drop them into memory rather than have them get pulled
;; into registers.
(define_predicate "cmp_fp_expander_operand"
  (ior (match_code "const_double")
       (match_operand 0 "general_operand")))

;; Return true if this is a valid binary floating-point operation.
(define_predicate "binary_fp_operator"
  (match_code "plus,minus,mult,div"))

;; Return true if this is a multiply operation.
(define_predicate "mult_operator"
  (match_code "mult"))

;; Return true if this is a division operation.
(define_predicate "div_operator"
  (match_code "div"))

;; Return true if this is a float extend operation.
(define_predicate "float_operator"
  (match_code "float"))

;; Return true for ARITHMETIC_P.
(define_predicate "arith_or_logical_operator"
  (match_code "plus,mult,and,ior,xor,smin,smax,umin,umax,compare,minus,div,
       mod,udiv,umod,ashift,rotate,ashiftrt,lshiftrt,rotatert"))

;; Return 1 if OP is a binary operator that can be promoted to wider mode.
;; Modern CPUs have same latency for HImode and SImode multiply,
;; but 386 and 486 do HImode multiply faster.  */
(define_predicate "promotable_binary_operator"
  (ior (match_code "plus,and,ior,xor,ashift")
       (and (match_code "mult")
```

```
    (match_test "ix86_tune > PROCESSOR_I486"))))

;; To avoid problems when jump re-emits comparisons like testqi_ext_ccno_0,
;; re-recognize the operand to avoid a copy_to_mode_reg that will fail.
;;
;; ??? It seems likely that this will only work because cmpsi is an
;; expander, and no actual insns use this.

(define_predicate "cmpsi_operand_1"
  (match_code "and")
{
  return (GET_MODE (op) == SImode
  && GET_CODE (XEXP (op, 0)) == ZERO_EXTRACT
  && GET_CODE (XEXP (XEXP (op, 0), 1)) == CONST_INT
  && GET_CODE (XEXP (XEXP (op, 0), 2)) == CONST_INT
  && INTVAL (XEXP (XEXP (op, 0), 1)) == 8
  && INTVAL (XEXP (XEXP (op, 0), 2)) == 8
  && GET_CODE (XEXP (op, 1)) == CONST_INT);
})

(define_predicate "cmpsi_operand"
  (ior (match_operand 0 "nonimmediate_operand")
       (match_operand 0 "cmpsi_operand_1")))

(define_predicate "compare_operator"
  (match_code "compare"))

(define_predicate "absneg_operator"
  (match_code "abs,neg"))
```

1.7 Compare instructions

```
;; All compare insns have expanders that save the operands away without
;; actually generating RTL.  The bCOND or sCOND (emitted immediately
;; after the cmp) will actually emit the cmpM.

(define_expand "cmpti"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:TI 0 "nonimmediate_operand" "")
    (match_operand:TI 1 "x86_64_general_operand" "")))]
  "TARGET_64BIT"
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (TImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})
```

```
(define_expand "cmpdi"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:DI 0 "nonimmediate_operand" "")
    (match_operand:DI 1 "x86_64_general_operand" "")))]
  ""
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (DImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

(define_expand "cmpsi"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:SI 0 "cmpsi_operand" "")
    (match_operand:SI 1 "general_operand" "")))]
  ""
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (SImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

(define_expand "cmphi"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:HI 0 "nonimmediate_operand" "")
    (match_operand:HI 1 "general_operand" "")))]
  ""
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (HImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

(define_expand "cmpqi"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:QI 0 "nonimmediate_operand" "")
    (match_operand:QI 1 "general_operand" "")))]
  "TARGET_QIMODE_MATH"
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (QImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
```

```
})

(define_insn "cmpdi_ccno_1_rex64"
  [(set (reg FLAGS_REG)
(compare (match_operand:DI 0 "nonimmediate_operand" "r,?mr")
 (match_operand:DI 1 "const0_operand" "n,n")))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  "@
   test{q}\t{%0, %0|%0, %0}
   cmp{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test,icmp")
   (set_attr "length_immediate" "0,1")
   (set_attr "mode" "DI")])

(define_insn "*cmpdi_minus_1_rex64"
  [(set (reg FLAGS_REG)
(compare (minus:DI (match_operand:DI 0 "nonimmediate_operand" "rm,r")
   (match_operand:DI 1 "x86_64_general_operand" "re,mr"))
 (const_int 0)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)"
  "cmp{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "DI")])

(define_expand "cmpdi_1_rex64"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:DI 0 "nonimmediate_operand" "")
    (match_operand:DI 1 "general_operand" "")))]
  "TARGET_64BIT"
  "")

(define_insn "cmpdi_1_insn_rex64"
  [(set (reg FLAGS_REG)
(compare (match_operand:DI 0 "nonimmediate_operand" "mr,r")
 (match_operand:DI 1 "x86_64_general_operand" "re,mr")))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "DI")])


(define_insn "*cmpsi_ccno_1"
  [(set (reg FLAGS_REG)
(compare (match_operand:SI 0 "nonimmediate_operand" "r,?mr")
 (match_operand:SI 1 "const0_operand" "n,n")))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "@
   test{l}\t{%0, %0|%0, %0}
   cmp{l}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test,icmp")
```

```
   (set_attr "length_immediate" "0,1")
   (set_attr "mode" "SI")])

(define_insn "*cmpsi_minus_1"
  [(set (reg FLAGS_REG)
(compare (minus:SI (match_operand:SI 0 "nonimmediate_operand" "rm,r")
   (match_operand:SI 1 "general_operand" "ri,mr"))
 (const_int 0)))]
  "ix86_match_ccmode (insn, CCGOCmode)"
  "cmp{l}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "SI")])

(define_expand "cmpsi_1"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:SI 0 "nonimmediate_operand" "rm,r")
    (match_operand:SI 1 "general_operand" "ri,mr")))]
  ""
  "")

(define_insn "*cmpsi_1_insn"
  [(set (reg FLAGS_REG)
(compare (match_operand:SI 0 "nonimmediate_operand" "rm,r")
 (match_operand:SI 1 "general_operand" "ri,mr")))]
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
    && ix86_match_ccmode (insn, CCmode)"
  "cmp{l}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "SI")])

(define_insn "*cmphi_ccno_1"
  [(set (reg FLAGS_REG)
(compare (match_operand:HI 0 "nonimmediate_operand" "r,?mr")
 (match_operand:HI 1 "const0_operand" "n,n")))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "@
   test{w}\t{%0, %0|%0, %0}
   cmp{w}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test,icmp")
   (set_attr "length_immediate" "0,1")
   (set_attr "mode" "HI")])

(define_insn "*cmphi_minus_1"
  [(set (reg FLAGS_REG)
(compare (minus:HI (match_operand:HI 0 "nonimmediate_operand" "rm,r")
   (match_operand:HI 1 "general_operand" "ri,mr"))
 (const_int 0)))]
  "ix86_match_ccmode (insn, CCGOCmode)"
  "cmp{w}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
```

```
   (set_attr "mode" "HI")])

(define_insn "*cmphi_1"
  [(set (reg FLAGS_REG)
(compare (match_operand:HI 0 "nonimmediate_operand" "rm,r")
 (match_operand:HI 1 "general_operand" "ri,mr")))]
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && ix86_match_ccmode (insn, CCmode)"
  "cmp{w}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "HI")])

(define_insn "*cmpqi_ccno_1"
  [(set (reg FLAGS_REG)
(compare (match_operand:QI 0 "nonimmediate_operand" "q,?mq")
 (match_operand:QI 1 "const0_operand" "n,n")))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "@
   test{b}\t{%0, %0|%0, %0}
   cmp{b}\t{$0, %0|%0, 0}"
  [(set_attr "type" "test,icmp")
   (set_attr "length_immediate" "0,1")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_1"
  [(set (reg FLAGS_REG)
(compare (match_operand:QI 0 "nonimmediate_operand" "qm,q")
 (match_operand:QI 1 "general_operand" "qi,mq")))]
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
    && ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_minus_1"
  [(set (reg FLAGS_REG)
(compare (minus:QI (match_operand:QI 0 "nonimmediate_operand" "qm,q")
   (match_operand:QI 1 "general_operand" "qi,mq"))
 (const_int 0)))]
  "ix86_match_ccmode (insn, CCGOCmode)"
  "cmp{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_ext_1"
  [(set (reg FLAGS_REG)
(compare
  (match_operand:QI 0 "general_operand" "Qm")
  (subreg:QI
    (zero_extract:SI
```

```
      (match_operand 1 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)))]
  "!TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%h1, %0|%0, %h1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_ext_1_rex64"
  [(set (reg FLAGS_REG)
(compare
  (match_operand:QI 0 "register_operand" "Q")
  (subreg:QI
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%h1, %0|%0, %h1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_ext_2"
  [(set (reg FLAGS_REG)
(compare
  (subreg:QI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)
  (match_operand:QI 1 "const0_operand" "n")))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "test{b}\t%h0, %h0"
  [(set_attr "type" "test")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_expand "cmpqi_ext_3"
  [(set (reg:CC FLAGS_REG)
(compare:CC
  (subreg:QI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "")
      (const_int 8)
      (const_int 8)) 0)
  (match_operand:QI 1 "general_operand" "")))]
  ""
  "")

(define_insn "cmpqi_ext_3_insn"
```

```
  [(set (reg FLAGS_REG)
(compare
  (subreg:QI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)
  (match_operand:QI 1 "general_operand" "Qmn")))]
  "!TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%1, %h0|%h0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "cmpqi_ext_3_insn_rex64"
  [(set (reg FLAGS_REG)
(compare
  (subreg:QI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)
  (match_operand:QI 1 "nonmemory_operand" "Qn")))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%1, %h0|%h0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

(define_insn "*cmpqi_ext_4"
  [(set (reg FLAGS_REG)
(compare
  (subreg:QI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)
  (subreg:QI
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)) 0)))]
  "ix86_match_ccmode (insn, CCmode)"
  "cmp{b}\t{%h1, %h0|%h0, %h1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "QI")])

;; These implement float point compares.
;; %%% See if we can get away with VOIDmode operands on the actual insns,
;; which would allow mix and match FP modes on the compares.  Which is what
;; the old patterns did, but with many more of them.
```

```
(define_expand "cmpxf"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:XF 0 "nonmemory_operand" "")
    (match_operand:XF 1 "nonmemory_operand" "")))]
  "TARGET_80387"
{
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

(define_expand "cmpdf"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:DF 0 "cmp_fp_expander_operand" "")
    (match_operand:DF 1 "cmp_fp_expander_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
{
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

(define_expand "cmpsf"
  [(set (reg:CC FLAGS_REG)
(compare:CC (match_operand:SF 0 "cmp_fp_expander_operand" "")
    (match_operand:SF 1 "cmp_fp_expander_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
{
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})

;; FP compares, step 1:
;; Set the FP condition codes.
;;
;; CCFPmode compare with exceptions
;; CCFPUmode compare with no exceptions

;; We may not use "#" to split and emit these, since the REG_DEAD notes
;; used to manage the reg stack popping would not be preserved.

(define_insn "*cmpfp_0"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFP
     (match_operand 1 "register_operand" "f")
     (match_operand 2 "const0_operand" "X"))]
UNSPEC_FNSTSW))]
  "TARGET_80387
```

```
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])"
  "* return output_fp_compare (insn, operands, 0, 0);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
   (set (attr "mode")
     (cond [(match_operand:SF 1 "" "")
      (const_string "SF")
    (match_operand:DF 1 "" "")
      (const_string "DF")
   ]
   (const_string "XF")))])

(define_insn "*cmpfp_sf"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFP
     (match_operand:SF 1 "register_operand" "f")
     (match_operand:SF 2 "nonimmediate_operand" "fm"))]
  UNSPEC_FNSTSW))]
  "TARGET_80387"
  "* return output_fp_compare (insn, operands, 0, 0);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
   (set_attr "mode" "SF")])

(define_insn "*cmpfp_df"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFP
     (match_operand:DF 1 "register_operand" "f")
     (match_operand:DF 2 "nonimmediate_operand" "fm"))]
  UNSPEC_FNSTSW))]
  "TARGET_80387"
  "* return output_fp_compare (insn, operands, 0, 0);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
   (set_attr "mode" "DF")])

(define_insn "*cmpfp_xf"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFP
     (match_operand:XF 1 "register_operand" "f")
     (match_operand:XF 2 "register_operand" "f"))]
  UNSPEC_FNSTSW))]
  "TARGET_80387"
  "* return output_fp_compare (insn, operands, 0, 0);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
```

```
   (set_attr "mode" "XF")])

(define_insn "*cmpfp_u"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFPU
     (match_operand 1 "register_operand" "f")
     (match_operand 2 "register_operand" "f"))]
  UNSPEC_FNSTSW))]
  "TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])"
  "* return output_fp_compare (insn, operands, 0, 1);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
   (set (attr "mode")
     (cond [(match_operand:SF 1 "" "")
      (const_string "SF")
    (match_operand:DF 1 "" "")
      (const_string "DF")
   ]
   (const_string "XF")))])

(define_insn "*cmpfp_<mode>"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI
  [(compare:CCFP
     (match_operand 1 "register_operand" "f")
     (match_operator 3 "float_operator"
       [(match_operand:X87MODEI12 2 "memory_operand" "m")]))]
  UNSPEC_FNSTSW))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && (GET_MODE (operands [3]) == GET_MODE (operands[1]))"
  "* return output_fp_compare (insn, operands, 0, 0);"
  [(set_attr "type" "multi")
   (set_attr "unit" "i387")
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])

;; FP compares, step 2
;; Move the fpsw to ax.

(define_insn "x86_fnstsw_1"
  [(set (match_operand:HI 0 "register_operand" "=a")
(unspec:HI [(reg:CCFP FPSR_REG)] UNSPEC_FNSTSW))]
  "TARGET_80387"
  "fnstsw\t%0"
  [(set_attr "length" "2")
   (set_attr "mode" "SI")
```

```
   (set_attr "unit" "i387")])

;; FP compares, step 3
;; Get ax into flags, general case.

(define_insn "x86_sahf_1"
  [(set (reg:CC FLAGS_REG)
(unspec:CC [(match_operand:HI 0 "register_operand" "a")] UNSPEC_SAHF))]
  "!TARGET_64BIT"
  "sahf"
  [(set_attr "length" "1")
   (set_attr "athlon_decode" "vector")
   (set_attr "mode" "SI")])

;; Pentium Pro can do steps 1 through 3 in one go.

(define_insn "*cmpfp_i_mixed"
  [(set (reg:CCFP FLAGS_REG)
(compare:CCFP (match_operand 0 "register_operand" "f#x,x#f")
      (match_operand 1 "nonimmediate_operand" "f#x,xm#f")))]
  "TARGET_MIX_SSE_I387
   && SSE_FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 0);"
  [(set_attr "type" "fcmp,ssecomi")
   (set (attr "mode")
     (if_then_else (match_operand:SF 1 "" "")
        (const_string "SF")
        (const_string "DF")))
   (set_attr "athlon_decode" "vector")])

(define_insn "*cmpfp_i_sse"
  [(set (reg:CCFP FLAGS_REG)
(compare:CCFP (match_operand 0 "register_operand" "x")
      (match_operand 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE_MATH
   && SSE_FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 0);"
  [(set_attr "type" "ssecomi")
   (set (attr "mode")
     (if_then_else (match_operand:SF 1 "" "")
        (const_string "SF")
        (const_string "DF")))
   (set_attr "athlon_decode" "vector")])

(define_insn "*cmpfp_i_i387"
  [(set (reg:CCFP FLAGS_REG)
(compare:CCFP (match_operand 0 "register_operand" "f")
      (match_operand 1 "register_operand" "f")))]
```

```
  "TARGET_80387 && TARGET_CMOVE
   && (!TARGET_SSE_MATH || !SSE_FLOAT_MODE_P (GET_MODE (operands[0])))
   && FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 0);"
  [(set_attr "type" "fcmp")
   (set (attr "mode")
     (cond [(match_operand:SF 1 "" "")
      (const_string "SF")
    (match_operand:DF 1 "" "")
      (const_string "DF")
   ]
   (const_string "XF")))
   (set_attr "athlon_decode" "vector")])

(define_insn "*cmpfp_iu_mixed"
  [(set (reg:CCFPU FLAGS_REG)
(compare:CCFPU (match_operand 0 "register_operand" "f#x,x#f")
       (match_operand 1 "nonimmediate_operand" "f#x,xm#f")))]
  "TARGET_MIX_SSE_I387
   && SSE_FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 1);"
  [(set_attr "type" "fcmp,ssecomi")
   (set (attr "mode")
     (if_then_else (match_operand:SF 1 "" "")
        (const_string "SF")
        (const_string "DF")))
   (set_attr "athlon_decode" "vector")])

(define_insn "*cmpfp_iu_sse"
  [(set (reg:CCFPU FLAGS_REG)
(compare:CCFPU (match_operand 0 "register_operand" "x")
       (match_operand 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE_MATH
   && SSE_FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 1);"
  [(set_attr "type" "ssecomi")
   (set (attr "mode")
     (if_then_else (match_operand:SF 1 "" "")
        (const_string "SF")
        (const_string "DF")))
   (set_attr "athlon_decode" "vector")])

(define_insn "*cmpfp_iu_387"
  [(set (reg:CCFPU FLAGS_REG)
(compare:CCFPU (match_operand 0 "register_operand" "f")
       (match_operand 1 "register_operand" "f")))]
  "TARGET_80387 && TARGET_CMOVE
```

```
   && (!TARGET_SSE_MATH || !SSE_FLOAT_MODE_P (GET_MODE (operands[0])))
   && FLOAT_MODE_P (GET_MODE (operands[0]))
   && GET_MODE (operands[0]) == GET_MODE (operands[1])"
  "* return output_fp_compare (insn, operands, 1, 1);"
  [(set_attr "type" "fcmp")
   (set (attr "mode")
     (cond [(match_operand:SF 1 "" "")
      (const_string "SF")
    (match_operand:DF 1 "" "")
      (const_string "DF")
   ]
   (const_string "XF")))
   (set_attr "athlon_decode" "vector")])

;; Move instructions.

;; General case of fullword move.

(define_expand "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(match_operand:SI 1 "general_operand" ""))]
  ""
  "ix86_expand_move (SImode, operands); DONE;")

;; Push/pop instructions.  They are separate since autoinc/dec is not a
;; general_operand.
;;
;; %%% We don't use a post-inc memory reference because x86 is not a
;; general AUTO_INC_DEC host, which impacts how it is treated in flow.
;; Changing this impacts compiler performance on other non-AUTO_INC_DEC
;; targets without our curiosities, and it is just as easy to represent
;; this differently.

(define_insn "*pushsi2"
  [(set (match_operand:SI 0 "push_operand" "=<")
(match_operand:SI 1 "general_no_elim_operand" "ri*m"))]
  "!TARGET_64BIT"
  "push{l}\t%1"
  [(set_attr "type" "push")
   (set_attr "mode" "SI")])

;; For 64BIT abi we always round up to 8 bytes.
(define_insn "*pushsi2_rex64"
  [(set (match_operand:SI 0 "push_operand" "=X")
(match_operand:SI 1 "nonmemory_no_elim_operand" "ri"))]
  "TARGET_64BIT"
  "push{q}\t%q1"
  [(set_attr "type" "push")
   (set_attr "mode" "SI")])
```

```
(define_insn "*pushsi2_prologue"
  [(set (match_operand:SI 0 "push_operand" "=<")
(match_operand:SI 1 "general_no_elim_operand" "ri*m"))
   (clobber (mem:BLK (scratch)))]
  "!TARGET_64BIT"
  "push{l}\t%1"
  [(set_attr "type" "push")
   (set_attr "mode" "SI")])

(define_insn "*popsi1_epilogue"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r*m")
(mem:SI (reg:SI SP_REG)))
   (set (reg:SI SP_REG)
(plus:SI (reg:SI SP_REG) (const_int 4)))
   (clobber (mem:BLK (scratch)))]
  "!TARGET_64BIT"
  "pop{l}\t%0"
  [(set_attr "type" "pop")
   (set_attr "mode" "SI")])

(define_insn "popsi1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r*m")
(mem:SI (reg:SI SP_REG)))
   (set (reg:SI SP_REG)
(plus:SI (reg:SI SP_REG) (const_int 4)))]
  "!TARGET_64BIT"
  "pop{l}\t%0"
  [(set_attr "type" "pop")
   (set_attr "mode" "SI")])

(define_insn "*movsi_xor"
  [(set (match_operand:SI 0 "register_operand" "=r")
(match_operand:SI 1 "const0_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed && (!TARGET_USE_MOV0 || optimize_size)"
  "xor{l}\t{%0, %0|%0, %0}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")
   (set_attr "length_immediate" "0")])

(define_insn "*movsi_or"
  [(set (match_operand:SI 0 "register_operand" "=r")
(match_operand:SI 1 "immediate_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && operands[1] == constm1_rtx
   && (TARGET_PENTIUM || optimize_size)"
{
  operands[1] = constm1_rtx;
  return "or{l}\t{%1, %0|%0, %1}";
```

```
}
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")
   (set_attr "length_immediate" "1")])

(define_insn "*movsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand"
"=r  ,m  ,*y,*y,?rm,?*y,*x,*x,?r,m ,?*Y,*x")
(match_operand:SI 1 "general_operand"
"rinm,rin,C ,*y,*y ,rm ,C ,*x,*Y,*x,r  ,m "))]
  "!(MEM_P (operands[0]) && MEM_P (operands[1]))"
{
  switch (get_attr_type (insn))
    {
    case TYPE_SSELOG1:
      if (get_attr_mode (insn) == MODE_TI)
        return "pxor\t%0, %0";
      return "xorps\t%0, %0";

    case TYPE_SSEMOV:
      switch (get_attr_mode (insn))
{
case MODE_TI:
  return "movdqa\t{%1, %0|%0, %1}";
case MODE_V4SF:
  return "movaps\t{%1, %0|%0, %1}";
case MODE_SI:
          return "movd\t{%1, %0|%0, %1}";
case MODE_SF:
          return "movss\t{%1, %0|%0, %1}";
default:
  gcc_unreachable ();
}

    case TYPE_MMXADD:
      return "pxor\t%0, %0";

    case TYPE_MMXMOV:
      if (get_attr_mode (insn) == MODE_DI)
return "movq\t{%1, %0|%0, %1}";
      return "movd\t{%1, %0|%0, %1}";

    case TYPE_LEA:
      return "lea{l}\t{%1, %0|%0, %1}";

    default:
      gcc_assert (!flag_pic || LEGITIMATE_PIC_OPERAND_P (operands[1]));
      return "mov{l}\t{%1, %0|%0, %1}";
    }
}
```

```
  [(set (attr "type")
     (cond [(eq_attr "alternative" "2")
      (const_string "mmx")
    (eq_attr "alternative" "3,4,5")
      (const_string "mmxmov")
    (eq_attr "alternative" "6")
      (const_string "sselog1")
    (eq_attr "alternative" "7,8,9,10,11")
      (const_string "ssemov")
      (match_operand:DI 1 "pic_32bit_operand" "")
      (const_string "lea")
   ]
   (const_string "imov")))
   (set (attr "mode")
     (cond [(eq_attr "alternative" "2,3")
      (const_string "DI")
    (eq_attr "alternative" "6,7")
      (if_then_else
        (eq (symbol_ref "TARGET_SSE2") (const_int 0))
        (const_string "V4SF")
        (const_string "TI"))
    (and (eq_attr "alternative" "8,9,10,11")
         (eq (symbol_ref "TARGET_SSE2") (const_int 0)))
      (const_string "SF")
   ]
   (const_string "SI")))])

;; Stores and loads of ax to arbitrary constant address.
;; We fake an second form of instruction to force reload to load address
;; into register when rax is not available
(define_insn "*movabssi_1_rex64"
  [(set (mem:SI (match_operand:DI 0 "x86_64_movabs_operand" "i,r"))
(match_operand:SI 1 "nonmemory_operand" "a,er"))]
  "TARGET_64BIT && ix86_check_movabs (insn, 0)"
  "@
   movabs{l}\t{%1, %P0|%P0, %1}
   mov{l}\t{%1, %a0|%a0, %1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0,*")
   (set_attr "memory" "store")
   (set_attr "mode" "SI")])

(define_insn "*movabssi_2_rex64"
  [(set (match_operand:SI 0 "register_operand" "=a,r")
        (mem:SI (match_operand:DI 1 "x86_64_movabs_operand" "i,r")))]
  "TARGET_64BIT && ix86_check_movabs (insn, 1)"
  "@
   movabs{l}\t{%P1, %0|%0, %P1}
```

```
   mov{l}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0")
   (set_attr "memory" "load")
   (set_attr "mode" "SI")])

(define_insn "*swapsi"
  [(set (match_operand:SI 0 "register_operand" "+r")
(match_operand:SI 1 "register_operand" "+r"))
   (set (match_dup 1)
(match_dup 0))]
  ""
  "xchg{l}\t%1, %0"
  [(set_attr "type" "imov")
   (set_attr "mode" "SI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_expand "movhi"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "ix86_expand_move (HImode, operands); DONE;")

(define_insn "*pushhi2"
  [(set (match_operand:HI 0 "push_operand" "=<,<")
(match_operand:HI 1 "general_no_elim_operand" "n,r*m"))]
  "!TARGET_64BIT"
  "@
   push{w}\t{|WORD PTR }%1
   push{w}\t%1"
  [(set_attr "type" "push")
   (set_attr "mode" "HI")])

;; For 64BIT abi we always round up to 8 bytes.
(define_insn "*pushhi2_rex64"
  [(set (match_operand:HI 0 "push_operand" "=X")
(match_operand:HI 1 "nonmemory_no_elim_operand" "ri"))]
  "TARGET_64BIT"
  "push{q}\t%q1"
  [(set_attr "type" "push")
   (set_attr "mode" "QI")])

(define_insn "*movhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=r,r,r,m")
(match_operand:HI 1 "general_operand" "r,rn,rm,rn"))]
  "GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM"
{
```

```
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      /* movzwl is faster than movw on p2 due to partial word stalls,
 though not as fast as an aligned movl.  */
      return "movz{wl|x}\t{%1, %k0|%k0, %1}";
    default:
      if (get_attr_mode (insn) == MODE_SI)
        return "mov{l}\t{%k1, %k0|%k0, %k1}";
      else
        return "mov{w}\t{%1, %0|%0, %1}";
    }
}
  [(set (attr "type")
     (cond [(ne (symbol_ref "optimize_size") (const_int 0))
      (const_string "imov")
    (and (eq_attr "alternative" "0")
 (ior (eq (symbol_ref "TARGET_PARTIAL_REG_STALL")
  (const_int 0))
      (eq (symbol_ref "TARGET_HIMODE_MATH")
  (const_int 0))))
      (const_string "imov")
    (and (eq_attr "alternative" "1,2")
 (match_operand:HI 1 "aligned_operand" ""))
      (const_string "imov")
    (and (ne (symbol_ref "TARGET_MOVX")
     (const_int 0))
 (eq_attr "alternative" "0,2"))
      (const_string "imovx")
   ]
   (const_string "imov")))
    (set (attr "mode")
      (cond [(eq_attr "type" "imovx")
       (const_string "SI")
     (and (eq_attr "alternative" "1,2")
  (match_operand:HI 1 "aligned_operand" ""))
       (const_string "SI")
      (and (eq_attr "alternative" "0")
  (ior (eq (symbol_ref "TARGET_PARTIAL_REG_STALL")
   (const_int 0))
       (eq (symbol_ref "TARGET_HIMODE_MATH")
   (const_int 0))))
       (const_string "SI")
    ]
    (const_string "HI")))])

;; Stores and loads of ax to arbitrary constant address.
;; We fake an second form of instruction to force reload to load address
;; into register when rax is not available
(define_insn "*movabshi_1_rex64"
```

```
  [(set (mem:HI (match_operand:DI 0 "x86_64_movabs_operand" "i,r"))
(match_operand:HI 1 "nonmemory_operand" "a,er"))]
  "TARGET_64BIT && ix86_check_movabs (insn, 0)"
  "@
   movabs{w}\t{%1, %P0|%P0, %1}
   mov{w}\t{%1, %a0|%a0, %1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0,*")
   (set_attr "memory" "store")
   (set_attr "mode" "HI")])

(define_insn "*movabshi_2_rex64"
  [(set (match_operand:HI 0 "register_operand" "=a,r")
        (mem:HI (match_operand:DI 1 "x86_64_movabs_operand" "i,r")))]
  "TARGET_64BIT && ix86_check_movabs (insn, 1)"
  "@
   movabs{w}\t{%P1, %0|%0, %P1}
   mov{w}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0")
   (set_attr "memory" "load")
   (set_attr "mode" "HI")])

(define_insn "*swaphi_1"
  [(set (match_operand:HI 0 "register_operand" "+r")
(match_operand:HI 1 "register_operand" "+r"))
   (set (match_dup 1)
(match_dup 0))]
  "!TARGET_PARTIAL_REG_STALL || optimize_size"
  "xchg{l}\t%k1, %k0"
  [(set_attr "type" "imov")
   (set_attr "mode" "SI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_insn "*swaphi_2"
  [(set (match_operand:HI 0 "register_operand" "+r")
(match_operand:HI 1 "register_operand" "+r"))
   (set (match_dup 1)
(match_dup 0))]
  "TARGET_PARTIAL_REG_STALL"
  "xchg{w}\t%1, %0"
  [(set_attr "type" "imov")
   (set_attr "mode" "HI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])
```

```
(define_expand "movstricthi"
  [(set (strict_low_part (match_operand:HI 0 "nonimmediate_operand" ""))
(match_operand:HI 1 "general_operand" ""))]
  "! TARGET_PARTIAL_REG_STALL || optimize_size"
{
  /* Don't generate memory->memory moves, go through a register */
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (HImode, operands[1]);
})

(define_insn "*movstricthi_1"
  [(set (strict_low_part (match_operand:HI 0 "nonimmediate_operand" "+rm,r"))
(match_operand:HI 1 "general_operand" "rn,m"))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "mov{w}\t{%1, %0|%0, %1}"
  [(set_attr "type" "imov")
   (set_attr "mode" "HI")])

(define_insn "*movstricthi_xor"
  [(set (strict_low_part (match_operand:HI 0 "register_operand" "+r"))
(match_operand:HI 1 "const0_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && ((!TARGET_USE_MOV0 && !TARGET_PARTIAL_REG_STALL) || optimize_size)"
  "xor{w}\t{%0, %0|%0, %0}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "HI")
   (set_attr "length_immediate" "0")])

(define_expand "movqi"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(match_operand:QI 1 "general_operand" ""))]
  ""
  "ix86_expand_move (QImode, operands); DONE;")

;; emit_push_insn when it calls move_by_pieces requires an insn to
;; "push a byte".  But actually we use pushw, which has the effect
;; of rounding the amount pushed up to a halfword.

(define_insn "*pushqi2"
  [(set (match_operand:QI 0 "push_operand" "=X,X")
(match_operand:QI 1 "nonmemory_no_elim_operand" "n,r"))]
  "!TARGET_64BIT"
  "@
   push{w}\t{|word ptr }%1
   push{w}\t%w1"
  [(set_attr "type" "push")
   (set_attr "mode" "HI")])
```

```
;; For 64BIT abi we always round up to 8 bytes.
(define_insn "*pushqi2_rex64"
  [(set (match_operand:QI 0 "push_operand" "=X")
(match_operand:QI 1 "nonmemory_no_elim_operand" "qi"))]
  "TARGET_64BIT"
  "push{q}\t%q1"
  [(set_attr "type" "push")
   (set_attr "mode" "QI")])

;; Situation is quite tricky about when to choose full sized (SImode) move
;; over QImode moves.  For Q_REG -> Q_REG move we use full size only for
;; partial register dependency machines (such as AMD Athlon), where QImode
;; moves issue extra dependency and for partial register stalls machines
;; that don't use QImode patterns (and QImode move cause stall on the next
;; instruction).
;;
;; For loads of Q_REG to NONQ_REG we use full sized moves except for partial
;; register stall machines with, where we use QImode instructions, since
;; partial register stall can be caused there.  Then we use movzx.
(define_insn "*movqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=q,q ,q ,r,r ,?r,m")
(match_operand:QI 1 "general_operand"      " q,qn,qm,q,rn,m ,qn"))]
  "GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      gcc_assert (ANY_QI_REG_P (operands[1]) || GET_CODE (operands[1]) == MEM);
      return "movz{bl|x}\t{%1, %k0|%k0, %1}";
    default:
      if (get_attr_mode (insn) == MODE_SI)
        return "mov{l}\t{%k1, %k0|%k0, %k1}";
      else
        return "mov{b}\t{%1, %0|%0, %1}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "5")
      (const_string "imovx")
    (ne (symbol_ref "optimize_size") (const_int 0))
      (const_string "imov")
    (and (eq_attr "alternative" "3")
 (ior (eq (symbol_ref "TARGET_PARTIAL_REG_STALL")
  (const_int 0))
      (eq (symbol_ref "TARGET_QIMODE_MATH")
  (const_int 0))))
      (const_string "imov")
    (eq_attr "alternative" "3")
      (const_string "imovx")
```

```
    (and (ne (symbol_ref "TARGET_MOVX")
     (const_int 0))
 (eq_attr "alternative" "2"))
      (const_string "imovx")
   ]
   (const_string "imov")))
   (set (attr "mode")
      (cond [(eq_attr "alternative" "3,4,5")
       (const_string "SI")
     (eq_attr "alternative" "6")
       (const_string "QI")
     (eq_attr "type" "imovx")
       (const_string "SI")
     (and (eq_attr "type" "imov")
  (and (eq_attr "alternative" "0,1")
       (ne (symbol_ref "TARGET_PARTIAL_REG_DEPENDENCY")
   (const_int 0))))
       (const_string "SI")
     ;; Avoid partial register stalls when not using QImode arithmetic
     (and (eq_attr "type" "imov")
  (and (eq_attr "alternative" "0,1")
       (and (ne (symbol_ref "TARGET_PARTIAL_REG_STALL")
(const_int 0))
    (eq (symbol_ref "TARGET_QIMODE_MATH")
(const_int 0)))))
       (const_string "SI")
   ]
   (const_string "QI")))])

(define_expand "reload_outqi"
  [(parallel [(match_operand:QI 0 "" "=m")
              (match_operand:QI 1 "register_operand" "r")
              (match_operand:QI 2 "register_operand" "=&q")])]
  ""
{
  rtx op0, op1, op2;
  op0 = operands[0]; op1 = operands[1]; op2 = operands[2];

  gcc_assert (!reg_overlap_mentioned_p (op2, op0));
  if (! q_regs_operand (op1, QImode))
    {
      emit_insn (gen_movqi (op2, op1));
      op1 = op2;
    }
  emit_insn (gen_movqi (op0, op1));
  DONE;
})

(define_insn "*swapqi_1"
  [(set (match_operand:QI 0 "register_operand" "+r")
```

```
(match_operand:QI 1 "register_operand" "+r"))
   (set (match_dup 1)
(match_dup 0))]
  "!TARGET_PARTIAL_REG_STALL || optimize_size"
  "xchg{l}\t%k1, %k0"
  [(set_attr "type" "imov")
   (set_attr "mode" "SI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_insn "*swapqi_2"
  [(set (match_operand:QI 0 "register_operand" "+q")
(match_operand:QI 1 "register_operand" "+q"))
   (set (match_dup 1)
(match_dup 0))]
  "TARGET_PARTIAL_REG_STALL"
  "xchg{b}\t%1, %0"
  [(set_attr "type" "imov")
   (set_attr "mode" "QI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_expand "movstrictqi"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" ""))
(match_operand:QI 1 "general_operand" ""))]
  "! TARGET_PARTIAL_REG_STALL || optimize_size"
{
  /* Don't generate memory->memory moves, go through a register.  */
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (QImode, operands[1]);
})

(define_insn "*movstrictqi_1"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,q"))
(match_operand:QI 1 "general_operand" "*qn,m"))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "mov{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "imov")
   (set_attr "mode" "QI")])

(define_insn "*movstrictqi_xor"
  [(set (strict_low_part (match_operand:QI 0 "q_regs_operand" "+q"))
(match_operand:QI 1 "const0_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed && (!TARGET_USE_MOV0 || optimize_size)"
  "xor{b}\t{%0, %0|%0, %0}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")
   (set_attr "length_immediate" "0")])
```

```
(define_insn "*movsi_extv_1"
  [(set (match_operand:SI 0 "register_operand" "=R")
(sign_extract:SI (match_operand 1 "ext_register_operand" "Q")
 (const_int 8)
 (const_int 8)))]
  ""
  "movs{bl|x}\t{%h1, %0|%0, %h1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")])

(define_insn "*movhi_extv_1"
  [(set (match_operand:HI 0 "register_operand" "=R")
(sign_extract:HI (match_operand 1 "ext_register_operand" "Q")
 (const_int 8)
 (const_int 8)))]
  ""
  "movs{bl|x}\t{%h1, %k0|%k0, %h1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")])

(define_insn "*movqi_extv_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=Qm,?r")
        (sign_extract:QI (match_operand 1 "ext_register_operand" "Q,Q")
                         (const_int 8)
                         (const_int 8)))]
  "!TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      return "movs{bl|x}\t{%h1, %k0|%k0, %h1}";
    default:
      return "mov{b}\t{%h1, %0|%0, %h1}";
    }
}
  [(set (attr "type")
     (if_then_else (and (match_operand:QI 0 "register_operand" "")
(ior (not (match_operand:QI 0 "q_regs_operand" ""))
     (ne (symbol_ref "TARGET_MOVX")
 (const_int 0))))
(const_string "imovx")
(const_string "imov")))
   (set (attr "mode")
     (if_then_else (eq_attr "type" "imovx")
(const_string "SI")
(const_string "QI")))])

(define_insn "*movqi_extv_1_rex64"
  [(set (match_operand:QI 0 "register_operand" "=Q,?R")
```

```
        (sign_extract:QI (match_operand 1 "ext_register_operand" "Q,Q")
                         (const_int 8)
                         (const_int 8)))]
  "TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      return "movs{bl|x}\t{%h1, %k0|%k0, %h1}";
    default:
      return "mov{b}\t{%h1, %0|%0, %h1}";
    }
}
  [(set (attr "type")
     (if_then_else (and (match_operand:QI 0 "register_operand" "")
(ior (not (match_operand:QI 0 "q_regs_operand" ""))
     (ne (symbol_ref "TARGET_MOVX")
 (const_int 0))))
(const_string "imovx")
(const_string "imov")))
   (set (attr "mode")
     (if_then_else (eq_attr "type" "imovx")
(const_string "SI")
(const_string "QI")))])

;; Stores and loads of ax to arbitrary constant address.
;; We fake an second form of instruction to force reload to load address
;; into register when rax is not available
(define_insn "*movabsqi_1_rex64"
  [(set (mem:QI (match_operand:DI 0 "x86_64_movabs_operand" "i,r"))
(match_operand:QI 1 "nonmemory_operand" "a,er"))]
  "TARGET_64BIT && ix86_check_movabs (insn, 0)"
  "@
   movabs{b}\t{%1, %P0|%P0, %1}
   mov{b}\t{%1, %a0|%a0, %1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0,*")
   (set_attr "memory" "store")
   (set_attr "mode" "QI")])

(define_insn "*movabsqi_2_rex64"
  [(set (match_operand:QI 0 "register_operand" "=a,r")
        (mem:QI (match_operand:DI 1 "x86_64_movabs_operand" "i,r")))]
  "TARGET_64BIT && ix86_check_movabs (insn, 1)"
  "@
   movabs{b}\t{%P1, %0|%0, %P1}
   mov{b}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "imov")
```

```
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0")
   (set_attr "memory" "load")
   (set_attr "mode" "QI")])

(define_insn "*movdi_extzv_1"
  [(set (match_operand:DI 0 "register_operand" "=R")
(zero_extract:DI (match_operand 1 "ext_register_operand" "Q")
 (const_int 8)
 (const_int 8)))]
  "TARGET_64BIT"
  "movz{bl|x}\t{%h1, %k0|%k0, %h1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "DI")])

(define_insn "*movsi_extzv_1"
  [(set (match_operand:SI 0 "register_operand" "=R")
(zero_extract:SI (match_operand 1 "ext_register_operand" "Q")
 (const_int 8)
 (const_int 8)))]
  ""
  "movz{bl|x}\t{%h1, %0|%0, %h1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")])

(define_insn "*movqi_extzv_2"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=Qm,?R")
        (subreg:QI (zero_extract:SI (match_operand 1 "ext_register_operand" "Q,Q")
    (const_int 8)
    (const_int 8)) 0))]
  "!TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      return "movz{bl|x}\t{%h1, %k0|%k0, %h1}";
    default:
      return "mov{b}\t{%h1, %0|%0, %h1}";
    }
}
  [(set (attr "type")
     (if_then_else (and (match_operand:QI 0 "register_operand" "")
(ior (not (match_operand:QI 0 "q_regs_operand" ""))
     (ne (symbol_ref "TARGET_MOVX")
 (const_int 0))))
(const_string "imovx")
(const_string "imov")))
   (set (attr "mode")
     (if_then_else (eq_attr "type" "imovx")
```

```
(const_string "SI")
(const_string "QI")))])

(define_insn "*movqi_extzv_2_rex64"
  [(set (match_operand:QI 0 "register_operand" "=Q,?R")
        (subreg:QI (zero_extract:SI (match_operand 1 "ext_register_operand" "Q,Q")
    (const_int 8)
    (const_int 8)) 0))]
  "TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      return "movz{bl|x}\t{%h1, %k0|%k0, %h1}";
    default:
      return "mov{b}\t{%h1, %0|%0, %h1}";
    }
}
  [(set (attr "type")
     (if_then_else (ior (not (match_operand:QI 0 "q_regs_operand" ""))
(ne (symbol_ref "TARGET_MOVX")
    (const_int 0)))
(const_string "imovx")
(const_string "imov")))
   (set (attr "mode")
     (if_then_else (eq_attr "type" "imovx")
(const_string "SI")
(const_string "QI")))])

(define_insn "movsi_insv_1"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "+Q")
 (const_int 8)
 (const_int 8))
(match_operand:SI 1 "general_operand" "Qmn"))]
  "!TARGET_64BIT"
  "mov{b}\t{%b1, %h0|%h0, %b1}"
  [(set_attr "type" "imov")
   (set_attr "mode" "QI")])

(define_insn "movdi_insv_1_rex64"
  [(set (zero_extract:DI (match_operand 0 "ext_register_operand" "+Q")
 (const_int 8)
 (const_int 8))
(match_operand:DI 1 "nonmemory_operand" "Qn"))]
  "TARGET_64BIT"
  "mov{b}\t{%b1, %h0|%h0, %b1}"
  [(set_attr "type" "imov")
   (set_attr "mode" "QI")])

(define_insn "*movqi_insv_2"
```

```
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "+Q")
 (const_int 8)
 (const_int 8))
(lshiftrt:SI (match_operand:SI 1 "register_operand" "Q")
     (const_int 8)))]
  ""
  "mov{b}\t{%h1, %h0|%h0, %h1}"
  [(set_attr "type" "imov")
   (set_attr "mode" "QI")])

(define_expand "movdi"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(match_operand:DI 1 "general_operand" ""))]
  ""
  "ix86_expand_move (DImode, operands); DONE;")

(define_insn "*pushdi"
  [(set (match_operand:DI 0 "push_operand" "=<")
(match_operand:DI 1 "general_no_elim_operand" "riF*m"))]
  "!TARGET_64BIT"
  "#")

(define_insn "*pushdi2_rex64"
  [(set (match_operand:DI 0 "push_operand" "=<,!<")
(match_operand:DI 1 "general_no_elim_operand" "re*m,n"))]
  "TARGET_64BIT"
  "@
   push{q}\t%1
   #"
  [(set_attr "type" "push,multi")
   (set_attr "mode" "DI")])

;; Convert impossible pushes of immediate to existing instructions.
;; First try to get scratch register and go through it.  In case this
;; fails, push sign extended lower part first and then overwrite
;; upper part by 32bit move.
(define_peephole2
  [(match_scratch:DI 2 "r")
   (set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

;; We need to define this as both peepholer and splitter for case
;; peephole2 pass is not run.
;; "&& 1" is needed to keep it from matching the previous pattern.
(define_peephole2
```

```
  [(set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode) && 1"
  [(set (match_dup 0) (match_dup 1))
   (set (match_dup 2) (match_dup 3))]
  "split_di (operands + 1, 1, operands + 2, operands + 3);
   operands[1] = gen_lowpart (DImode, operands[2]);
   operands[2] = gen_rtx_MEM (SImode, gen_rtx_PLUS (DImode, stack_pointer_rtx,
    GEN_INT (4)));
  ")

(define_split
  [(set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && (flag_peephole2 ? flow2_completed : reload_completed)
   && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 0) (match_dup 1))
   (set (match_dup 2) (match_dup 3))]
  "split_di (operands + 1, 1, operands + 2, operands + 3);
   operands[1] = gen_lowpart (DImode, operands[2]);
   operands[2] = gen_rtx_MEM (SImode, gen_rtx_PLUS (DImode, stack_pointer_rtx,
    GEN_INT (4)));
  ")

(define_insn "*pushdi2_prologue_rex64"
  [(set (match_operand:DI 0 "push_operand" "=<")
(match_operand:DI 1 "general_no_elim_operand" "re*m"))
   (clobber (mem:BLK (scratch)))]
  "TARGET_64BIT"
  "push{q}\t%1"
  [(set_attr "type" "push")
   (set_attr "mode" "DI")])

(define_insn "*popdi1_epilogue_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r*m")
(mem:DI (reg:DI SP_REG)))
   (set (reg:DI SP_REG)
(plus:DI (reg:DI SP_REG) (const_int 8)))
   (clobber (mem:BLK (scratch)))]
  "TARGET_64BIT"
  "pop{q}\t%0"
  [(set_attr "type" "pop")
   (set_attr "mode" "DI")])

(define_insn "popdi1"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r*m")
(mem:DI (reg:DI SP_REG)))
   (set (reg:DI SP_REG)
```

```
(plus:DI (reg:DI SP_REG) (const_int 8)))]
  "TARGET_64BIT"
  "pop{q}\t%0"
  [(set_attr "type" "pop")
   (set_attr "mode" "DI")])

(define_insn "*movdi_xor_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(match_operand:DI 1 "const0_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (!TARGET_USE_MOV0 || optimize_size)
   && reload_completed"
  "xor{l}\t{%k0, %k0|%k0, %k0}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")
   (set_attr "length_immediate" "0")])

(define_insn "*movdi_or_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(match_operand:DI 1 "const_int_operand" "i"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (TARGET_PENTIUM || optimize_size)
   && reload_completed
   && operands[1] == constm1_rtx"
{
  operands[1] = constm1_rtx;
  return "or{q}\t{%1, %0|%0, %1}";
}
  [(set_attr "type" "alu1")
   (set_attr "mode" "DI")
   (set_attr "length_immediate" "1")])

(define_insn "*movdi_2"
  [(set (match_operand:DI 0 "nonimmediate_operand"
"=r  ,o  ,*y,m*y,*y,*Y,m ,*Y,*Y,*x,m ,*x,*x")
(match_operand:DI 1 "general_operand"
"riFo,riF,C ,*y ,m ,C ,*Y,*Y,m ,C ,*x,*x,m "))]
  "!TARGET_64BIT && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
   #
   #
   pxor\t%0, %0
   movq\t{%1, %0|%0, %1}
   movq\t{%1, %0|%0, %1}
   pxor\t%0, %0
   movq\t{%1, %0|%0, %1}
   movdqa\t{%1, %0|%0, %1}
   movq\t{%1, %0|%0, %1}
   xorps\t%0, %0
   movlps\t{%1, %0|%0, %1}
```

```
   movaps\t{%1, %0|%0, %1}
   movlps\t{%1, %0|%0, %1}"
  [(set_attr "type" "*,*,mmx,mmxmov,mmxmov,sselog1,ssemov,ssemov,ssemov,sselog1,ssemov,ssemov,ssemov")
   (set_attr "mode" "DI,DI,DI,DI,DI,TI,DI,TI,DI,V4SF,V2SF,V4SF,V2SF")])

(define_split
  [(set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "general_operand" ""))]
  "!TARGET_64BIT && reload_completed
   && (! MMX_REG_P (operands[1]) && !SSE_REG_P (operands[1]))"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

;; %%% This multiword shite has got to go.
(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
        (match_operand:DI 1 "general_operand" ""))]
  "!TARGET_64BIT && reload_completed
   && (!MMX_REG_P (operands[0]) && !SSE_REG_P (operands[0]))
   && (!MMX_REG_P (operands[1]) && !SSE_REG_P (operands[1]))"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

(define_insn "*movdi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand"
"=r,r  ,r,m ,!m,*y,*y,?rm,?*y,*x,*x,?rm,?*x,?*x,?*y")
(match_operand:DI 1 "general_operand"
"Z ,rem,i,re,n ,C ,*y,*y ,rm ,C ,*x,*x ,rm ,*y ,*x"))]
  "TARGET_64BIT && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
{
  switch (get_attr_type (insn))
    {
    case TYPE_SSECVT:
      if (which_alternative == 13)
return "movq2dq\t{%1, %0|%0, %1}";
      else
return "movdq2q\t{%1, %0|%0, %1}";
    case TYPE_SSEMOV:
      if (get_attr_mode (insn) == MODE_TI)
  return "movdqa\t{%1, %0|%0, %1}";
      /* FALLTHRU */
    case TYPE_MMXMOV:
      /* Moves from and into integer register is done using movd opcode with
   REX prefix.  */
      if (GENERAL_REG_P (operands[0]) || GENERAL_REG_P (operands[1]))
  return "movd\t{%1, %0|%0, %1}";
      return "movq\t{%1, %0|%0, %1}";
    case TYPE_SSELOG1:
    case TYPE_MMXADD:
      return "pxor\t%0, %0";
```

```
    case TYPE_MULTI:
      return "#";
    case TYPE_LEA:
      return "lea{q}\t{%a1, %0|%0, %a1}";
    default:
      gcc_assert (!flag_pic || LEGITIMATE_PIC_OPERAND_P (operands[1]));
      if (get_attr_mode (insn) == MODE_SI)
return "mov{l}\t{%k1, %k0|%k0, %k1}";
      else if (which_alternative == 2)
return "movabs{q}\t{%1, %0|%0, %1}";
      else
return "mov{q}\t{%1, %0|%0, %1}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "5")
      (const_string "mmx")
    (eq_attr "alternative" "6,7,8")
      (const_string "mmxmov")
    (eq_attr "alternative" "9")
      (const_string "sselog1")
    (eq_attr "alternative" "10,11,12")
      (const_string "ssemov")
    (eq_attr "alternative" "13,14")
      (const_string "ssecvt")
    (eq_attr "alternative" "4")
      (const_string "multi")
      (match_operand:DI 1 "pic_32bit_operand" "")
      (const_string "lea")
   ]
   (const_string "imov")))
   (set_attr "modrm" "*,0,0,*,*,*,*,*,*,*,*,*,*,*,*")
   (set_attr "length_immediate" "*,4,8,*,*,*,*,*,*,*,*,*,*,*,*")
   (set_attr "mode" "SI,DI,DI,DI,SI,DI,DI,DI,DI,TI,TI,DI,DI,DI,DI")])

;; Stores and loads of ax to arbitrary constant address.
;; We fake an second form of instruction to force reload to load address
;; into register when rax is not available
(define_insn "*movabsdi_1_rex64"
  [(set (mem:DI (match_operand:DI 0 "x86_64_movabs_operand" "i,r"))
(match_operand:DI 1 "nonmemory_operand" "a,er"))]
  "TARGET_64BIT && ix86_check_movabs (insn, 0)"
  "@
   movabs{q}\t{%1, %P0|%P0, %1}
   mov{q}\t{%1, %a0|%a0, %1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0,*")
   (set_attr "memory" "store")
```

```
   (set_attr "mode" "DI")])

(define_insn "*movabsdi_2_rex64"
  [(set (match_operand:DI 0 "register_operand" "=a,r")
        (mem:DI (match_operand:DI 1 "x86_64_movabs_operand" "i,r")))]
  "TARGET_64BIT && ix86_check_movabs (insn, 1)"
  "@
   movabs{q}\t{%P1, %0|%0, %P1}
   mov{q}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0,*")
   (set_attr "length_address" "8,0")
   (set_attr "length_immediate" "0")
   (set_attr "memory" "load")
   (set_attr "mode" "DI")])

;; Convert impossible stores of immediate to existing instructions.
;; First try to get scratch register and go through it.  In case this
;; fails, move by 32bit parts.
(define_peephole2
  [(match_scratch:DI 2 "r")
   (set (match_operand:DI 0 "memory_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

;; We need to define this as both peepholer and splitter for case
;; peephole2 pass is not run.
;; "&& 1" is needed to keep it from matching the previous pattern.
(define_peephole2
  [(set (match_operand:DI 0 "memory_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode) && 1"
  [(set (match_dup 2) (match_dup 3))
   (set (match_dup 4) (match_dup 5))]
  "split_di (operands, 2, operands + 2, operands + 4);")

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && (flag_peephole2 ? flow2_completed : reload_completed)
   && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 2) (match_dup 3))
   (set (match_dup 4) (match_dup 5))]
  "split_di (operands, 2, operands + 2, operands + 4);")
```

```
(define_insn "*swapdi_rex64"
  [(set (match_operand:DI 0 "register_operand" "+r")
(match_operand:DI 1 "register_operand" "+r"))
   (set (match_dup 1)
(match_dup 0))]
  "TARGET_64BIT"
  "xchg{q}\t%1, %0"
  [(set_attr "type" "imov")
   (set_attr "mode" "DI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_expand "movti"
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
(match_operand:TI 1 "nonimmediate_operand" ""))]
  "TARGET_SSE || TARGET_64BIT"
{
  if (TARGET_64BIT)
    ix86_expand_move (TImode, operands);
  else
    ix86_expand_vector_move (TImode, operands);
  DONE;
})

(define_insn "*movti_internal"
  [(set (match_operand:TI 0 "nonimmediate_operand" "=x,x,m")
(match_operand:TI 1 "vector_move_operand" "C,xm,x"))]
  "TARGET_SSE && !TARGET_64BIT
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
      if (get_attr_mode (insn) == MODE_V4SF)
return "xorps\t%0, %0";
      else
return "pxor\t%0, %0";
    case 1:
    case 2:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
return "movdqa\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "ssemov,ssemov,ssemov")
   (set (attr "mode")
```

```
       (cond [(eq (symbol_ref "TARGET_SSE2") (const_int 0))
 (const_string "V4SF")

      (eq_attr "alternative" "0,1")
 (if_then_else
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
   (const_string "TI"))
       (eq_attr "alternative" "2")
 (if_then_else
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
   (const_string "TI"))]
       (const_string "TI")))])

(define_insn "*movti_rex64"
  [(set (match_operand:TI 0 "nonimmediate_operand" "=r,o,x,x,xm")
(match_operand:TI 1 "general_operand" "riFo,riF,C,xm,x"))]
  "TARGET_64BIT
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
    case 1:
      return "#";
    case 2:
      if (get_attr_mode (insn) == MODE_V4SF)
return "xorps\t%0, %0";
      else
return "pxor\t%0, %0";
    case 3:
    case 4:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
return "movdqa\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "*,*,ssemov,ssemov,ssemov")
   (set (attr "mode")
        (cond [(eq_attr "alternative" "2,3")
 (if_then_else
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
```

```
   (const_string "TI"))
       (eq_attr "alternative" "4")
 (if_then_else
   (ior (ne (symbol_ref "TARGET_SSE_TYPELESS_STORES")
    (const_int 0))
(ne (symbol_ref "optimize_size")
    (const_int 0)))
   (const_string "V4SF")
   (const_string "TI"))]
       (const_string "DI")))])

(define_split
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
        (match_operand:TI 1 "general_operand" ""))]
  "reload_completed && !SSE_REG_P (operands[0])
   && !SSE_REG_P (operands[1])"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

(define_expand "movsf"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
(match_operand:SF 1 "general_operand" ""))]
  ""
  "ix86_expand_move (SFmode, operands); DONE;")

(define_insn "*pushsf"
  [(set (match_operand:SF 0 "push_operand" "=<,<,<")
(match_operand:SF 1 "general_no_elim_operand" "f#rx,rFm#fx,x#rf"))]
  "!TARGET_64BIT"
{
  /* Anything else should be already split before reg-stack.  */
  gcc_assert (which_alternative == 1);
  return "push{l}\t%1";
}
  [(set_attr "type" "multi,push,multi")
   (set_attr "unit" "i387,*,*")
   (set_attr "mode" "SF,SI,SF")])

(define_insn "*pushsf_rex64"
  [(set (match_operand:SF 0 "push_operand" "=X,X,X")
(match_operand:SF 1 "nonmemory_no_elim_operand" "f#rx,rF#fx,x#rf"))]
  "TARGET_64BIT"
{
  /* Anything else should be already split before reg-stack.  */
  gcc_assert (which_alternative == 1);
  return "push{q}\t%q1";
}
  [(set_attr "type" "multi,push,multi")
   (set_attr "unit" "i387,*,*")
   (set_attr "mode" "SF,DI,SF")])
```

```
(define_split
  [(set (match_operand:SF 0 "push_operand" "")
(match_operand:SF 1 "memory_operand" ""))]
  "reload_completed
   && GET_CODE (operands[1]) == MEM
   && GET_CODE (XEXP (operands[1], 0)) == SYMBOL_REF
   && CONSTANT_POOL_ADDRESS_P (XEXP (operands[1], 0))"
  [(set (match_dup 0)
(match_dup 1))]
  "operands[1] = get_pool_constant (XEXP (operands[1], 0));")


;; %%% Kill this when call knows how to work this out.
(define_split
  [(set (match_operand:SF 0 "push_operand" "")
(match_operand:SF 1 "any_fp_register_operand" ""))]
  "!TARGET_64BIT"
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -4)))
   (set (mem:SF (reg:SI SP_REG)) (match_dup 1))])

(define_split
  [(set (match_operand:SF 0 "push_operand" "")
(match_operand:SF 1 "any_fp_register_operand" ""))]
  "TARGET_64BIT"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -8)))
   (set (mem:SF (reg:DI SP_REG)) (match_dup 1))])

(define_insn "*movsf_1"
  [(set (match_operand:SF 0 "nonimmediate_operand"
  "=f#xr,m   ,f#xr,r#xf  ,m    ,x#rf,x#rf,x#rf ,m   ,!*y,!rm,!*y")
(match_operand:SF 1 "general_operand"
  "fm#rx,f#rx,G   ,rmF#fx,Fr#fx,C   ,x   ,xm#rf,x#rf,rm ,*y ,*y"))]
  "!(MEM_P (operands[0]) && MEM_P (operands[1]))
   && (reload_in_progress || reload_completed
       || (ix86_cmodel == CM_MEDIUM || ix86_cmodel == CM_LARGE)
       || GET_CODE (operands[1]) != CONST_DOUBLE
       || memory_operand (operands[0], SFmode))"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fst%z0\t%y0";
```

```
    case 2:
      return standard_80387_constant_opcode (operands[1]);

    case 3:
    case 4:
      return "mov{l}\t{%1, %0|%0, %1}";
    case 5:
      if (get_attr_mode (insn) == MODE_TI)
return "pxor\t%0, %0";
      else
return "xorps\t%0, %0";
    case 6:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
return "movss\t{%1, %0|%0, %1}";
    case 7:
    case 8:
      return "movss\t{%1, %0|%0, %1}";

    case 9:
    case 10:
      return "movd\t{%1, %0|%0, %1}";

    case 11:
      return "movq\t{%1, %0|%0, %1}";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,fmov,imov,imov,ssemov,ssemov,ssemov,ssemov,mmxmov,mmxmov,mmxmov")
   (set (attr "mode")
        (cond [(eq_attr "alternative" "3,4,9,10")
 (const_string "SI")
       (eq_attr "alternative" "5")
 (if_then_else
   (and (and (ne (symbol_ref "TARGET_SSE_LOAD0_BY_PXOR")
      (const_int 0))
     (ne (symbol_ref "TARGET_SSE2")
 (const_int 0)))
(eq (symbol_ref "optimize_size")
    (const_int 0)))
   (const_string "TI")
   (const_string "V4SF"))
       /* For architectures resolving dependencies on
  whole SSE registers use APS move to break dependency
  chains, otherwise use short move to avoid extra work.

  Do the same for architectures resolving dependencies on
```

```
  the parts.  While in DF mode it is better to always handle
  just register parts, the SF mode is different due to lack
  of instructions to load just part of the register.  It is
  better to maintain the whole registers in single format
  to avoid problems on using packed logical operations.  */
       (eq_attr "alternative" "6")
 (if_then_else
   (ior (ne (symbol_ref "TARGET_SSE_PARTIAL_REG_DEPENDENCY")
    (const_int 0))
(ne (symbol_ref "TARGET_SSE_SPLIT_REGS")
    (const_int 0)))
   (const_string "V4SF")
   (const_string "SF"))
       (eq_attr "alternative" "11")
 (const_string "DI")]
       (const_string "SF")))])

(define_insn "*swapsf"
  [(set (match_operand:SF 0 "fp_register_operand" "+f")
(match_operand:SF 1 "fp_register_operand" "+f"))
   (set (match_dup 1)
(match_dup 0))]
  "reload_completed || TARGET_80387"
{
  if (STACK_TOP_P (operands[0]))
    return "fxch\t%1";
  else
    return "fxch\t%0";
}
  [(set_attr "type" "fxch")
   (set_attr "mode" "SF")])

(define_expand "movdf"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
(match_operand:DF 1 "general_operand" ""))]
  ""
  "ix86_expand_move (DFmode, operands); DONE;")

;; Size of pushdf is 3 (for sub) + 2 (for fstp) + memory operand size.
;; Size of pushdf using integer instructions is 2+2*memory operand size
;; On the average, pushdf using integers can be still shorter.  Allow this
;; pattern for optimize_size too.

(define_insn "*pushdf_nointeger"
  [(set (match_operand:DF 0 "push_operand" "=<,<,<,<")
(match_operand:DF 1 "general_no_elim_operand" "f#Y,Fo#fY,*r#fY,Y#f"))]
  "!TARGET_64BIT && !TARGET_INTEGER_DFMODE_MOVES"
{
  /* This insn should be already split before reg-stack.  */
  gcc_unreachable ();
```

```
}
  [(set_attr "type" "multi")
   (set_attr "unit" "i387,*,*,*")
   (set_attr "mode" "DF,SI,SI,DF")])

(define_insn "*pushdf_integer"
  [(set (match_operand:DF 0 "push_operand" "=<,<,<")
(match_operand:DF 1 "general_no_elim_operand" "f#rY,rFo#fY,Y#rf"))]
  "TARGET_64BIT || TARGET_INTEGER_DFMODE_MOVES"
{
  /* This insn should be already split before reg-stack.  */
  gcc_unreachable ();
}
  [(set_attr "type" "multi")
   (set_attr "unit" "i387,*,*")
   (set_attr "mode" "DF,SI,DF")])

;; %%% Kill this when call knows how to work this out.
(define_split
  [(set (match_operand:DF 0 "push_operand" "")
(match_operand:DF 1 "any_fp_register_operand" ""))]
  "!TARGET_64BIT && reload_completed"
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -8)))
   (set (mem:DF (reg:SI SP_REG)) (match_dup 1))]
  "")

(define_split
  [(set (match_operand:DF 0 "push_operand" "")
(match_operand:DF 1 "any_fp_register_operand" ""))]
  "TARGET_64BIT && reload_completed"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -8)))
   (set (mem:DF (reg:DI SP_REG)) (match_dup 1))]
  "")

(define_split
  [(set (match_operand:DF 0 "push_operand" "")
(match_operand:DF 1 "general_operand" ""))]
  "reload_completed"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

;; Moving is usually shorter when only FP registers are used. This separate
;; movdf pattern avoids the use of integer registers for FP operations
;; when optimizing for size.

(define_insn "*movdf_nointeger"
  [(set (match_operand:DF 0 "nonimmediate_operand"
"=f#Y,m  ,f#Y,*r  ,o  ,Y*x#f,Y*x#f,Y*x#f  ,m    ")
(match_operand:DF 1 "general_operand"
"fm#Y,f#Y,G  ,*roF,F*r,C    ,Y*x#f,HmY*x#f,Y*x#f"))]
```

```
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && ((optimize_size || !TARGET_INTEGER_DFMODE_MOVES) && !TARGET_64BIT)
   && (reload_in_progress || reload_completed
       || (ix86_cmodel == CM_MEDIUM || ix86_cmodel == CM_LARGE)
       || GET_CODE (operands[1]) != CONST_DOUBLE
       || memory_operand (operands[0], DFmode))"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fst%z0\t%y0";

    case 2:
      return standard_80387_constant_opcode (operands[1]);

    case 3:
    case 4:
      return "#";
    case 5:
      switch (get_attr_mode (insn))
{
case MODE_V4SF:
  return "xorps\t%0, %0";
case MODE_V2DF:
  return "xorpd\t%0, %0";
case MODE_TI:
  return "pxor\t%0, %0";
default:
  gcc_unreachable ();
}
    case 6:
    case 7:
    case 8:
      switch (get_attr_mode (insn))
{
case MODE_V4SF:
  return "movaps\t{%1, %0|%0, %1}";
case MODE_V2DF:
  return "movapd\t{%1, %0|%0, %1}";
case MODE_TI:
  return "movdqa\t{%1, %0|%0, %1}";
case MODE_DI:
  return "movq\t{%1, %0|%0, %1}";
case MODE_DF:
```

```
  return "movsd\t{%1, %0|%0, %1}";
case MODE_V1DF:
  return "movlpd\t{%1, %0|%0, %1}";
case MODE_V2SF:
  return "movlps\t{%1, %0|%0, %1}";
default:
  gcc_unreachable ();
}

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,fmov,multi,multi,ssemov,ssemov,ssemov,ssemov")
   (set (attr "mode")
        (cond [(eq_attr "alternative" "0,1,2")
 (const_string "DF")
       (eq_attr "alternative" "3,4")
 (const_string "SI")

       /* For SSE1, we have many fewer alternatives.  */
       (eq (symbol_ref "TARGET_SSE2") (const_int 0))
 (cond [(eq_attr "alternative" "5,6")
  (const_string "V4SF")
       ]
   (const_string "V2SF"))

       /* xorps is one byte shorter.  */
       (eq_attr "alternative" "5")
 (cond [(ne (symbol_ref "optimize_size")
    (const_int 0))
  (const_string "V4SF")
(ne (symbol_ref "TARGET_SSE_LOAD0_BY_PXOR")
    (const_int 0))
  (const_string "TI")
       ]
       (const_string "V2DF"))

       /* For architectures resolving dependencies on
  whole SSE registers use APD move to break dependency
  chains, otherwise use short move to avoid extra work.

  movaps encodes one byte shorter.  */
       (eq_attr "alternative" "6")
 (cond
   [(ne (symbol_ref "optimize_size")
        (const_int 0))
      (const_string "V4SF")
    (ne (symbol_ref "TARGET_SSE_PARTIAL_REG_DEPENDENCY")
        (const_int 0))
```

```
      (const_string "V2DF")
   ]
   (const_string "DF"))
       /* For architectures resolving dependencies on register
  parts we may avoid extra work to zero out upper part
  of register.  */
       (eq_attr "alternative" "7")
 (if_then_else
   (ne (symbol_ref "TARGET_SSE_SPLIT_REGS")
       (const_int 0))
   (const_string "V1DF")
   (const_string "DF"))
      ]
      (const_string "DF")))])

(define_insn "*movdf_integer"
  [(set (match_operand:DF 0 "nonimmediate_operand"
"=f#Yr,m   ,f#Yr,r#Yf  ,o    ,Y*x#rf,Y*x#rf,Y*x#rf,m")
(match_operand:DF 1 "general_operand"
"fm#Yr,f#Yr,G   ,roF#Yf,Fr#Yf,C     ,Y*x#rf,m     ,Y*x#rf"))]
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && ((!optimize_size && TARGET_INTEGER_DFMODE_MOVES) || TARGET_64BIT)
   && (reload_in_progress || reload_completed
       || (ix86_cmodel == CM_MEDIUM || ix86_cmodel == CM_LARGE)
       || GET_CODE (operands[1]) != CONST_DOUBLE
       || memory_operand (operands[0], DFmode))"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fst%z0\t%y0";

    case 2:
      return standard_80387_constant_opcode (operands[1]);

    case 3:
    case 4:
      return "#";

    case 5:
      switch (get_attr_mode (insn))
{
case MODE_V4SF:
  return "xorps\t%0, %0";
```

```
case MODE_V2DF:
  return "xorpd\t%0, %0";
case MODE_TI:
  return "pxor\t%0, %0";
default:
  gcc_unreachable ();
}
    case 6:
    case 7:
    case 8:
      switch (get_attr_mode (insn))
{
case MODE_V4SF:
  return "movaps\t{%1, %0|%0, %1}";
case MODE_V2DF:
  return "movapd\t{%1, %0|%0, %1}";
case MODE_TI:
  return "movdqa\t{%1, %0|%0, %1}";
case MODE_DI:
  return "movq\t{%1, %0|%0, %1}";
case MODE_DF:
  return "movsd\t{%1, %0|%0, %1}";
case MODE_V1DF:
  return "movlpd\t{%1, %0|%0, %1}";
case MODE_V2SF:
  return "movlps\t{%1, %0|%0, %1}";
default:
  gcc_unreachable ();
}

    default:
      gcc_unreachable();
    }
}
  [(set_attr "type" "fmov,fmov,fmov,multi,multi,ssemov,ssemov,ssemov,ssemov")
   (set (attr "mode")
        (cond [(eq_attr "alternative" "0,1,2")
 (const_string "DF")
       (eq_attr "alternative" "3,4")
 (const_string "SI")

       /* For SSE1, we have many fewer alternatives.  */
       (eq (symbol_ref "TARGET_SSE2") (const_int 0))
 (cond [(eq_attr "alternative" "5,6")
  (const_string "V4SF")
       ]
   (const_string "V2SF"))

       /* xorps is one byte shorter.  */
       (eq_attr "alternative" "5")
```

```
 (cond [(ne (symbol_ref "optimize_size")
    (const_int 0))
  (const_string "V4SF")
(ne (symbol_ref "TARGET_SSE_LOAD0_BY_PXOR")
    (const_int 0))
  (const_string "TI")
       ]
       (const_string "V2DF"))

       /* For architectures resolving dependencies on
  whole SSE registers use APD move to break dependency
  chains, otherwise use short move to avoid extra work.

  movaps encodes one byte shorter.  */
       (eq_attr "alternative" "6")
 (cond
   [(ne (symbol_ref "optimize_size")
        (const_int 0))
      (const_string "V4SF")
    (ne (symbol_ref "TARGET_SSE_PARTIAL_REG_DEPENDENCY")
        (const_int 0))
      (const_string "V2DF")
   ]
   (const_string "DF"))
       /* For architectures resolving dependencies on register
  parts we may avoid extra work to zero out upper part
  of register.  */
       (eq_attr "alternative" "7")
 (if_then_else
   (ne (symbol_ref "TARGET_SSE_SPLIT_REGS")
       (const_int 0))
   (const_string "V1DF")
   (const_string "DF"))
      ]
      (const_string "DF")))])

(define_split
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
(match_operand:DF 1 "general_operand" ""))]
  "reload_completed
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && ! (ANY_FP_REG_P (operands[0]) ||
 (GET_CODE (operands[0]) == SUBREG
  && ANY_FP_REG_P (SUBREG_REG (operands[0]))))
   && ! (ANY_FP_REG_P (operands[1]) ||
 (GET_CODE (operands[1]) == SUBREG
  && ANY_FP_REG_P (SUBREG_REG (operands[1])))))"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")
```

```
(define_insn "*swapdf"
  [(set (match_operand:DF 0 "fp_register_operand" "+f")
(match_operand:DF 1 "fp_register_operand" "+f"))
   (set (match_dup 1)
(match_dup 0))]
  "reload_completed || TARGET_80387"
{
  if (STACK_TOP_P (operands[0]))
    return "fxch\t%1";
  else
    return "fxch\t%0";
}
  [(set_attr "type" "fxch")
   (set_attr "mode" "DF")])

(define_expand "movxf"
  [(set (match_operand:XF 0 "nonimmediate_operand" "")
(match_operand:XF 1 "general_operand" ""))]
  ""
  "ix86_expand_move (XFmode, operands); DONE;")

;; Size of pushdf is 3 (for sub) + 2 (for fstp) + memory operand size.
;; Size of pushdf using integer instructions is 3+3*memory operand size
;; Pushing using integer instructions is longer except for constants
;; and direct memory references.
;; (assuming that any given constant is pushed only once, but this ought to be
;;  handled elsewhere).

(define_insn "*pushxf_nointeger"
  [(set (match_operand:XF 0 "push_operand" "=X,X,X")
(match_operand:XF 1 "general_no_elim_operand" "f,Fo,*r"))]
  "optimize_size"
{
  /* This insn should be already split before reg-stack.  */
  gcc_unreachable ();
}
  [(set_attr "type" "multi")
   (set_attr "unit" "i387,*,*")
   (set_attr "mode" "XF,SI,SI")])

(define_insn "*pushxf_integer"
  [(set (match_operand:XF 0 "push_operand" "=<,<")
(match_operand:XF 1 "general_no_elim_operand" "f#r,ro#f"))]
  "!optimize_size"
{
  /* This insn should be already split before reg-stack.  */
  gcc_unreachable ();
}
  [(set_attr "type" "multi")
   (set_attr "unit" "i387,*")
```

```
   (set_attr "mode" "XF,SI")])

(define_split
  [(set (match_operand 0 "push_operand" "")
(match_operand 1 "general_operand" ""))]
  "reload_completed
   && (GET_MODE (operands[0]) == XFmode
       || GET_MODE (operands[0]) == DFmode)
   && !ANY_FP_REG_P (operands[1])"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(match_operand:XF 1 "any_fp_register_operand" ""))]
  "!TARGET_64BIT"
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (match_dup 2)))
   (set (mem:XF (reg:SI SP_REG)) (match_dup 1))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(match_operand:XF 1 "any_fp_register_operand" ""))]
  "TARGET_64BIT"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (match_dup 2)))
   (set (mem:XF (reg:DI SP_REG)) (match_dup 1))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

;; Do not use integer registers when optimizing for size
(define_insn "*movxf_nointeger"
  [(set (match_operand:XF 0 "nonimmediate_operand" "=f,m,f,*r,o")
(match_operand:XF 1 "general_operand" "fm,f,G,*roF,F*r"))]
  "optimize_size
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && (reload_in_progress || reload_completed
       || GET_CODE (operands[1]) != CONST_DOUBLE
       || memory_operand (operands[0], XFmode))"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      /* There is no non-popping store to memory for XFmode.  So if
 we need one, follow the store with a load.  */
      if (! find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0\;fld%z0\t%y0";
      else
        return "fstp%z0\t%y0";
```

```
    case 2:
      return standard_80387_constant_opcode (operands[1]);

    case 3: case 4:
      return "#";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,fmov,multi,multi")
   (set_attr "mode" "XF,XF,XF,SI,SI")])

(define_insn "*movxf_integer"
  [(set (match_operand:XF 0 "nonimmediate_operand" "=f#r,m,f#r,r#f,o")
(match_operand:XF 1 "general_operand" "fm#r,f#r,G,roF#f,Fr#f"))]
  "!optimize_size
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && (reload_in_progress || reload_completed
       || GET_CODE (operands[1]) != CONST_DOUBLE
       || memory_operand (operands[0], XFmode))"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      /* There is no non-popping store to memory for XFmode.  So if
 we need one, follow the store with a load.  */
      if (! find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0\;fld%z0\t%y0";
      else
        return "fstp%z0\t%y0";

    case 2:
      return standard_80387_constant_opcode (operands[1]);

    case 3: case 4:
      return "#";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,fmov,multi,multi")
   (set_attr "mode" "XF,XF,XF,SI,SI")])

(define_split
  [(set (match_operand 0 "nonimmediate_operand" "")
```

```
(match_operand 1 "general_operand" ""))]
  "reload_completed
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && GET_MODE (operands[0]) == XFmode
   && ! (ANY_FP_REG_P (operands[0]) ||
 (GET_CODE (operands[0]) == SUBREG
  && ANY_FP_REG_P (SUBREG_REG (operands[0]))))
   && ! (ANY_FP_REG_P (operands[1]) ||
 (GET_CODE (operands[1]) == SUBREG
  && ANY_FP_REG_P (SUBREG_REG (operands[1])))))"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

(define_split
  [(set (match_operand 0 "register_operand" "")
(match_operand 1 "memory_operand" ""))]
  "reload_completed
   && GET_CODE (operands[1]) == MEM
   && (GET_MODE (operands[0]) == XFmode
       || GET_MODE (operands[0]) == SFmode || GET_MODE (operands[0]) == DFmode)
   && GET_CODE (XEXP (operands[1], 0)) == SYMBOL_REF
   && CONSTANT_POOL_ADDRESS_P (XEXP (operands[1], 0))"
  [(set (match_dup 0) (match_dup 1))]
{
  rtx c = get_pool_constant (XEXP (operands[1], 0));
  rtx r = operands[0];

  if (GET_CODE (r) == SUBREG)
    r = SUBREG_REG (r);

  if (SSE_REG_P (r))
    {
      if (!standard_sse_constant_p (c))
FAIL;
    }
  else if (FP_REG_P (r))
    {
      if (!standard_80387_constant_p (c))
FAIL;
    }
  else if (MMX_REG_P (r))
    FAIL;

  operands[1] = c;
})

(define_insn "swapxf"
  [(set (match_operand:XF 0 "register_operand" "+f")
(match_operand:XF 1 "register_operand" "+f"))
   (set (match_dup 1)
```

```
(match_dup 0))]
  "TARGET_80387"
{
  if (STACK_TOP_P (operands[0]))
    return "fxch\t%1";
  else
    return "fxch\t%0";
}
  [(set_attr "type" "fxch")
   (set_attr "mode" "XF")])

(define_expand "movtf"
  [(set (match_operand:TF 0 "nonimmediate_operand" "")
(match_operand:TF 1 "nonimmediate_operand" ""))]
  "TARGET_64BIT"
{
  ix86_expand_move (TFmode, operands);
  DONE;
})

(define_insn "*movtf_internal"
  [(set (match_operand:TF 0 "nonimmediate_operand" "=r,o,x,x,xm")
(match_operand:TF 1 "general_operand" "riFo,riF,C,xm,x"))]
  "TARGET_64BIT
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
    case 1:
      return "#";
    case 2:
      if (get_attr_mode (insn) == MODE_V4SF)
return "xorps\t%0, %0";
      else
return "pxor\t%0, %0";
    case 3:
    case 4:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
return "movdqa\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "*,*,ssemov,ssemov,ssemov")
   (set (attr "mode")
        (cond [(eq_attr "alternative" "2,3")
 (if_then_else
```

```
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
   (const_string "TI"))
       (eq_attr "alternative" "4")
 (if_then_else
   (ior (ne (symbol_ref "TARGET_SSE_TYPELESS_STORES")
    (const_int 0))
(ne (symbol_ref "optimize_size")
    (const_int 0)))
   (const_string "V4SF")
   (const_string "TI"))]
       (const_string "DI")))])

(define_split
  [(set (match_operand:TF 0 "nonimmediate_operand" "")
        (match_operand:TF 1 "general_operand" ""))]
  "reload_completed && !SSE_REG_P (operands[0])
   && !SSE_REG_P (operands[1])"
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

;; Zero extension instructions

(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "")
     (zero_extend:SI (match_operand:HI 1 "nonimmediate_operand" "")))]
  ""
{
  if (TARGET_ZERO_EXTEND_WITH_AND && !optimize_size)
    {
      operands[1] = force_reg (HImode, operands[1]);
      emit_insn (gen_zero_extendhisi2_and (operands[0], operands[1]));
      DONE;
    }
})

(define_insn "zero_extendhisi2_and"
  [(set (match_operand:SI 0 "register_operand" "=r")
     (zero_extend:SI (match_operand:HI 1 "register_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(zero_extend:SI (match_operand:HI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "reload_completed && TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
  [(parallel [(set (match_dup 0) (and:SI (match_dup 0) (const_int 65535)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_insn "*zero_extendhisi2_movzwl"
  [(set (match_operand:SI 0 "register_operand" "=r")
     (zero_extend:SI (match_operand:HI 1 "nonimmediate_operand" "rm")))]
  "!TARGET_ZERO_EXTEND_WITH_AND || optimize_size"
  "movz{wl|x}\t{%1, %0|%0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")])

(define_expand "zero_extendqihi2"
  [(parallel
    [(set (match_operand:HI 0 "register_operand" "")
       (zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*zero_extendqihi2_and"
  [(set (match_operand:HI 0 "register_operand" "=r,?&q")
     (zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "0,qm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "HI")])

(define_insn "*zero_extendqihi2_movzbw_and"
  [(set (match_operand:HI 0 "register_operand" "=r,r")
     (zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "qm,0")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_ZERO_EXTEND_WITH_AND || optimize_size"
  "#"
  [(set_attr "type" "imovx,alu1")
   (set_attr "mode" "HI")])

(define_insn "*zero_extendqihi2_movzbw"
  [(set (match_operand:HI 0 "register_operand" "=r")
     (zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "qm")))]
  "(!TARGET_ZERO_EXTEND_WITH_AND || optimize_size) && reload_completed"
  "movz{bw|x}\t{%1, %0|%0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "HI")])

;; For the movzbw case strip only the clobber
(define_split
  [(set (match_operand:HI 0 "register_operand" "")
```

```
(zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && (!TARGET_ZERO_EXTEND_WITH_AND || optimize_size)
   && (!REG_P (operands[1]) || ANY_QI_REG_P (operands[1]))"
  [(set (match_operand:HI 0 "register_operand" "")
(zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "")))])

;; When source and destination does not overlap, clear destination
;; first and then do the movb
(define_split
  [(set (match_operand:HI 0 "register_operand" "")
(zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && ANY_QI_REG_P (operands[0])
   && (TARGET_ZERO_EXTEND_WITH_AND && !optimize_size)
   && !reg_overlap_mentioned_p (operands[0], operands[1])"
  [(set (match_dup 0) (const_int 0))
   (set (strict_low_part (match_dup 2)) (match_dup 1))]
  "operands[2] = gen_lowpart (QImode, operands[0]);")

;; Rest is handled by single and.
(define_split
  [(set (match_operand:HI 0 "register_operand" "")
(zero_extend:HI (match_operand:QI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && true_regnum (operands[0]) == true_regnum (operands[1])"
  [(parallel [(set (match_dup 0) (and:HI (match_dup 0) (const_int 255)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_expand "zero_extendqisi2"
  [(parallel
    [(set (match_operand:SI 0 "register_operand" "")
       (zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*zero_extendqisi2_and"
  [(set (match_operand:SI 0 "register_operand" "=r,?&q")
     (zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "0,qm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")])
```

```
(define_insn "*zero_extendqisi2_movzbw_and"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
     (zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "qm,0")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_ZERO_EXTEND_WITH_AND || optimize_size"
  "#"
  [(set_attr "type" "imovx,alu1")
   (set_attr "mode" "SI")])

(define_insn "*zero_extendqisi2_movzbw"
  [(set (match_operand:SI 0 "register_operand" "=r")
     (zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "qm")))]
  "(!TARGET_ZERO_EXTEND_WITH_AND || optimize_size) && reload_completed"
  "movz{bl|x}\t{%1, %0|%0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")])

;; For the movzbl case strip only the clobber
(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && (!TARGET_ZERO_EXTEND_WITH_AND || optimize_size)
   && (!REG_P (operands[1]) || ANY_QI_REG_P (operands[1]))"
  [(set (match_dup 0)
(zero_extend:SI (match_dup 1)))])

;; When source and destination does not overlap, clear destination
;; first and then do the movb
(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && ANY_QI_REG_P (operands[0])
   && (ANY_QI_REG_P (operands[1]) || GET_CODE (operands[1]) == MEM)
   && (TARGET_ZERO_EXTEND_WITH_AND && !optimize_size)
   && !reg_overlap_mentioned_p (operands[0], operands[1])"
  [(set (match_dup 0) (const_int 0))
   (set (strict_low_part (match_dup 2)) (match_dup 1))]
  "operands[2] = gen_lowpart (QImode, operands[0]);")

;; Rest is handled by single and.
(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(zero_extend:SI (match_operand:QI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && true_regnum (operands[0]) == true_regnum (operands[1])"
```

```
  [(parallel [(set (match_dup 0) (and:SI (match_dup 0) (const_int 255)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

;; %%% Kill me once multi-word ops are sane.
(define_expand "zero_extendsidi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
     (zero_extend:DI (match_operand:SI 1 "nonimmediate_operand" "rm")))]
  ""
  "if (!TARGET_64BIT)
     {
       emit_insn (gen_zero_extendsidi2_32 (operands[0], operands[1]));
       DONE;
     }
  ")

(define_insn "zero_extendsidi2_32"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,?r,?*o,?*y,?*Y")
(zero_extend:DI (match_operand:SI 1 "nonimmediate_operand" "0,rm,r,rm,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "@
   #
   #
   #
   movd\t{%1, %0|%0, %1}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "mode" "SI,SI,SI,DI,TI")
   (set_attr "type" "multi,multi,multi,mmxmov,ssemov")])

(define_insn "zero_extendsidi2_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,o,?*y,?*Y")
     (zero_extend:DI (match_operand:SI 1 "nonimmediate_operand" "rm,0,rm,rm")))]
  "TARGET_64BIT"
  "@
   mov\t{%k1, %k0|%k0, %k1}
   #
   movd\t{%1, %0|%0, %1}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "imovx,imov,mmxmov,ssemov")
   (set_attr "mode" "SI,DI,SI,SI")])

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
     (zero_extend:DI (match_dup 0)))]
  "TARGET_64BIT"
  [(set (match_dup 4) (const_int 0))]
  "split_di (&operands[0], 1, &operands[3], &operands[4]);")

(define_split
```

```
  [(set (match_operand:DI 0 "register_operand" "")
(zero_extend:DI (match_operand:SI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && reload_completed
   && true_regnum (operands[0]) == true_regnum (operands[1])"
  [(set (match_dup 4) (const_int 0))]
  "split_di (&operands[0], 1, &operands[3], &operands[4]);")

(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(zero_extend:DI (match_operand:SI 1 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && reload_completed
   && !SSE_REG_P (operands[0]) && !MMX_REG_P (operands[0])"
  [(set (match_dup 3) (match_dup 1))
   (set (match_dup 4) (const_int 0))]
  "split_di (&operands[0], 1, &operands[3], &operands[4]);")

(define_insn "zero_extendhidi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
     (zero_extend:DI (match_operand:HI 1 "nonimmediate_operand" "rm")))]
  "TARGET_64BIT"
  "movz{wl|x}\t{%1, %k0|%k0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "DI")])

(define_insn "zero_extendqidi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
     (zero_extend:DI (match_operand:QI 1 "nonimmediate_operand" "rm")))]
  "TARGET_64BIT"
  "movz{bl|x}\t{%1, %k0|%k0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "DI")])

;; Sign extension instructions

(define_expand "extendsidi2"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (sign_extend:DI (match_operand:SI 1 "register_operand" "")))
      (clobber (reg:CC FLAGS_REG))
      (clobber (match_scratch:SI 2 ""))])]
  ""
{
  if (TARGET_64BIT)
    {
      emit_insn (gen_extendsidi2_rex64 (operands[0], operands[1]));
      DONE;
    }
})
```

```
(define_insn "*extendsidi2_1"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=*A,r,?r,?*o")
(sign_extend:DI (match_operand:SI 1 "register_operand" "0,0,r,r")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (match_scratch:SI 2 "=X,X,X,&r"))]
  "!TARGET_64BIT"
  "#")

(define_insn "extendsidi2_rex64"
  [(set (match_operand:DI 0 "register_operand" "=*a,r")
(sign_extend:DI (match_operand:SI 1 "nonimmediate_operand" "*0,rm")))]
  "TARGET_64BIT"
  "@
   {cltq|cdqe}
   movs{lq|x}\t{%1,%0|%0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "DI")
   (set_attr "prefix_0f" "0")
   (set_attr "modrm" "0,1")])

(define_insn "extendhidi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
(sign_extend:DI (match_operand:HI 1 "nonimmediate_operand" "rm")))]
  "TARGET_64BIT"
  "movs{wq|x}\t{%1,%0|%0, %1}"
  [(set_attr "type" "imovx")
   (set_attr "mode" "DI")])

(define_insn "extendqidi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
(sign_extend:DI (match_operand:QI 1 "nonimmediate_operand" "qm")))]
  "TARGET_64BIT"
  "movs{bq|x}\t{%1,%0|%0, %1}"
   [(set_attr "type" "imovx")
    (set_attr "mode" "DI")])

;; Extend to memory case when source register does die.
(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(sign_extend:DI (match_operand:SI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (match_operand:SI 2 "register_operand" ""))]
  "(reload_completed
    && dead_or_set_p (insn, operands[1])
    && !reg_mentioned_p (operands[1], operands[0]))"
  [(set (match_dup 3) (match_dup 1))
   (parallel [(set (match_dup 1) (ashiftrt:SI (match_dup 1) (const_int 31)))
      (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 4) (match_dup 1))]
  "split_di (&operands[0], 1, &operands[3], &operands[4]);")
```

```
;; Extend to memory case when source register does not die.
(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(sign_extend:DI (match_operand:SI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (match_operand:SI 2 "register_operand" ""))]
  "reload_completed"
  [(const_int 0)]
{
  split_di (&operands[0], 1, &operands[3], &operands[4]);

  emit_move_insn (operands[3], operands[1]);

  /* Generate a cltd if possible and doing so it profitable.  */
  if (true_regnum (operands[1]) == 0
      && true_regnum (operands[2]) == 1
      && (optimize_size || TARGET_USE_CLTD))
    {
      emit_insn (gen_ashrsi3_31 (operands[2], operands[1], GEN_INT (31)));
    }
  else
    {
      emit_move_insn (operands[2], operands[1]);
      emit_insn (gen_ashrsi3_31 (operands[2], operands[2], GEN_INT (31)));
    }
  emit_move_insn (operands[4], operands[2]);
  DONE;
})

;; Extend to register case.  Optimize case where source and destination
;; registers match and cases where we can use cltd.
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(sign_extend:DI (match_operand:SI 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (match_scratch:SI 2 ""))]
  "reload_completed"
  [(const_int 0)]
{
  split_di (&operands[0], 1, &operands[3], &operands[4]);

  if (true_regnum (operands[3]) != true_regnum (operands[1]))
    emit_move_insn (operands[3], operands[1]);

  /* Generate a cltd if possible and doing so it profitable.  */
  if (true_regnum (operands[3]) == 0
      && (optimize_size || TARGET_USE_CLTD))
    {
      emit_insn (gen_ashrsi3_31 (operands[4], operands[3], GEN_INT (31)));
```

```
      DONE;
    }

  if (true_regnum (operands[4]) != true_regnum (operands[1]))
    emit_move_insn (operands[4], operands[1]);

  emit_insn (gen_ashrsi3_31 (operands[4], operands[4], GEN_INT (31)));
  DONE;
})

(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "=*a,r")
(sign_extend:SI (match_operand:HI 1 "nonimmediate_operand" "*0,rm")))]
  ""
{
  switch (get_attr_prefix_0f (insn))
    {
    case 0:
      return "{cwtl|cwde}";
    default:
      return "movs{wl|x}\t{%1,%0|%0, %1}";
    }
}
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")
   (set (attr "prefix_0f")
     ;; movsx is short decodable while cwtl is vector decoded.
     (if_then_else (and (eq_attr "cpu" "!k6")
(eq_attr "alternative" "0"))
(const_string "0")
(const_string "1")))
   (set (attr "modrm")
     (if_then_else (eq_attr "prefix_0f" "0")
(const_string "0")
(const_string "1")))])

(define_insn "*extendhisi2_zext"
  [(set (match_operand:DI 0 "register_operand" "=*a,r")
(zero_extend:DI
  (sign_extend:SI (match_operand:HI 1 "nonimmediate_operand" "*0,rm"))))]
  "TARGET_64BIT"
{
  switch (get_attr_prefix_0f (insn))
    {
    case 0:
      return "{cwtl|cwde}";
    default:
      return "movs{wl|x}\t{%1,%k0|%k0, %1}";
    }
}
```

```
  [(set_attr "type" "imovx")
   (set_attr "mode" "SI")
   (set (attr "prefix_0f")
     ;; movsx is short decodable while cwtl is vector decoded.
     (if_then_else (and (eq_attr "cpu" "!k6")
(eq_attr "alternative" "0"))
(const_string "0")
(const_string "1")))
   (set (attr "modrm")
     (if_then_else (eq_attr "prefix_0f" "0")
(const_string "0")
(const_string "1")))])

(define_insn "extendqihi2"
  [(set (match_operand:HI 0 "register_operand" "=*a,r")
(sign_extend:HI (match_operand:QI 1 "nonimmediate_operand" "*0,qm")))]
  ""
{
  switch (get_attr_prefix_0f (insn))
    {
    case 0:
      return "{cbtw|cbw}";
    default:
      return "movs{bw|x}\t{%1,%0|%0, %1}";
    }
}
  [(set_attr "type" "imovx")
   (set_attr "mode" "HI")
   (set (attr "prefix_0f")
     ;; movsx is short decodable while cwtl is vector decoded.
     (if_then_else (and (eq_attr "cpu" "!k6")
(eq_attr "alternative" "0"))
(const_string "0")
(const_string "1")))
   (set (attr "modrm")
     (if_then_else (eq_attr "prefix_0f" "0")
(const_string "0")
(const_string "1")))])

(define_insn "extendqisi2"
  [(set (match_operand:SI 0 "register_operand" "=r")
(sign_extend:SI (match_operand:QI 1 "nonimmediate_operand" "qm")))]
  ""
  "movs{bl|x}\t{%1,%0|%0, %1}"
   [(set_attr "type" "imovx")
    (set_attr "mode" "SI")])

(define_insn "*extendqisi2_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
```

```
  (sign_extend:SI (match_operand:QI 1 "nonimmediate_operand" "qm"))))]
  "TARGET_64BIT"
  "movs{bl|x}\t{%1,%k0|%k0, %1}"
   [(set_attr "type" "imovx")
    (set_attr "mode" "SI")])

;; Conversions between float and double.

;; These are all no-ops in the model used for the 80387.  So just
;; emit moves.

;; %%% Kill these when call knows how to work out a DFmode push earlier.
(define_insn "*dummy_extendsfdf2"
  [(set (match_operand:DF 0 "push_operand" "=<")
(float_extend:DF (match_operand:SF 1 "nonimmediate_operand" "fY")))]
  "0"
  "#")

(define_split
  [(set (match_operand:DF 0 "push_operand" "")
(float_extend:DF (match_operand:SF 1 "fp_register_operand" "")))]
  "!TARGET_64BIT"
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -8)))
   (set (mem:DF (reg:SI SP_REG)) (float_extend:DF (match_dup 1)))])

(define_split
  [(set (match_operand:DF 0 "push_operand" "")
(float_extend:DF (match_operand:SF 1 "fp_register_operand" "")))]
  "TARGET_64BIT"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -8)))
   (set (mem:DF (reg:DI SP_REG)) (float_extend:DF (match_dup 1)))])

(define_insn "*dummy_extendsfxf2"
  [(set (match_operand:XF 0 "push_operand" "=<")
(float_extend:XF (match_operand:SF 1 "nonimmediate_operand" "f")))]
  "0"
  "#")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(float_extend:XF (match_operand:SF 1 "fp_register_operand" "")))]
  ""
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (match_dup 2)))
   (set (mem:XF (reg:SI SP_REG)) (float_extend:XF (match_dup 1)))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(float_extend:XF (match_operand:SF 1 "fp_register_operand" "")))]
  "TARGET_64BIT"
```

```
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (match_dup 2)))
   (set (mem:DF (reg:DI SP_REG)) (float_extend:XF (match_dup 1)))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(float_extend:XF (match_operand:DF 1 "fp_register_operand" "")))]
  ""
  [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (match_dup 2)))
   (set (mem:DF (reg:SI SP_REG)) (float_extend:XF (match_dup 1)))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

(define_split
  [(set (match_operand:XF 0 "push_operand" "")
(float_extend:XF (match_operand:DF 1 "fp_register_operand" "")))]
  "TARGET_64BIT"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (match_dup 2)))
   (set (mem:XF (reg:DI SP_REG)) (float_extend:XF (match_dup 1)))]
  "operands[2] = GEN_INT (TARGET_128BIT_LONG_DOUBLE ? -16 : -12);")

(define_expand "extendsfdf2"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
        (float_extend:DF (match_operand:SF 1 "general_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
{
  /* ??? Needed for compress_float_constant since all fp constants
     are LEGITIMATE_CONSTANT_P.  */
  if (GET_CODE (operands[1]) == CONST_DOUBLE)
    operands[1] = validize_mem (force_const_mem (SFmode, operands[1]));
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (SFmode, operands[1]);
})

(define_insn "*extendsfdf2_mixed"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=f#Y,m#fY,Y#f")
        (float_extend:DF (match_operand:SF 1 "nonimmediate_operand" "fm#Y,f#Y,mY#f")))]
  "TARGET_SSE2 && TARGET_MIX_SSE_I387
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fst%z0\t%y0";
```

```
    case 2:
      return "cvtss2sd\t{%1, %0|%0, %1}";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,ssecvt")
   (set_attr "mode" "SF,XF,DF")])

(define_insn "*extendsfdf2_sse"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=Y")
        (float_extend:DF (match_operand:SF 1 "nonimmediate_operand" "mY")))]
  "TARGET_SSE2 && TARGET_SSE_MATH
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "cvtss2sd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "DF")])

(define_insn "*extendsfdf2_i387"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=f,m")
        (float_extend:DF (match_operand:SF 1 "nonimmediate_operand" "fm,f")))]
  "TARGET_80387
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fst%z0\t%y0";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF,XF")])

(define_expand "extendsfxf2"
  [(set (match_operand:XF 0 "nonimmediate_operand" "")
        (float_extend:XF (match_operand:SF 1 "general_operand" "")))]
  "TARGET_80387"
{
  /* ??? Needed for compress_float_constant since all fp constants
     are LEGITIMATE_CONSTANT_P.  */
```

```
  if (GET_CODE (operands[1]) == CONST_DOUBLE)
    operands[1] = validize_mem (force_const_mem (SFmode, operands[1]));
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (SFmode, operands[1]);
})

(define_insn "*extendsfxf2_i387"
  [(set (match_operand:XF 0 "nonimmediate_operand" "=f,m")
        (float_extend:XF (match_operand:SF 1 "nonimmediate_operand" "fm,f")))]
  "TARGET_80387
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      /* There is no non-popping store to memory for XFmode.  So if
 we need one, follow the store with a load.  */
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0";
      else
        return "fstp%z0\t%y0\n\tfld%z0\t%y0";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF,XF")])

(define_expand "extenddfxf2"
  [(set (match_operand:XF 0 "nonimmediate_operand" "")
        (float_extend:XF (match_operand:DF 1 "general_operand" "")))]
  "TARGET_80387"
{
  /* ??? Needed for compress_float_constant since all fp constants
     are LEGITIMATE_CONSTANT_P.  */
  if (GET_CODE (operands[1]) == CONST_DOUBLE)
    operands[1] = validize_mem (force_const_mem (DFmode, operands[1]));
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (DFmode, operands[1]);
})

(define_insn "*extenddfxf2_i387"
  [(set (match_operand:XF 0 "nonimmediate_operand" "=f,m")
        (float_extend:XF (match_operand:DF 1 "nonimmediate_operand" "fm,f")))]
  "TARGET_80387
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
```

```
{
  switch (which_alternative)
    {
    case 0:
      return output_387_reg_move (insn, operands);

    case 1:
      /* There is no non-popping store to memory for XFmode.  So if
 we need one, follow the store with a load.  */
      if (! find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
        return "fstp%z0\t%y0\n\tfld%z0\t%y0";
      else
        return "fstp%z0\t%y0";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "DF,XF")])

;; %%% This seems bad bad news.
;; This cannot output into an f-reg because there is no way to be sure
;; of truncating in that case.  Otherwise this is just like a simple move
;; insn.  So we pretend we can output to a reg in order to get better
;; register preferencing, but we really use a stack slot.

;; Conversion from DFmode to SFmode.

(define_expand "truncdfsf2"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
(float_truncate:SF
  (match_operand:DF 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
{
  if (MEM_P (operands[0]) && MEM_P (operands[1]))
    operands[1] = force_reg (DFmode, operands[1]);

  if (TARGET_SSE2 && TARGET_SSE_MATH && !TARGET_MIX_SSE_I387)
    ;
  else if (flag_unsafe_math_optimizations)
    ;
  else
    {
      rtx temp = assign_386_stack_local (SFmode, SLOT_TEMP);
      emit_insn (gen_truncdfsf2_with_temp (operands[0], operands[1], temp));
      DONE;
    }
})
```

```
(define_expand "truncdfsf2_with_temp"
  [(parallel [(set (match_operand:SF 0 "" "")
   (float_truncate:SF (match_operand:DF 1 "" "")))
      (clobber (match_operand:SF 2 "" ""))])]
  "")

(define_insn "*truncdfsf_fast_mixed"
  [(set (match_operand:SF 0 "nonimmediate_operand"   "=m,f,Y")
        (float_truncate:SF
          (match_operand:DF 1 "nonimmediate_operand" "f ,f,Ym")))]
  "TARGET_SSE2 && TARGET_MIX_SSE_I387 && flag_unsafe_math_optimizations"
{
  switch (which_alternative)
    {
    case 0:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
return "fstp%z0\t%y0";
      else
return "fst%z0\t%y0";
    case 1:
      return output_387_reg_move (insn, operands);
    case 2:
      return "cvtsd2ss\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,fmov,ssecvt")
   (set_attr "mode" "SF")])

;; Yes, this one doesn't depend on flag_unsafe_math_optimizations,
;; because nothing we do here is unsafe.
(define_insn "*truncdfsf_fast_sse"
  [(set (match_operand:SF 0 "nonimmediate_operand"   "=Y")
        (float_truncate:SF
          (match_operand:DF 1 "nonimmediate_operand" "Ym")))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "cvtsd2ss\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "SF")])

(define_insn "*truncdfsf_fast_i387"
  [(set (match_operand:SF 0 "nonimmediate_operand"   "=fm")
        (float_truncate:SF
          (match_operand:DF 1 "nonimmediate_operand" "f")))]
  "TARGET_80387 && flag_unsafe_math_optimizations"
  "* return output_387_reg_move (insn, operands);"
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF")])
```

```
(define_insn "*truncdfsf_mixed"
  [(set (match_operand:SF 0 "nonimmediate_operand"   "=m,?fx*r,Y")
(float_truncate:SF
  (match_operand:DF 1 "nonimmediate_operand" "f ,f    ,Ym")))
   (clobber (match_operand:SF 2 "memory_operand"     "=X,m    ,X"))]
  "TARGET_MIX_SSE_I387"
{
  switch (which_alternative)
    {
    case 0:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
return "fstp%z0\t%y0";
      else
return "fst%z0\t%y0";
    case 1:
      return "#";
    case 2:
      return "cvtsd2ss\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,multi,ssecvt")
   (set_attr "unit" "*,i387,*")
   (set_attr "mode" "SF")])

(define_insn "*truncdfsf_i387"
  [(set (match_operand:SF 0 "nonimmediate_operand" "=m,?fx*r")
(float_truncate:SF
  (match_operand:DF 1 "nonimmediate_operand" "f,f")))
   (clobber (match_operand:SF 2 "memory_operand" "=X,m"))]
  "TARGET_80387"
{
  switch (which_alternative)
    {
    case 0:
      if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
return "fstp%z0\t%y0";
      else
return "fst%z0\t%y0";
    case 1:
      return "#";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "fmov,multi")
   (set_attr "unit" "*,i387")
   (set_attr "mode" "SF")])
```

```
(define_insn "*truncdfsf2_i387_1"
  [(set (match_operand:SF 0 "memory_operand" "=m")
(float_truncate:SF
  (match_operand:DF 1 "register_operand" "f")))]
  "TARGET_80387
   && !(TARGET_SSE2 && TARGET_SSE_MATH)
   && !TARGET_MIX_SSE_I387"
{
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
  else
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF")])

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF
 (match_operand:DF 1 "fp_register_operand" "")))
   (clobber (match_operand 2 "" ""))]
  "reload_completed"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
{
  operands[1] = gen_rtx_REG (SFmode, true_regnum (operands[1]));
})

;; Conversion from XFmode to SFmode.

(define_expand "truncxfsf2"
  [(parallel [(set (match_operand:SF 0 "nonimmediate_operand" "")
   (float_truncate:SF
    (match_operand:XF 1 "register_operand" "")))
      (clobber (match_dup 2))])]
  "TARGET_80387"
{
  if (flag_unsafe_math_optimizations)
    {
      rtx reg = REG_P (operands[0]) ? operands[0] : gen_reg_rtx (SFmode);
      emit_insn (gen_truncxfsf2_i387_noop (reg, operands[1]));
      if (reg != operands[0])
emit_move_insn (operands[0], reg);
      DONE;
    }
  else
    operands[2] = assign_386_stack_local (SFmode, SLOT_TEMP);
})

(define_insn "*truncxfsf2_mixed"
```

```
  [(set (match_operand:SF 0 "nonimmediate_operand" "=m,?f#rx,?r#fx,?x#rf")
(float_truncate:SF
 (match_operand:XF 1 "register_operand" "f,f,f,f")))
   (clobber (match_operand:SF 2 "memory_operand" "=X,m,m,m"))]
  "TARGET_MIX_SSE_I387"
{
  gcc_assert (!which_alternative);
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
  else
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov,multi,multi,multi")
   (set_attr "unit" "*,i387,i387,i387")
   (set_attr "mode" "SF")])

(define_insn "truncxfsf2_i387_noop"
  [(set (match_operand:SF 0 "register_operand" "=f")
(float_truncate:SF (match_operand:XF 1 "register_operand" "f")))]
  "TARGET_80387 && flag_unsafe_math_optimizations"
{
  return output_387_reg_move (insn, operands);
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF")])

(define_insn "*truncxfsf2_i387"
  [(set (match_operand:SF 0 "nonimmediate_operand" "=m,?f#r,?r#f")
(float_truncate:SF
 (match_operand:XF 1 "register_operand" "f,f,f")))
   (clobber (match_operand:SF 2 "memory_operand" "=X,m,m"))]
  "TARGET_80387"
{
  gcc_assert (!which_alternative);
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
   else
     return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov,multi,multi")
   (set_attr "unit" "*,i387,i387")
   (set_attr "mode" "SF")])

(define_insn "*truncxfsf2_i387_1"
  [(set (match_operand:SF 0 "memory_operand" "=m")
(float_truncate:SF
 (match_operand:XF 1 "register_operand" "f")))]
  "TARGET_80387"
{
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
```

```
    return "fstp%z0\t%y0";
  else
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "SF")])

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF
 (match_operand:XF 1 "register_operand" "")))
   (clobber (match_operand:SF 2 "memory_operand" ""))]
  "TARGET_80387 && reload_completed"
  [(set (match_dup 2) (float_truncate:SF (match_dup 1)))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_split
  [(set (match_operand:SF 0 "memory_operand" "")
(float_truncate:SF
 (match_operand:XF 1 "register_operand" "")))
   (clobber (match_operand:SF 2 "memory_operand" ""))]
  "TARGET_80387"
  [(set (match_dup 0) (float_truncate:SF (match_dup 1)))]
  "")

;; Conversion from XFmode to DFmode.

(define_expand "truncxfdf2"
  [(parallel [(set (match_operand:DF 0 "nonimmediate_operand" "")
   (float_truncate:DF
    (match_operand:XF 1 "register_operand" "")))
      (clobber (match_dup 2))])]
  "TARGET_80387"
{
  if (flag_unsafe_math_optimizations)
    {
      rtx reg = REG_P (operands[0]) ? operands[0] : gen_reg_rtx (DFmode);
      emit_insn (gen_truncxfdf2_i387_noop (reg, operands[1]));
      if (reg != operands[0])
emit_move_insn (operands[0], reg);
      DONE;
    }
  else
    operands[2] = assign_386_stack_local (DFmode, SLOT_TEMP);
})

(define_insn "*truncxfdf2_mixed"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=m,?f#rY,?r#fY,?Y#rf")
(float_truncate:DF
```

```
 (match_operand:XF 1 "register_operand" "f,f,f,f")))
   (clobber (match_operand:DF 2 "memory_operand" "=X,m,m,m"))]
  "TARGET_SSE2 && TARGET_MIX_SSE_I387"
{
  gcc_assert (!which_alternative);
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
  else
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov,multi,multi,multi")
   (set_attr "unit" "*,i387,i387,i387")
   (set_attr "mode" "DF")])

(define_insn "truncxfdf2_i387_noop"
  [(set (match_operand:DF 0 "register_operand" "=f")
(float_truncate:DF (match_operand:XF 1 "register_operand" "f")))]
  "TARGET_80387 && flag_unsafe_math_optimizations"
{
  return output_387_reg_move (insn, operands);
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "DF")])

(define_insn "*truncxfdf2_i387"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=m,?f#r,?r#f")
(float_truncate:DF
 (match_operand:XF 1 "register_operand" "f,f,f")))
   (clobber (match_operand:DF 2 "memory_operand" "=X,m,m"))]
  "TARGET_80387"
{
  gcc_assert (!which_alternative);
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
  else
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov,multi,multi")
   (set_attr "unit" "*,i387,i387")
   (set_attr "mode" "DF")])

(define_insn "*truncxfdf2_i387_1"
  [(set (match_operand:DF 0 "memory_operand" "=m")
(float_truncate:DF
  (match_operand:XF 1 "register_operand" "f")))]
  "TARGET_80387"
{
  if (find_regno_note (insn, REG_DEAD, REGNO (operands[1])))
    return "fstp%z0\t%y0";
  else
```

```
    return "fst%z0\t%y0";
}
  [(set_attr "type" "fmov")
   (set_attr "mode" "DF")])

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF
 (match_operand:XF 1 "register_operand" "")))
   (clobber (match_operand:DF 2 "memory_operand" ""))]
  "TARGET_80387 && reload_completed"
  [(set (match_dup 2) (float_truncate:DF (match_dup 1)))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_split
  [(set (match_operand:DF 0 "memory_operand" "")
(float_truncate:DF
 (match_operand:XF 1 "register_operand" "")))
   (clobber (match_operand:DF 2 "memory_operand" ""))]
  "TARGET_80387"
  [(set (match_dup 0) (float_truncate:DF (match_dup 1)))]
  "")

;; Signed conversion to DImode.

(define_expand "fix_truncxfdi2"
  [(parallel [(set (match_operand:DI 0 "nonimmediate_operand" "")
                   (fix:DI (match_operand:XF 1 "register_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_80387"
{
  if (TARGET_FISTTP)
   {
     emit_insn (gen_fix_truncdi_fisttp_i387_1 (operands[0], operands[1]));
     DONE;
   }
})

(define_expand "fix_trunc<mode>di2"
  [(parallel [(set (match_operand:DI 0 "nonimmediate_operand" "")
                   (fix:DI (match_operand:SSEMODEF 1 "register_operand" "")))
             (clobber (reg:CC FLAGS_REG))])]
  "TARGET_80387 || (TARGET_64BIT && SSE_FLOAT_MODE_P (<MODE>mode))"
{
  if (TARGET_FISTTP
      && !(TARGET_64BIT && SSE_FLOAT_MODE_P (<MODE>mode) && TARGET_SSE_MATH))
   {
     emit_insn (gen_fix_truncdi_fisttp_i387_1 (operands[0], operands[1]));
     DONE;
```

```
   }
  if (TARGET_64BIT && SSE_FLOAT_MODE_P (<MODE>mode))
   {
     rtx out = REG_P (operands[0]) ? operands[0] : gen_reg_rtx (DImode);
     emit_insn (gen_fix_trunc<mode>di_sse (out, operands[1]));
     if (out != operands[0])
emit_move_insn (operands[0], out);
     DONE;
   }
})

;; Signed conversion to SImode.

(define_expand "fix_truncxfsi2"
  [(parallel [(set (match_operand:SI 0 "nonimmediate_operand" "")
                   (fix:SI (match_operand:XF 1 "register_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_80387"
{
  if (TARGET_FISTTP)
   {
     emit_insn (gen_fix_truncsi_fisttp_i387_1 (operands[0], operands[1]));
     DONE;
   }
})

(define_expand "fix_trunc<mode>si2"
  [(parallel [(set (match_operand:SI 0 "nonimmediate_operand" "")
           (fix:SI (match_operand:SSEMODEF 1 "register_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_80387 || (SSE_FLOAT_MODE_P (<MODE>mode))"
{
  if (TARGET_FISTTP
      && !(SSE_FLOAT_MODE_P (<MODE>mode) && TARGET_SSE_MATH))
   {
     emit_insn (gen_fix_truncsi_fisttp_i387_1 (operands[0], operands[1]));
     DONE;
   }
  if (SSE_FLOAT_MODE_P (<MODE>mode))
   {
     rtx out = REG_P (operands[0]) ? operands[0] : gen_reg_rtx (SImode);
     emit_insn (gen_fix_trunc<mode>si_sse (out, operands[1]));
     if (out != operands[0])
emit_move_insn (operands[0], out);
     DONE;
   }
})

;; Signed conversion to HImode.
```

```
(define_expand "fix_trunc<mode>hi2"
  [(parallel [(set (match_operand:HI 0 "nonimmediate_operand" "")
          (fix:HI (match_operand:X87MODEF 1 "register_operand" "")))
             (clobber (reg:CC FLAGS_REG))])]
  "TARGET_80387
   && !(SSE_FLOAT_MODE_P (<MODE>mode) && (!TARGET_FISTTP || TARGET_SSE_MATH))"
{
  if (TARGET_FISTTP)
   {
     emit_insn (gen_fix_trunchi_fisttp_i387_1 (operands[0], operands[1]));
     DONE;
   }
})

;; When SSE is available, it is always faster to use it!
(define_insn "fix_truncsfdi_sse"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(fix:DI (match_operand:SF 1 "nonimmediate_operand" "x,xm")))]
  "TARGET_64BIT && TARGET_SSE && (!TARGET_FISTTP || TARGET_SSE_MATH)"
  "cvttss2si{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "SF")
   (set_attr "athlon_decode" "double,vector")])

(define_insn "fix_truncdfdi_sse"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(fix:DI (match_operand:DF 1 "nonimmediate_operand" "Y,Ym")))]
  "TARGET_64BIT && TARGET_SSE2 && (!TARGET_FISTTP || TARGET_SSE_MATH)"
  "cvttsd2si{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,vector")])

(define_insn "fix_truncsfsi_sse"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(fix:SI (match_operand:SF 1 "nonimmediate_operand" "x,xm")))]
  "TARGET_SSE && (!TARGET_FISTTP || TARGET_SSE_MATH)"
  "cvttss2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,vector")])

(define_insn "fix_truncdfsi_sse"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(fix:SI (match_operand:DF 1 "nonimmediate_operand" "Y,Ym")))]
  "TARGET_SSE2 && (!TARGET_FISTTP || TARGET_SSE_MATH)"
  "cvttsd2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,vector")])
```

```
;; Avoid vector decoded forms of the instruction.
(define_peephole2
  [(match_scratch:DF 2 "Y")
   (set (match_operand:SSEMODEI24 0 "register_operand" "")
(fix:SSEMODEI24 (match_operand:DF 1 "memory_operand" "")))]
  "TARGET_K8 && !optimize_size"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (fix:SSEMODEI24 (match_dup 2)))]
  "")

(define_peephole2
  [(match_scratch:SF 2 "x")
   (set (match_operand:SSEMODEI24 0 "register_operand" "")
(fix:SSEMODEI24 (match_operand:SF 1 "memory_operand" "")))]
  "TARGET_K8 && !optimize_size"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (fix:SSEMODEI24 (match_dup 2)))]
  "")

(define_insn_and_split "fix_trunc<mode>_fisttp_i387_1"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(fix:X87MODEI (match_operand 1 "register_operand" "f,f")))]
  "TARGET_80387 && TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !((SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
 && (TARGET_64BIT || <MODE>mode != DImode))
&& TARGET_SSE_MATH)
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  if (memory_operand (operands[0], VOIDmode))
    emit_insn (gen_fix_trunc<mode>_i387_fisttp (operands[0], operands[1]));
  else
    {
      operands[2] = assign_386_stack_local (<MODE>mode, SLOT_TEMP);
      emit_insn (gen_fix_trunc<mode>_i387_fisttp_with_temp (operands[0],
    operands[1],
    operands[2]));
    }
  DONE;
}
  [(set_attr "type" "fisttp")
   (set_attr "mode" "<MODE>")])

(define_insn "fix_trunc<mode>_i387_fisttp"
  [(set (match_operand:X87MODEI 0 "memory_operand" "=m")
(fix:X87MODEI (match_operand 1 "register_operand" "f")))
```

```
   (clobber (match_scratch:XF 2 "=&1f"))]
  "TARGET_80387 && TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !((SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
 && (TARGET_64BIT || <MODE>mode != DImode))
&& TARGET_SSE_MATH)"
  "* return output_fix_trunc (insn, operands, 1);"
  [(set_attr "type" "fisttp")
   (set_attr "mode" "<MODE>")])

(define_insn "fix_trunc<mode>_i387_fisttp_with_temp"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(fix:X87MODEI (match_operand 1 "register_operand" "f,f")))
   (clobber (match_operand:X87MODEI 2 "memory_operand" "=m,m"))
   (clobber (match_scratch:XF 3 "=&1f,&1f"))]
  "TARGET_80387 && TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !((SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
&& (TARGET_64BIT || <MODE>mode != DImode))
&& TARGET_SSE_MATH)"
  "#"
  [(set_attr "type" "fisttp")
   (set_attr "mode" "<MODE>")])

(define_split
  [(set (match_operand:X87MODEI 0 "register_operand" "")
(fix:X87MODEI (match_operand 1 "register_operand" "")))
   (clobber (match_operand:X87MODEI 2 "memory_operand" ""))
   (clobber (match_scratch 3 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 2) (fix:X87MODEI (match_dup 1)))
      (clobber (match_dup 3))])
   (set (match_dup 0) (match_dup 2))]
  "")

(define_split
  [(set (match_operand:X87MODEI 0 "memory_operand" "")
(fix:X87MODEI (match_operand 1 "register_operand" "")))
   (clobber (match_operand:X87MODEI 2 "memory_operand" ""))
   (clobber (match_scratch 3 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (fix:X87MODEI (match_dup 1)))
      (clobber (match_dup 3))])]
  "")

;; See the comments in i386.h near OPTIMIZE_MODE_SWITCHING for the description
;; of the machinery. Please note the clobber of FLAGS_REG. In i387 control
;; word calculation (inserted by LCM in mode switching pass) a FLAGS_REG
;; clobbering insns can be used. Look at emit_i387_cw_initialization ()
;; function in i386.c.
```

```
(define_insn_and_split "*fix_trunc<mode>_i387_1"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(fix:X87MODEI (match_operand 1 "register_operand" "f,f")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_80387 && !TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !(SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
 && (TARGET_64BIT || <MODE>mode != DImode))
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_TRUNC] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_TRUNC);
  if (memory_operand (operands[0], VOIDmode))
    emit_insn (gen_fix_trunc<mode>_i387 (operands[0], operands[1],
 operands[2], operands[3]));
  else
    {
      operands[4] = assign_386_stack_local (<MODE>mode, SLOT_TEMP);
      emit_insn (gen_fix_trunc<mode>_i387_with_temp (operands[0], operands[1],
     operands[2], operands[3],
     operands[4]));
    }
  DONE;
}
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "<MODE>")])

(define_insn "fix_truncdi_i387"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(fix:DI (match_operand 1 "register_operand" "f")))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))
   (clobber (match_scratch:XF 4 "=&1f"))]
  "TARGET_80387 && !TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !(TARGET_64BIT && SSE_FLOAT_MODE_P (GET_MODE (operands[1])))"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "DI")])

(define_insn "fix_truncdi_i387_with_temp"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=m,?r")
(fix:DI (match_operand 1 "register_operand" "f,f")))
```

```
   (use (match_operand:HI 2 "memory_operand" "m,m"))
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:DI 4 "memory_operand" "=m,m"))
   (clobber (match_scratch:XF 5 "=&1f,&1f"))]
  "TARGET_80387 && !TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !(TARGET_64BIT && SSE_FLOAT_MODE_P (GET_MODE (operands[1])))"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(fix:DI (match_operand 1 "register_operand" "")))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 4) (fix:DI (match_dup 1)))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(fix:DI (match_operand 1 "register_operand" "")))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (fix:DI (match_dup 1)))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])]
  "")

(define_insn "fix_trunc<mode>_i387"
  [(set (match_operand:X87MODEI12 0 "memory_operand" "=m")
(fix:X87MODEI12 (match_operand 1 "register_operand" "f")))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_80387 && !TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !SSE_FLOAT_MODE_P (GET_MODE (operands[1]))"
  "* return output_fix_trunc (insn, operands, 0);"
```

```
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "<MODE>")])

(define_insn "fix_trunc<mode>_i387_with_temp"
  [(set (match_operand:X87MODEI12 0 "nonimmediate_operand" "=m,?r")
(fix:X87MODEI12 (match_operand 1 "register_operand" "f,f")))
   (use (match_operand:HI 2 "memory_operand" "m,m"))
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" "=m,m"))]
  "TARGET_80387 && !TARGET_FISTTP
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && !SSE_FLOAT_MODE_P (GET_MODE (operands[1]))"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "<MODE>")])

(define_split
  [(set (match_operand:X87MODEI12 0 "register_operand" "")
(fix:X87MODEI12 (match_operand 1 "register_operand" "")))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 4) (fix:X87MODEI12 (match_dup 1)))
      (use (match_dup 2))
      (use (match_dup 3))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:X87MODEI12 0 "memory_operand" "")
(fix:X87MODEI12 (match_operand 1 "register_operand" "")))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (fix:X87MODEI12 (match_dup 1)))
      (use (match_dup 2))
      (use (match_dup 3))])]
  "")

(define_insn "x86_fnstcw_1"
  [(set (match_operand:HI 0 "memory_operand" "=m")
(unspec:HI [(reg:HI FPSR_REG)] UNSPEC_FSTCW))]
  "TARGET_80387"
  "fnstcw\t%0"
  [(set_attr "length" "2")
   (set_attr "mode" "HI")
```

```
   (set_attr "unit" "i387")])

(define_insn "x86_fldcw_1"
  [(set (reg:HI FPSR_REG)
(unspec:HI [(match_operand:HI 0 "memory_operand" "m")] UNSPEC_FLDCW))]
  "TARGET_80387"
  "fldcw\t%0"
  [(set_attr "length" "2")
   (set_attr "mode" "HI")
   (set_attr "unit" "i387")
   (set_attr "athlon_decode" "vector")])

;; Conversion between fixed point and floating point.

;; Even though we only accept memory inputs, the backend _really_
;; wants to be able to do this between registers.

(define_expand "floathisf2"
  [(set (match_operand:SF 0 "register_operand" "")
(float:SF (match_operand:HI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
{
  if (TARGET_SSE_MATH)
    {
      emit_insn (gen_floatsisf2 (operands[0],
 convert_to_mode (SImode, operands[1], 0)));
      DONE;
    }
})

(define_insn "*floathisf2_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(float:SF (match_operand:HI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387 && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "SF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_expand "floatsisf2"
  [(set (match_operand:SF 0 "register_operand" "")
(float:SF (match_operand:SI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "")

(define_insn "*floatsisf2_mixed"
  [(set (match_operand:SF 0 "register_operand" "=f#x,?f#x,x#f,x#f")
```

```
(float:SF (match_operand:SI 1 "nonimmediate_operand" "m,r,r,mr")))]
  "TARGET_MIX_SSE_I387"
  "@
   fild%z1\t%1
   #
   cvtsi2ss\t{%1, %0|%0, %1}
   cvtsi2ss\t{%1, %0|%0, %1}"
  [(set_attr "type" "fmov,multi,sseicvt,sseicvt")
   (set_attr "mode" "SF")
   (set_attr "unit" "*,i387,*,*")
   (set_attr "athlon_decode" "*,*,vector,double")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatsisf2_sse"
  [(set (match_operand:SF 0 "register_operand" "=x,x")
(float:SF (match_operand:SI 1 "nonimmediate_operand" "r,mr")))]
  "TARGET_SSE_MATH"
  "cvtsi2ss\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "SF")
   (set_attr "athlon_decode" "vector,double")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatsisf2_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(float:SF (match_operand:SI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "SF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_expand "floatdisf2"
  [(set (match_operand:SF 0 "register_operand" "")
(float:SF (match_operand:DI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_64BIT && TARGET_SSE_MATH)"
  "")

(define_insn "*floatdisf2_mixed"
  [(set (match_operand:SF 0 "register_operand" "=f#x,?f#x,x#f,x#f")
(float:SF (match_operand:DI 1 "nonimmediate_operand" "m,r,r,mr")))]
  "TARGET_64BIT && TARGET_MIX_SSE_I387"
  "@
   fild%z1\t%1
   #
   cvtsi2ss{q}\t{%1, %0|%0, %1}
   cvtsi2ss{q}\t{%1, %0|%0, %1}"
```

```
  [(set_attr "type" "fmov,multi,sseicvt,sseicvt")
   (set_attr "mode" "SF")
   (set_attr "unit" "*,i387,*,*")
   (set_attr "athlon_decode" "*,*,vector,double")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatdisf2_sse"
  [(set (match_operand:SF 0 "register_operand" "=x,x")
(float:SF (match_operand:DI 1 "nonimmediate_operand" "r,mr")))]
  "TARGET_64BIT && TARGET_SSE_MATH"
  "cvtsi2ss{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "SF")
   (set_attr "athlon_decode" "vector,double")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatdisf2_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(float:SF (match_operand:DI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "SF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_expand "floathidf2"
  [(set (match_operand:DF 0 "register_operand" "")
(float:DF (match_operand:HI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
{
  if (TARGET_SSE2 && TARGET_SSE_MATH)
    {
      emit_insn (gen_floatsidf2 (operands[0],
 convert_to_mode (SImode, operands[1], 0)));
      DONE;
    }
})

(define_insn "*floathidf2_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(float:DF (match_operand:HI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387 && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "DF")
```

```
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_expand "floatsidf2"
  [(set (match_operand:DF 0 "register_operand" "")
(float:DF (match_operand:SI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "")

(define_insn "*floatsidf2_mixed"
  [(set (match_operand:DF 0 "register_operand" "=f#Y,?f#Y,Y#f,Y#f")
(float:DF (match_operand:SI 1 "nonimmediate_operand" "m,r,r,mr")))]
  "TARGET_SSE2 && TARGET_MIX_SSE_I387"
  "@
   fild%z1\t%1
   #
   cvtsi2sd\t{%1, %0|%0, %1}
   cvtsi2sd\t{%1, %0|%0, %1}"
  [(set_attr "type" "fmov,multi,sseicvt,sseicvt")
   (set_attr "mode" "DF")
   (set_attr "unit" "*,i387,*,*")
   (set_attr "athlon_decode" "*,*,double,direct")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatsidf2_sse"
  [(set (match_operand:DF 0 "register_operand" "=Y,Y")
(float:DF (match_operand:SI 1 "nonimmediate_operand" "r,mr")))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "cvtsi2sd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,direct")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatsidf2_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(float:DF (match_operand:SI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "DF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_expand "floatdidf2"
  [(set (match_operand:DF 0 "register_operand" "")
(float:DF (match_operand:DI 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_64BIT && TARGET_SSE2 && TARGET_SSE_MATH)"
```

```
  "")

(define_insn "*floatdidf2_mixed"
  [(set (match_operand:DF 0 "register_operand" "=f#Y,?f#Y,Y#f,Y#f")
(float:DF (match_operand:DI 1 "nonimmediate_operand" "m,r,r,mr")))]
  "TARGET_64BIT && TARGET_SSE2 && TARGET_MIX_SSE_I387"
  "@
   fild%z1\t%1
   #
   cvtsi2sd{q}\t{%1, %0|%0, %1}
   cvtsi2sd{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "fmov,multi,sseicvt,sseicvt")
   (set_attr "mode" "DF")
   (set_attr "unit" "*,i387,*,*")
   (set_attr "athlon_decode" "*,*,double,direct")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatdidf2_sse"
  [(set (match_operand:DF 0 "register_operand" "=Y,Y")
(float:DF (match_operand:DI 1 "nonimmediate_operand" "r,mr")))]
  "TARGET_64BIT && TARGET_SSE2 && TARGET_SSE_MATH"
  "cvtsi2sd{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,direct")
   (set_attr "fp_int_src" "true")])

(define_insn "*floatdidf2_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(float:DF (match_operand:DI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "DF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_insn "floathixf2"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(float:XF (match_operand:HI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "XF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])
```

```
(define_insn "floatsixf2"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(float:XF (match_operand:SI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "XF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

(define_insn "floatdixf2"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(float:XF (match_operand:DI 1 "nonimmediate_operand" "m,?r")))]
  "TARGET_80387"
  "@
   fild%z1\t%1
   #"
  [(set_attr "type" "fmov,multi")
   (set_attr "mode" "XF")
   (set_attr "unit" "*,i387")
   (set_attr "fp_int_src" "true")])

;; %%% Kill these when reload knows how to do it.
(define_split
  [(set (match_operand 0 "fp_register_operand" "")
(float (match_operand 1 "register_operand" "")))]
  "reload_completed
   && TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[0]))"
  [(const_int 0)]
{
  operands[2] = ix86_force_to_memory (GET_MODE (operands[1]), operands[1]);
  operands[2] = gen_rtx_FLOAT (GET_MODE (operands[0]), operands[2]);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], operands[2]));
  ix86_free_from_memory (GET_MODE (operands[1]));
  DONE;
})

(define_expand "floatunssisf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SI 1 "register_operand" ""))]
  "!TARGET_64BIT && TARGET_SSE_MATH"
  "x86_emit_floatuns (operands); DONE;")

(define_expand "floatunsdisf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:DI 1 "register_operand" ""))]
```

```
  "TARGET_64BIT && TARGET_SSE_MATH"
  "x86_emit_floatuns (operands); DONE;")

(define_expand "floatunsdidf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DI 1 "register_operand" ""))]
  "TARGET_64BIT && TARGET_SSE2 && TARGET_SSE_MATH"
  "x86_emit_floatuns (operands); DONE;")

;; SSE extract/set expanders


;; Add instructions

;; %%% splits for addditi3

(define_expand "addti3"
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
(plus:TI (match_operand:TI 1 "nonimmediate_operand" "")
 (match_operand:TI 2 "x86_64_general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (PLUS, TImode, operands); DONE;")

(define_insn "*addti3_1"
  [(set (match_operand:TI 0 "nonimmediate_operand" "=r,o")
(plus:TI (match_operand:TI 1 "nonimmediate_operand" "%0,0")
 (match_operand:TI 2 "general_operand" "roiF,riF")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, TImode, operands)"
  "#")

(define_split
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
(plus:TI (match_operand:TI 1 "nonimmediate_operand" "")
 (match_operand:TI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(parallel [(set (reg:CC FLAGS_REG) (unspec:CC [(match_dup 1) (match_dup 2)]
  UNSPEC_ADD_CARRY))
      (set (match_dup 0) (plus:DI (match_dup 1) (match_dup 2)))])
   (parallel [(set (match_dup 3)
   (plus:DI (plus:DI (ltu:DI (reg:CC FLAGS_REG) (const_int 0))
     (match_dup 4))
    (match_dup 5)))
      (clobber (reg:CC FLAGS_REG))])]
  "split_ti (operands+0, 1, operands+0, operands+3);
   split_ti (operands+1, 1, operands+1, operands+4);
   split_ti (operands+2, 1, operands+2, operands+5);")
```

```
;; %%% splits for addsidi3
;  [(set (match_operand:DI 0 "nonimmediate_operand" "")
; (plus:DI (match_operand:DI 1 "general_operand" "")
;  (zero_extend:DI (match_operand:SI 2 "general_operand" ""))))]

(define_expand "adddi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(plus:DI (match_operand:DI 1 "nonimmediate_operand" "")
 (match_operand:DI 2 "x86_64_general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (PLUS, DImode, operands); DONE;")

(define_insn "*adddi3_1"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,o")
(plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
 (match_operand:DI 2 "general_operand" "roiF,riF")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && ix86_binary_operator_ok (PLUS, DImode, operands)"
  "#")

(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(plus:DI (match_operand:DI 1 "nonimmediate_operand" "")
 (match_operand:DI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && reload_completed"
  [(parallel [(set (reg:CC FLAGS_REG) (unspec:CC [(match_dup 1) (match_dup 2)]
  UNSPEC_ADD_CARRY))
      (set (match_dup 0) (plus:SI (match_dup 1) (match_dup 2)))])
   (parallel [(set (match_dup 3)
   (plus:SI (plus:SI (ltu:SI (reg:CC FLAGS_REG) (const_int 0))
     (match_dup 4))
    (match_dup 5)))
     (clobber (reg:CC FLAGS_REG))])]
  "split_di (operands+0, 1, operands+0, operands+3);
   split_di (operands+1, 1, operands+1, operands+4);
   split_di (operands+2, 1, operands+2, operands+5);")

(define_insn "adddi3_carry_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
  (plus:DI (plus:DI (match_operand:DI 3 "ix86_carry_flag_operator" "")
    (match_operand:DI 1 "nonimmediate_operand" "%0,0"))
   (match_operand:DI 2 "x86_64_general_operand" "re,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, DImode, operands)"
  "adc{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "DI")])
```

```
(define_insn "*adddi3_cc_rex64"
  [(set (reg:CC FLAGS_REG)
(unspec:CC [(match_operand:DI 1 "nonimmediate_operand" "%0,0")
    (match_operand:DI 2 "x86_64_general_operand" "re,rm")]
   UNSPEC_ADD_CARRY))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(plus:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, DImode, operands)"
  "add{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "addqi3_carry"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q")
  (plus:QI (plus:QI (match_operand:QI 3 "ix86_carry_flag_operator" "")
    (match_operand:QI 1 "nonimmediate_operand" "%0,0"))
   (match_operand:QI 2 "general_operand" "qi,qm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (PLUS, QImode, operands)"
  "adc{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "QI")])

(define_insn "addhi3_carry"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
  (plus:HI (plus:HI (match_operand:HI 3 "ix86_carry_flag_operator" "")
    (match_operand:HI 1 "nonimmediate_operand" "%0,0"))
   (match_operand:HI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (PLUS, HImode, operands)"
  "adc{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "HI")])

(define_insn "addsi3_carry"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
  (plus:SI (plus:SI (match_operand:SI 3 "ix86_carry_flag_operator" "")
    (match_operand:SI 1 "nonimmediate_operand" "%0,0"))
   (match_operand:SI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (PLUS, SImode, operands)"
  "adc{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "SI")])

(define_insn "*addsi3_carry_zext"
```

```
  [(set (match_operand:DI 0 "register_operand" "=r")
  (zero_extend:DI
    (plus:SI (plus:SI (match_operand:SI 3 "ix86_carry_flag_operator" "")
      (match_operand:SI 1 "nonimmediate_operand" "%0"))
     (match_operand:SI 2 "general_operand" "rim"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, SImode, operands)"
  "adc{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "SI")])

(define_insn "*addsi3_cc"
  [(set (reg:CC FLAGS_REG)
(unspec:CC [(match_operand:SI 1 "nonimmediate_operand" "%0,0")
    (match_operand:SI 2 "general_operand" "ri,rm")]
   UNSPEC_ADD_CARRY))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(plus:SI (match_dup 1) (match_dup 2)))]
  "ix86_binary_operator_ok (PLUS, SImode, operands)"
  "add{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "addqi3_cc"
  [(set (reg:CC FLAGS_REG)
(unspec:CC [(match_operand:QI 1 "nonimmediate_operand" "%0,0")
    (match_operand:QI 2 "general_operand" "qi,qm")]
   UNSPEC_ADD_CARRY))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm,q")
(plus:QI (match_dup 1) (match_dup 2)))]
  "ix86_binary_operator_ok (PLUS, QImode, operands)"
  "add{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_expand "addsi3"
  [(parallel [(set (match_operand:SI 0 "nonimmediate_operand" "")
   (plus:SI (match_operand:SI 1 "nonimmediate_operand" "")
    (match_operand:SI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_binary_operator (PLUS, SImode, operands); DONE;")

(define_insn "*lea_1"
  [(set (match_operand:SI 0 "register_operand" "=r")
(match_operand:SI 1 "no_seg_address_operand" "p"))]
  "!TARGET_64BIT"
  "lea{l}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "lea")
```

```
   (set_attr "mode" "SI")])

(define_insn "*lea_1_rex64"
  [(set (match_operand:SI 0 "register_operand" "=r")
(subreg:SI (match_operand:DI 1 "no_seg_address_operand" "p") 0))]
  "TARGET_64BIT"
  "lea{l}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn "*lea_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
 (subreg:SI (match_operand:DI 1 "no_seg_address_operand" "p") 0)))]
  "TARGET_64BIT"
  "lea{l}\t{%a1, %k0|%k0, %a1}"
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn "*lea_2_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(match_operand:DI 1 "no_seg_address_operand" "p"))]
  "TARGET_64BIT"
  "lea{q}\t{%a1, %0|%0, %a1}"
  [(set_attr "type" "lea")
   (set_attr "mode" "DI")])

;; The lea patterns for non-Pmodes needs to be matched by several
;; insns converted to real lea by splitters.

(define_insn_and_split "*lea_general_1"
  [(set (match_operand 0 "register_operand" "=r")
(plus (plus (match_operand 1 "index_register_operand" "l")
    (match_operand 2 "register_operand" "r"))
      (match_operand 3 "immediate_operand" "i")))]
  "(GET_MODE (operands[0]) == QImode || GET_MODE (operands[0]) == HImode
    || (TARGET_64BIT && GET_MODE (operands[0]) == SImode))
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)
   && GET_MODE (operands[0]) == GET_MODE (operands[1])
   && GET_MODE (operands[0]) == GET_MODE (operands[2])
   && (GET_MODE (operands[0]) == GET_MODE (operands[3])
       || GET_MODE (operands[3]) == VOIDmode)"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  rtx pat;
  operands[0] = gen_lowpart (SImode, operands[0]);
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[2] = gen_lowpart (Pmode, operands[2]);
```

```
  operands[3] = gen_lowpart (Pmode, operands[3]);
  pat = gen_rtx_PLUS (Pmode, gen_rtx_PLUS (Pmode, operands[1], operands[2]),
         operands[3]);
  if (Pmode != SImode)
    pat = gen_rtx_SUBREG (SImode, pat, 0);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], pat));
  DONE;
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn_and_split "*lea_general_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (plus:SI (plus:SI (match_operand:SI 1 "index_register_operand" "l")
    (match_operand:SI 2 "register_operand" "r"))
   (match_operand:SI 3 "immediate_operand" "i"))))]
  "TARGET_64BIT"
  "#"
  "&& reload_completed"
  [(set (match_dup 0)
(zero_extend:DI (subreg:SI (plus:DI (plus:DI (match_dup 1)
     (match_dup 2))
    (match_dup 3)) 0)))]
{
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[2] = gen_lowpart (Pmode, operands[2]);
  operands[3] = gen_lowpart (Pmode, operands[3]);
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn_and_split "*lea_general_2"
  [(set (match_operand 0 "register_operand" "=r")
(plus (mult (match_operand 1 "index_register_operand" "l")
    (match_operand 2 "const248_operand" "i"))
      (match_operand 3 "nonmemory_operand" "ri")))]
  "(GET_MODE (operands[0]) == QImode || GET_MODE (operands[0]) == HImode
    || (TARGET_64BIT && GET_MODE (operands[0]) == SImode))
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)
   && GET_MODE (operands[0]) == GET_MODE (operands[1])
   && (GET_MODE (operands[0]) == GET_MODE (operands[3])
       || GET_MODE (operands[3]) == VOIDmode)"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  rtx pat;
  operands[0] = gen_lowpart (SImode, operands[0]);
  operands[1] = gen_lowpart (Pmode, operands[1]);
```

```
  operands[3] = gen_lowpart (Pmode, operands[3]);
  pat = gen_rtx_PLUS (Pmode, gen_rtx_MULT (Pmode, operands[1], operands[2]),
         operands[3]);
  if (Pmode != SImode)
    pat = gen_rtx_SUBREG (SImode, pat, 0);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], pat));
  DONE;
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn_and_split "*lea_general_2_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (plus:SI (mult:SI (match_operand:SI 1 "index_register_operand" "l")
    (match_operand:SI 2 "const248_operand" "n"))
   (match_operand:SI 3 "nonmemory_operand" "ri"))))]
  "TARGET_64BIT"
  "#"
  "&& reload_completed"
  [(set (match_dup 0)
(zero_extend:DI (subreg:SI (plus:DI (mult:DI (match_dup 1)
     (match_dup 2))
    (match_dup 3)) 0)))]
{
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[3] = gen_lowpart (Pmode, operands[3]);
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn_and_split "*lea_general_3"
  [(set (match_operand 0 "register_operand" "=r")
(plus (plus (mult (match_operand 1 "index_register_operand" "l")
  (match_operand 2 "const248_operand" "i"))
    (match_operand 3 "register_operand" "r"))
      (match_operand 4 "immediate_operand" "i")))]
  "(GET_MODE (operands[0]) == QImode || GET_MODE (operands[0]) == HImode
    || (TARGET_64BIT && GET_MODE (operands[0]) == SImode))
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)
   && GET_MODE (operands[0]) == GET_MODE (operands[1])
   && GET_MODE (operands[0]) == GET_MODE (operands[3])"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  rtx pat;
  operands[0] = gen_lowpart (SImode, operands[0]);
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[3] = gen_lowpart (Pmode, operands[3]);
```

```
  operands[4] = gen_lowpart (Pmode, operands[4]);
  pat = gen_rtx_PLUS (Pmode,
         gen_rtx_PLUS (Pmode, gen_rtx_MULT (Pmode, operands[1],
        operands[2]),
    operands[3]),
          operands[4]);
  if (Pmode != SImode)
    pat = gen_rtx_SUBREG (SImode, pat, 0);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], pat));
  DONE;
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn_and_split "*lea_general_3_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (plus:SI (plus:SI (mult:SI
      (match_operand:SI 1 "index_register_operand" "l")
      (match_operand:SI 2 "const248_operand" "n"))
    (match_operand:SI 3 "register_operand" "r"))
   (match_operand:SI 4 "immediate_operand" "i"))))]
  "TARGET_64BIT"
  "#"
  "&& reload_completed"
  [(set (match_dup 0)
(zero_extend:DI (subreg:SI (plus:DI (plus:DI (mult:DI (match_dup 1)
      (match_dup 2))
     (match_dup 3))
    (match_dup 4)) 0)))]
{
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[3] = gen_lowpart (Pmode, operands[3]);
  operands[4] = gen_lowpart (Pmode, operands[4]);
}
  [(set_attr "type" "lea")
   (set_attr "mode" "SI")])

(define_insn "*adddi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,rm,r")
(plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0,r")
 (match_operand:DI 2 "x86_64_general_operand" "rme,re,le")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, DImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      operands[2] = SET_SRC (XVECEXP (PATTERN (insn), 0, 0));
      return "lea{q}\t{%a2, %0|%0, %a2}";
```

```
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{q}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{q}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));

      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1)))
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{q}\t{%2, %0|%0, %2}";
        }
      return "add{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "2")
      (const_string "lea")
    ; Current assemblers are broken and do not allow @GOTOFF in
    ; ought but a memory context.
    (match_operand:DI 2 "pic_symbolic_operand" "")
      (const_string "lea")
    (match_operand:DI 2 "incdec_operand" "")
      (const_string "incdec")
   ]
   (const_string "alu")))
   (set_attr "mode" "DI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(plus:DI (match_operand:DI 1 "register_operand" "")
 (match_operand:DI 2 "x86_64_nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])"
```

```
  [(set (match_dup 0)
(plus:DI (match_dup 1)
 (match_dup 2)))]
  "")

(define_insn "*adddi_2_rex64"
  [(set (reg FLAGS_REG)
(compare
  (plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
   (match_operand:DI 2 "x86_64_general_operand" "rme,re"))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=r,rm")
(plus:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (PLUS, DImode, operands)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{q}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{q}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* ???? We ought to handle there the 32bit case too
 - do we need new constraint?  */
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1)))
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{q}\t{%2, %0|%0, %2}";
        }
      return "add{q}\t{%2, %0|%0, %2}";
    }
}
```

```
  [(set (attr "type")
     (if_then_else (match_operand:DI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "DI")])

(define_insn "*adddi_3_rex64"
  [(set (reg FLAGS_REG)
(compare (neg:DI (match_operand:DI 2 "x86_64_general_operand" "rme"))
 (match_operand:DI 1 "x86_64_general_operand" "%0")))
   (clobber (match_scratch:DI 0 "=r"))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCZmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{q}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{q}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* ???? We ought to handle there the 32bit case too
 - do we need new constraint?  */
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1)))
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{q}\t{%2, %0|%0, %2}";
        }
      return "add{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
```

```
     (if_then_else (match_operand:DI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "DI")])

; For comparisons against 1, -1 and 128, we may generate better code
; by converting cmp to add, inc or dec as done by peephole2.  This pattern
; is matched then.  We can't accept general immediate, because for
; case of overflows,  the result is messed up.
; This pattern also don't hold of 0x8000000000000000, since the value overflows
; when negated.
; Also carry flag is reversed compared to cmp, so this conversion is valid
; only for comparisons not depending on it.
(define_insn "*adddi_4_rex64"
  [(set (reg FLAGS_REG)
(compare (match_operand:DI 1 "nonimmediate_operand" "0")
 (match_operand:DI 2 "x86_64_immediate_operand" "e")))
   (clobber (match_scratch:DI 0 "=rm"))]
  "TARGET_64BIT
   &&  ix86_match_ccmode (insn, CCGCmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == constm1_rtx)
        return "inc{q}\t%0";
      else
        {
  gcc_assert (operands[2] == const1_rtx);
          return "dec{q}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if ((INTVAL (operands[2]) == -128
   || (INTVAL (operands[2]) > 0
       && INTVAL (operands[2]) != 128))
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1))))
return "sub{q}\t{%2, %0|%0, %2}";
      operands[2] = GEN_INT (-INTVAL (operands[2]));
      return "add{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:DI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
```

```
   (set_attr "mode" "DI")])

(define_insn "*adddi_5_rex64"
  [(set (reg FLAGS_REG)
(compare
  (plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0")
   (match_operand:DI 2 "x86_64_general_operand" "rme"))
  (const_int 0)))
   (clobber (match_scratch:DI 0 "=r"))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCGOCmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{q}\t%0";
      else
        {
          gcc_assert (operands[2] == constm1_rtx);
          return "dec{q}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1)))
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{q}\t{%2, %0|%0, %2}";
        }
      return "add{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:DI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "DI")])
```

```
(define_insn "*addsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,rm,r")
(plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0,r")
 (match_operand:SI 2 "general_operand" "rmni,rni,lni")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (PLUS, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      operands[2] = SET_SRC (XVECEXP (PATTERN (insn), 0, 0));
      return "lea{l}\t{%a2, %0|%0, %a2}";

    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{l}\t%0";
      else
{
     gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));

      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %0|%0, %2}";
        }
      return "add{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "2")
      (const_string "lea")
    ; Current assemblers are broken and do not allow @GOTOFF in
    ; ought but a memory context.
    (match_operand:SI 2 "pic_symbolic_operand" "")
      (const_string "lea")
    (match_operand:SI 2 "incdec_operand" "")
      (const_string "incdec")
```

```
   ]
   (const_string "alu")))
   (set_attr "mode" "SI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand 0 "register_operand" "")
(plus (match_operand 1 "register_operand" "")
              (match_operand 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])"
  [(const_int 0)]
{
  rtx pat;
  /* In -fPIC mode the constructs like (const (unspec [symbol_ref]))
     may confuse gen_lowpart.  */
  if (GET_MODE (operands[0]) != Pmode)
    {
      operands[1] = gen_lowpart (Pmode, operands[1]);
      operands[2] = gen_lowpart (Pmode, operands[2]);
    }
  operands[0] = gen_lowpart (SImode, operands[0]);
  pat = gen_rtx_PLUS (Pmode, operands[1], operands[2]);
  if (Pmode != SImode)
    pat = gen_rtx_SUBREG (SImode, pat, 0);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], pat));
  DONE;
})

;; It may seem that nonimmediate operand is proper one for operand 1.
;; The addsi_1 pattern allows nonimmediate operand at that place and
;; we take care in ix86_binary_operator_ok to not allow two memory
;; operands so proper swapping will be done in reload.  This allow
;; patterns constructed from addsi_1 to match.
(define_insn "addsi_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI
  (plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0,r")
   (match_operand:SI 2 "general_operand" "rmni,lni"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      operands[2] = SET_SRC (XVECEXP (PATTERN (insn), 0, 0));
      return "lea{l}\t{%a2, %k0|%k0, %a2}";

    case TYPE_INCDEC:
```

```
      if (operands[2] == const1_rtx)
        return "inc{l}\t%k0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%k0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %k0|%k0, %2}";
        }
      return "add{l}\t{%2, %k0|%k0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "1")
      (const_string "lea")
    ; Current assemblers are broken and do not allow @GOTOFF in
    ; ought but a memory context.
    (match_operand:SI 2 "pic_symbolic_operand" "")
      (const_string "lea")
    (match_operand:SI 2 "incdec_operand" "")
      (const_string "incdec")
   ]
   (const_string "alu")))
   (set_attr "mode" "SI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(zero_extend:DI
  (plus:SI (match_operand:SI 1 "register_operand" "")
   (match_operand:SI 2 "nonmemory_operand" ""))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])"
  [(set (match_dup 0)
(zero_extend:DI (subreg:SI (plus:DI (match_dup 1) (match_dup 2)) 0)))]
{
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[2] = gen_lowpart (Pmode, operands[2]);
})
```

```
(define_insn "*addsi_2"
  [(set (reg FLAGS_REG)
(compare
  (plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
   (match_operand:SI 2 "general_operand" "rmni,rni"))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=r,rm")
(plus:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (PLUS, SImode, operands)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{l}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %0|%0, %2}";
        }
      return "add{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
(define_insn "*addsi_2_zext"
```

```
  [(set (reg FLAGS_REG)
(compare
  (plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
   (match_operand:SI 2 "general_operand" "rmni"))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (plus:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (PLUS, SImode, operands)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
        return "inc{l}\t%k0";
      else
{
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%k0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %k0|%k0, %2}";
        }
      return "add{l}\t{%2, %k0|%k0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

(define_insn "*addsi_3"
  [(set (reg FLAGS_REG)
(compare (neg:SI (match_operand:SI 2 "general_operand" "rmni"))
 (match_operand:SI 1 "nonimmediate_operand" "%0")))
   (clobber (match_scratch:SI 0 "=r"))]
  "ix86_match_ccmode (insn, CCZmode)
```

```
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{l}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %0|%0, %2}";
        }
      return "add{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
(define_insn "*addsi_3_zext"
  [(set (reg FLAGS_REG)
(compare (neg:SI (match_operand:SI 2 "general_operand" "rmni"))
 (match_operand:SI 1 "nonimmediate_operand" "%0")))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (plus:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCZmode)
   && ix86_binary_operator_ok (PLUS, SImode, operands)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
```

```
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
        return "inc{l}\t%k0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%k0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %k0|%k0, %2}";
        }
      return "add{l}\t{%2, %k0|%k0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

; For comparisons against 1, -1 and 128, we may generate better code
; by converting cmp to add, inc or dec as done by peephole2.  This pattern
; is matched then.  We can't accept general immediate, because for
; case of overflows,  the result is messed up.
; This pattern also don't hold of 0x80000000, since the value overflows
; when negated.
; Also carry flag is reversed compared to cmp, so this conversion is valid
; only for comparisons not depending on it.
(define_insn "*addsi_4"
  [(set (reg FLAGS_REG)
(compare (match_operand:SI 1 "nonimmediate_operand" "0")
 (match_operand:SI 2 "const_int_operand" "n")))
   (clobber (match_scratch:SI 0 "=rm"))]
  "ix86_match_ccmode (insn, CCGCmode)
   && (INTVAL (operands[2]) & 0xffffffff) != 0x80000000"
{
  switch (get_attr_type (insn))
    {
```

```
    case TYPE_INCDEC:
      if (operands[2] == constm1_rtx)
        return "inc{l}\t%0";
      else
        {
  gcc_assert (operands[2] == const1_rtx);
          return "dec{l}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if ((INTVAL (operands[2]) == -128
   || (INTVAL (operands[2]) > 0
       && INTVAL (operands[2]) != 128)))
return "sub{l}\t{%2, %0|%0, %2}";
      operands[2] = GEN_INT (-INTVAL (operands[2]));
      return "add{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

(define_insn "*addsi_5"
  [(set (reg FLAGS_REG)
(compare
  (plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
   (match_operand:SI 2 "general_operand" "rmni"))
  (const_int 0)))
   (clobber (match_scratch:SI 0 "=r"))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)
   /* Current assemblers are broken and do not allow @GOTOFF in
      ought but a memory context.  */
   && ! pic_symbolic_operand (operands[2], VOIDmode)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (operands[2] == const1_rtx)
        return "inc{l}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
          return "dec{l}\t%0";
```

```
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "sub{l}\t{%2, %0|%0, %2}";
        }
      return "add{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:SI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])

(define_expand "addhi3"
  [(parallel [(set (match_operand:HI 0 "nonimmediate_operand" "")
   (plus:HI (match_operand:HI 1 "nonimmediate_operand" "")
    (match_operand:HI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (PLUS, HImode, operands); DONE;")

;; %%% After Dave's SUBREG_BYTE stuff goes in, re-enable incb %ah
;; type optimizations enabled by define-splits.  This is not important
;; for PII, and in fact harmful because of partial register stalls.

(define_insn "*addhi_1_lea"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r,r")
(plus:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0,r")
 (match_operand:HI 2 "general_operand" "ri,rm,lni")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (PLUS, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      return "#";
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{w}\t%0";
```

```
      else
{
  gcc_assert (operands[2] == constm1_rtx);
  return "dec{w}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{w}\t{%2, %0|%0, %2}";
}
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (eq_attr "alternative" "2")
(const_string "lea")
(if_then_else (match_operand:HI 2 "incdec_operand" "")
   (const_string "incdec")
   (const_string "alu"))))
   (set_attr "mode" "HI,HI,SI")])

(define_insn "*addhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
(plus:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
 (match_operand:HI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (PLUS, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{w}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
  return "dec{w}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
```

```
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{w}\t{%2, %0|%0, %2}";
}
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "HI")])

(define_insn "*addhi_2"
  [(set (reg FLAGS_REG)
(compare
  (plus:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
   (match_operand:HI 2 "general_operand" "rmni,rni"))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=r,rm")
(plus:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (PLUS, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{w}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
  return "dec{w}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{w}\t{%2, %0|%0, %2}";
}
```

```
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "HI")])

(define_insn "*addhi_3"
  [(set (reg FLAGS_REG)
(compare (neg:HI (match_operand:HI 2 "general_operand" "rmni"))
 (match_operand:HI 1 "nonimmediate_operand" "%0")))
   (clobber (match_scratch:HI 0 "=r"))]
  "ix86_match_ccmode (insn, CCZmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{w}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx);
  return "dec{w}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{w}\t{%2, %0|%0, %2}";
}
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "HI")])

; See comments above addsi_4 for details.
(define_insn "*addhi_4"
```

```
  [(set (reg FLAGS_REG)
(compare (match_operand:HI 1 "nonimmediate_operand" "0")
 (match_operand:HI 2 "const_int_operand" "n")))
   (clobber (match_scratch:HI 0 "=rm"))]
  "ix86_match_ccmode (insn, CCGCmode)
   && (INTVAL (operands[2]) & 0xffff) != 0x8000"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == constm1_rtx)
        return "inc{w}\t%0";
      else
{
  gcc_assert (operands[2] == const1_rtx);
          return "dec{w}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if ((INTVAL (operands[2]) == -128
   || (INTVAL (operands[2]) > 0
       && INTVAL (operands[2]) != 128)))
return "sub{w}\t{%2, %0|%0, %2}";
      operands[2] = GEN_INT (-INTVAL (operands[2]));
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "SI")])


(define_insn "*addhi_5"
  [(set (reg FLAGS_REG)
(compare
  (plus:HI (match_operand:HI 1 "nonimmediate_operand" "%0")
   (match_operand:HI 2 "general_operand" "rmni"))
  (const_int 0)))
   (clobber (match_scratch:HI 0 "=r"))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
```

```
      if (operands[2] == const1_rtx)
return "inc{w}\t%0";
      else
{
  gcc_assert (operands[2] == constm1_rtx);
  return "dec{w}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{w}\t{%2, %0|%0, %2}";
}
      return "add{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "HI")])

(define_expand "addqi3"
  [(parallel [(set (match_operand:QI 0 "nonimmediate_operand" "")
   (plus:QI (match_operand:QI 1 "nonimmediate_operand" "")
    (match_operand:QI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (PLUS, QImode, operands); DONE;")

;; %%% Potential partial reg stall on alternative 2.  What to do?
(define_insn "*addqi_1_lea"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q,r,r")
(plus:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0,0,r")
 (match_operand:QI 2 "general_operand" "qn,qmn,rn,ln")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (PLUS, QImode, operands)"
{
  int widen = (which_alternative == 2);
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      return "#";
```

```
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return widen ? "inc{l}\t%k0" : "inc{b}\t%0";
      else
{
  gcc_assert (operands[2] == constm1_rtx);
  return widen ? "dec{l}\t%k0" : "dec{b}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  if (widen)
    return "sub{l}\t{%2, %k0|%k0, %2}";
  else
    return "sub{b}\t{%2, %0|%0, %2}";
}
      if (widen)
        return "add{l}\t{%k2, %k0|%k0, %k2}";
      else
        return "add{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (eq_attr "alternative" "3")
(const_string "lea")
(if_then_else (match_operand:QI 2 "incdec_operand" "")
   (const_string "incdec")
   (const_string "alu"))))
   (set_attr "mode" "QI,QI,SI,SI")])

(define_insn "*addqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q,r")
(plus:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0,0")
 (match_operand:QI 2 "general_operand" "qn,qmn,rn")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (PLUS, QImode, operands)"
{
  int widen = (which_alternative == 2);
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
```

```
return widen ? "inc{l}\t%k0" : "inc{b}\t%0";
      else
{
  gcc_assert (operands[2] == constm1_rtx);
  return widen ? "dec{l}\t%k0" : "dec{b}\t%0";
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.
 Exceptions: -128 encodes smaller than 128, so swap sign and op.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
  && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  if (widen)
    return "sub{l}\t{%2, %k0|%k0, %2}";
  else
    return "sub{b}\t{%2, %0|%0, %2}";
}
      if (widen)
        return "add{l}\t{%k2, %k0|%k0, %k2}";
      else
        return "add{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI,QI,SI")])

(define_insn "*addqi_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,q"))
(plus:QI (match_dup 0)
 (match_operand:QI 1 "general_operand" "qn,qnm")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[1] == const1_rtx)
return "inc{b}\t%0";
      else
{
  gcc_assert (operands[1] == constm1_rtx);
  return "dec{b}\t%0";
```

```
}

    default:
      /* Make things pretty and `subl $4,%eax' rather than `addl $-4, %eax'.  */
      if (GET_CODE (operands[1]) == CONST_INT
  && INTVAL (operands[1]) < 0)
{
  operands[1] = GEN_INT (-INTVAL (operands[1]));
  return "sub{b}\t{%1, %0|%0, %1}";
}
      return "add{b}\t{%1, %0|%0, %1}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 1 "incdec_operand" "")
(const_string "incdec")
(const_string "alu1")))
   (set (attr "memory")
     (if_then_else (match_operand 1 "memory_operand" "")
        (const_string "load")
        (const_string "none")))
   (set_attr "mode" "QI")])

(define_insn "*addqi_2"
  [(set (reg FLAGS_REG)
(compare
  (plus:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0")
   (match_operand:QI 2 "general_operand" "qmni,qni"))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=q,qm")
(plus:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (PLUS, QImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{b}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx
      || (GET_CODE (operands[2]) == CONST_INT
          && INTVAL (operands[2]) == 255));
  return "dec{b}\t%0";
}

    default:
      /* Make things pretty and `subb $4,%al' rather than `addb $-4, %al'.  */
      if (GET_CODE (operands[2]) == CONST_INT
```

```
          && INTVAL (operands[2]) < 0)
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{b}\t{%2, %0|%0, %2}";
}
      return "add{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI")])

(define_insn "*addqi_3"
  [(set (reg FLAGS_REG)
(compare (neg:QI (match_operand:QI 2 "general_operand" "qmni"))
 (match_operand:QI 1 "nonimmediate_operand" "%0")))
   (clobber (match_scratch:QI 0 "=q"))]
  "ix86_match_ccmode (insn, CCZmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{b}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx
      || (GET_CODE (operands[2]) == CONST_INT
  && INTVAL (operands[2]) == 255));
  return "dec{b}\t%0";
}

    default:
      /* Make things pretty and `subb $4,%al' rather than `addb $-4, %al'.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && INTVAL (operands[2]) < 0)
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{b}\t{%2, %0|%0, %2}";
}
      return "add{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
```

```
   (set_attr "mode" "QI")])

; See comments above addsi_4 for details.
(define_insn "*addqi_4"
  [(set (reg FLAGS_REG)
(compare (match_operand:QI 1 "nonimmediate_operand" "0")
 (match_operand:QI 2 "const_int_operand" "n")))
   (clobber (match_scratch:QI 0 "=qm"))]
  "ix86_match_ccmode (insn, CCGCmode)
   && (INTVAL (operands[2]) & 0xff) != 0x80"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == constm1_rtx
  || (GET_CODE (operands[2]) == CONST_INT
      && INTVAL (operands[2]) == 255))
        return "inc{b}\t%0";
      else
{
  gcc_assert (operands[2] == const1_rtx);
          return "dec{b}\t%0";
}

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (INTVAL (operands[2]) < 0)
        {
          operands[2] = GEN_INT (-INTVAL (operands[2]));
          return "add{b}\t{%2, %0|%0, %2}";
        }
      return "sub{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:HI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI")])


(define_insn "*addqi_5"
  [(set (reg FLAGS_REG)
(compare
  (plus:QI (match_operand:QI 1 "nonimmediate_operand" "%0")
   (match_operand:QI 2 "general_operand" "qmni"))
  (const_int 0)))
   (clobber (match_scratch:QI 0 "=q"))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
```

```
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{b}\t%0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx
      || (GET_CODE (operands[2]) == CONST_INT
  && INTVAL (operands[2]) == 255));
  return "dec{b}\t%0";
}

    default:
      /* Make things pretty and `subb $4,%al' rather than `addb $-4, %al'.  */
      if (GET_CODE (operands[2]) == CONST_INT
          && INTVAL (operands[2]) < 0)
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{b}\t{%2, %0|%0, %2}";
}
      return "add{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI")])


(define_insn "addqi_ext_1"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(plus:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (match_operand:QI 2 "general_operand" "Qmn")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{b}\t%h0";
```

```
      else
        {
  gcc_assert (operands[2] == constm1_rtx
      || (GET_CODE (operands[2]) == CONST_INT
  && INTVAL (operands[2]) == 255));
          return "dec{b}\t%h0";
}

    default:
      return "add{b}\t{%2, %h0|%h0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI")])

(define_insn "*addqi_ext_1_rex64"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(plus:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (match_operand:QI 2 "nonmemory_operand" "Qn")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_INCDEC:
      if (operands[2] == const1_rtx)
return "inc{b}\t%h0";
      else
        {
  gcc_assert (operands[2] == constm1_rtx
      || (GET_CODE (operands[2]) == CONST_INT
  && INTVAL (operands[2]) == 255));
          return "dec{b}\t%h0";
        }

    default:
      return "add{b}\t{%2, %h0|%h0, %2}";
    }
}
  [(set (attr "type")
     (if_then_else (match_operand:QI 2 "incdec_operand" "")
```

```
(const_string "incdec")
(const_string "alu")))
   (set_attr "mode" "QI")])

(define_insn "*addqi_ext_2"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(plus:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "%0")
    (const_int 8)
    (const_int 8))
  (zero_extract:SI
    (match_operand 2 "ext_register_operand" "Q")
    (const_int 8)
    (const_int 8))))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "add{b}\t{%h2, %h0|%h0, %h2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

;; The patterns that match these are at the end of this file.

(define_expand "addxf3"
  [(set (match_operand:XF 0 "register_operand" "")
(plus:XF (match_operand:XF 1 "register_operand" "")
 (match_operand:XF 2 "register_operand" "")))]
  "TARGET_80387"
  "")

(define_expand "adddf3"
  [(set (match_operand:DF 0 "register_operand" "")
(plus:DF (match_operand:DF 1 "register_operand" "")
 (match_operand:DF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "")

(define_expand "addsf3"
  [(set (match_operand:SF 0 "register_operand" "")
(plus:SF (match_operand:SF 1 "register_operand" "")
 (match_operand:SF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "")

;; Subtract instructions

;; %%% splits for subditi3
```

```
(define_expand "subti3"
  [(parallel [(set (match_operand:TI 0 "nonimmediate_operand" "")
   (minus:TI (match_operand:TI 1 "nonimmediate_operand" "")
     (match_operand:TI 2 "x86_64_general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (MINUS, TImode, operands); DONE;")

(define_insn "*subti3_1"
  [(set (match_operand:TI 0 "nonimmediate_operand" "=r,o")
(minus:TI (match_operand:TI 1 "nonimmediate_operand" "0,0")
  (match_operand:TI 2 "general_operand" "roiF,riF")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (MINUS, TImode, operands)"
  "#")

(define_split
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
(minus:TI (match_operand:TI 1 "nonimmediate_operand" "")
  (match_operand:TI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(parallel [(set (reg:CC FLAGS_REG) (compare:CC (match_dup 1) (match_dup 2)))
      (set (match_dup 0) (minus:DI (match_dup 1) (match_dup 2)))])
   (parallel [(set (match_dup 3)
   (minus:DI (match_dup 4)
     (plus:DI (ltu:DI (reg:CC FLAGS_REG) (const_int 0))
      (match_dup 5))))
      (clobber (reg:CC FLAGS_REG))])]
  "split_ti (operands+0, 1, operands+0, operands+3);
   split_ti (operands+1, 1, operands+1, operands+4);
   split_ti (operands+2, 1, operands+2, operands+5);")

;; %%% splits for subsidi3

(define_expand "subdi3"
  [(parallel [(set (match_operand:DI 0 "nonimmediate_operand" "")
   (minus:DI (match_operand:DI 1 "nonimmediate_operand" "")
     (match_operand:DI 2 "x86_64_general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_binary_operator (MINUS, DImode, operands); DONE;")

(define_insn "*subdi3_1"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,o")
(minus:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
  (match_operand:DI 2 "general_operand" "roiF,riF")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && ix86_binary_operator_ok (MINUS, DImode, operands)"
  "#")
```

```
(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(minus:DI (match_operand:DI 1 "nonimmediate_operand" "")
  (match_operand:DI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && reload_completed"
  [(parallel [(set (reg:CC FLAGS_REG) (compare:CC (match_dup 1) (match_dup 2)))
      (set (match_dup 0) (minus:SI (match_dup 1) (match_dup 2)))])
   (parallel [(set (match_dup 3)
   (minus:SI (match_dup 4)
     (plus:SI (ltu:SI (reg:CC FLAGS_REG) (const_int 0))
      (match_dup 5))))
      (clobber (reg:CC FLAGS_REG))])]
  "split_di (operands+0, 1, operands+0, operands+3);
   split_di (operands+1, 1, operands+1, operands+4);
   split_di (operands+2, 1, operands+2, operands+5);")

(define_insn "subdi3_carry_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
  (minus:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
    (plus:DI (match_operand:DI 3 "ix86_carry_flag_operator" "")
       (match_operand:DI 2 "x86_64_general_operand" "re,rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (MINUS, DImode, operands)"
  "sbb{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "DI")])

(define_insn "*subdi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(minus:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
  (match_operand:DI 2 "x86_64_general_operand" "re,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (MINUS, DImode, operands)"
  "sub{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "*subdi_2_rex64"
  [(set (reg FLAGS_REG)
(compare
  (minus:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
    (match_operand:DI 2 "x86_64_general_operand" "re,rm"))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(minus:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (MINUS, DImode, operands)"
```

```
  "sub{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "*subdi_3_rex63"
  [(set (reg FLAGS_REG)
(compare (match_operand:DI 1 "nonimmediate_operand" "0,0")
 (match_operand:DI 2 "x86_64_general_operand" "re,rm")))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(minus:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "subqi3_carry"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q")
  (minus:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
    (plus:QI (match_operand:QI 3 "ix86_carry_flag_operator" "")
       (match_operand:QI 2 "general_operand" "qi,qm"))))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, QImode, operands)"
  "sbb{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "QI")])

(define_insn "subhi3_carry"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
  (minus:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
    (plus:HI (match_operand:HI 3 "ix86_carry_flag_operator" "")
       (match_operand:HI 2 "general_operand" "ri,rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, HImode, operands)"
  "sbb{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "HI")])

(define_insn "subsi3_carry"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
  (minus:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
    (plus:SI (match_operand:SI 3 "ix86_carry_flag_operator" "")
       (match_operand:SI 2 "general_operand" "ri,rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sbb{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
```

```
   (set_attr "mode" "SI")])

(define_insn "subsi3_carry_zext"
  [(set (match_operand:DI 0 "register_operand" "=rm,r")
  (zero_extend:DI
    (minus:SI (match_operand:SI 1 "register_operand" "0,0")
      (plus:SI (match_operand:SI 3 "ix86_carry_flag_operator" "")
 (match_operand:SI 2 "general_operand" "ri,rm")))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sbb{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "SI")])

(define_expand "subsi3"
  [(parallel [(set (match_operand:SI 0 "nonimmediate_operand" "")
   (minus:SI (match_operand:SI 1 "nonimmediate_operand" "")
     (match_operand:SI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_binary_operator (MINUS, SImode, operands); DONE;")

(define_insn "*subsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(minus:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
  (match_operand:SI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*subsi_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (minus:SI (match_operand:SI 1 "register_operand" "0")
    (match_operand:SI 2 "general_operand" "rim"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*subsi_2"
  [(set (reg FLAGS_REG)
(compare
  (minus:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
    (match_operand:SI 2 "general_operand" "ri,rm"))
  (const_int 0)))
```

```
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(minus:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*subsi_2_zext"
  [(set (reg FLAGS_REG)
(compare
  (minus:SI (match_operand:SI 1 "register_operand" "0")
    (match_operand:SI 2 "general_operand" "rim"))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (minus:SI (match_dup 1)
    (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*subsi_3"
  [(set (reg FLAGS_REG)
(compare (match_operand:SI 1 "nonimmediate_operand" "0,0")
 (match_operand:SI 2 "general_operand" "ri,rm")))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(minus:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*subsi_3_zext"
  [(set (reg FLAGS_REG)
(compare (match_operand:SI 1 "register_operand" "0")
 (match_operand:SI 2 "general_operand" "rim")))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (minus:SI (match_dup 1)
    (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])
```

```
(define_expand "subhi3"
  [(parallel [(set (match_operand:HI 0 "nonimmediate_operand" "")
   (minus:HI (match_operand:HI 1 "nonimmediate_operand" "")
     (match_operand:HI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (MINUS, HImode, operands); DONE;")

(define_insn "*subhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
(minus:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
  (match_operand:HI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, HImode, operands)"
  "sub{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*subhi_2"
  [(set (reg FLAGS_REG)
(compare
  (minus:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
    (match_operand:HI 2 "general_operand" "ri,rm"))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
(minus:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (MINUS, HImode, operands)"
  "sub{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*subhi_3"
  [(set (reg FLAGS_REG)
(compare (match_operand:HI 1 "nonimmediate_operand" "0,0")
 (match_operand:HI 2 "general_operand" "ri,rm")))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
(minus:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, HImode, operands)"
  "sub{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_expand "subqi3"
  [(parallel [(set (match_operand:QI 0 "nonimmediate_operand" "")
   (minus:QI (match_operand:QI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
```

```
  "ix86_expand_binary_operator (MINUS, QImode, operands); DONE;")

(define_insn "*subqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q")
(minus:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
  (match_operand:QI 2 "general_operand" "qn,qmn")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (MINUS, QImode, operands)"
  "sub{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*subqi_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,q"))
(minus:QI (match_dup 0)
  (match_operand:QI 1 "general_operand" "qn,qmn")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "sub{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "*subqi_2"
  [(set (reg FLAGS_REG)
(compare
  (minus:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
    (match_operand:QI 2 "general_operand" "qi,qm"))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=qm,q")
(minus:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (MINUS, QImode, operands)"
  "sub{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*subqi_3"
  [(set (reg FLAGS_REG)
(compare (match_operand:QI 1 "nonimmediate_operand" "0,0")
 (match_operand:QI 2 "general_operand" "qi,qm")))
   (set (match_operand:HI 0 "nonimmediate_operand" "=qm,q")
(minus:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, QImode, operands)"
  "sub{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

;; The patterns that match these are at the end of this file.
```

```
(define_expand "subxf3"
  [(set (match_operand:XF 0 "register_operand" "")
(minus:XF (match_operand:XF 1 "register_operand" "")
  (match_operand:XF 2 "register_operand" "")))]
  "TARGET_80387"
  "")

(define_expand "subdf3"
  [(set (match_operand:DF 0 "register_operand" "")
(minus:DF (match_operand:DF 1 "register_operand" "")
  (match_operand:DF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "")

(define_expand "subsf3"
  [(set (match_operand:SF 0 "register_operand" "")
(minus:SF (match_operand:SF 1 "register_operand" "")
  (match_operand:SF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "")

;; Multiply instructions

(define_expand "muldi3"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (match_operand:DI 1 "register_operand" "")
    (match_operand:DI 2 "x86_64_general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*muldi3_1_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r,r,r")
(mult:DI (match_operand:DI 1 "nonimmediate_operand" "%rm,rm,0")
 (match_operand:DI 2 "x86_64_general_operand" "K,e,mr")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "@
   imul{q}\t{%2, %1, %0|%0, %1, %2}
   imul{q}\t{%2, %1, %0|%0, %1, %2}
   imul{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "imul")
   (set_attr "prefix_0f" "0,0,1")
   (set (attr "athlon_decode")
(cond [(eq_attr "cpu" "athlon")
  (const_string "vector")
       (eq_attr "alternative" "1")
  (const_string "vector")
```

```
       (and (eq_attr "alternative" "2")
    (match_operand 1 "memory_operand" ""))
  (const_string "vector")]
      (const_string "direct")))
   (set_attr "mode" "DI")])

(define_expand "mulsi3"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (mult:SI (match_operand:SI 1 "register_operand" "")
    (match_operand:SI 2 "general_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*mulsi3_1"
  [(set (match_operand:SI 0 "register_operand" "=r,r,r")
(mult:SI (match_operand:SI 1 "nonimmediate_operand" "%rm,rm,0")
 (match_operand:SI 2 "general_operand" "K,i,mr")))
   (clobber (reg:CC FLAGS_REG))]
  "GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM"
  "@
   imul{l}\t{%2, %1, %0|%0, %1, %2}
   imul{l}\t{%2, %1, %0|%0, %1, %2}
   imul{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "imul")
   (set_attr "prefix_0f" "0,0,1")
   (set (attr "athlon_decode")
(cond [(eq_attr "cpu" "athlon")
  (const_string "vector")
       (eq_attr "alternative" "1")
  (const_string "vector")
       (and (eq_attr "alternative" "2")
    (match_operand 1 "memory_operand" ""))
  (const_string "vector")]
      (const_string "direct")))
   (set_attr "mode" "SI")])

(define_insn "*mulsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r,r")
(zero_extend:DI
  (mult:SI (match_operand:SI 1 "nonimmediate_operand" "%rm,rm,0")
   (match_operand:SI 2 "general_operand" "K,i,mr"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "@
   imul{l}\t{%2, %1, %k0|%k0, %1, %2}
   imul{l}\t{%2, %1, %k0|%k0, %1, %2}
   imul{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "imul")
```

```
   (set_attr "prefix_0f" "0,0,1")
   (set (attr "athlon_decode")
(cond [(eq_attr "cpu" "athlon")
  (const_string "vector")
       (eq_attr "alternative" "1")
  (const_string "vector")
       (and (eq_attr "alternative" "2")
    (match_operand 1 "memory_operand" ""))
  (const_string "vector")]
      (const_string "direct")))
   (set_attr "mode" "SI")])

(define_expand "mulhi3"
  [(parallel [(set (match_operand:HI 0 "register_operand" "")
   (mult:HI (match_operand:HI 1 "register_operand" "")
    (match_operand:HI 2 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_HIMODE_MATH"
  "")

(define_insn "*mulhi3_1"
  [(set (match_operand:HI 0 "register_operand" "=r,r,r")
(mult:HI (match_operand:HI 1 "nonimmediate_operand" "%rm,rm,0")
 (match_operand:HI 2 "general_operand" "K,i,mr")))
   (clobber (reg:CC FLAGS_REG))]
  "GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM"
  "@
   imul{w}\t{%2, %1, %0|%0, %1, %2}
   imul{w}\t{%2, %1, %0|%0, %1, %2}
   imul{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "imul")
   (set_attr "prefix_0f" "0,0,1")
   (set (attr "athlon_decode")
(cond [(eq_attr "cpu" "athlon")
  (const_string "vector")
       (eq_attr "alternative" "1,2")
  (const_string "vector")]
      (const_string "direct")))
   (set_attr "mode" "HI")])

(define_expand "mulqi3"
  [(parallel [(set (match_operand:QI 0 "register_operand" "")
   (mult:QI (match_operand:QI 1 "nonimmediate_operand" "")
    (match_operand:QI 2 "register_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
  "")

(define_insn "*mulqi3_1"
  [(set (match_operand:QI 0 "register_operand" "=a")
```

```
(mult:QI (match_operand:QI 1 "nonimmediate_operand" "%0")
 (match_operand:QI 2 "nonimmediate_operand" "qm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{b}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "direct")))
   (set_attr "mode" "QI")])

(define_expand "umulqihi3"
  [(parallel [(set (match_operand:HI 0 "register_operand" "")
   (mult:HI (zero_extend:HI
      (match_operand:QI 1 "nonimmediate_operand" ""))
    (zero_extend:HI
      (match_operand:QI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
  "")

(define_insn "*umulqihi3_1"
  [(set (match_operand:HI 0 "register_operand" "=a")
(mult:HI (zero_extend:HI (match_operand:QI 1 "nonimmediate_operand" "%0"))
 (zero_extend:HI (match_operand:QI 2 "nonimmediate_operand" "qm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{b}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "direct")))
   (set_attr "mode" "QI")])

(define_expand "mulqihi3"
  [(parallel [(set (match_operand:HI 0 "register_operand" "")
   (mult:HI (sign_extend:HI (match_operand:QI 1 "nonimmediate_operand" ""))
    (sign_extend:HI (match_operand:QI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
  "")

(define_insn "*mulqihi3_insn"
  [(set (match_operand:HI 0 "register_operand" "=a")
```

```
(mult:HI (sign_extend:HI (match_operand:QI 1 "nonimmediate_operand" "%0"))
 (sign_extend:HI (match_operand:QI 2 "nonimmediate_operand" "qm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "imul{b}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "direct")))
   (set_attr "mode" "QI")])

(define_expand "umulditi3"
  [(parallel [(set (match_operand:TI 0 "register_operand" "")
   (mult:TI (zero_extend:TI
      (match_operand:DI 1 "nonimmediate_operand" ""))
    (zero_extend:TI
      (match_operand:DI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*umulditi3_insn"
  [(set (match_operand:TI 0 "register_operand" "=A")
(mult:TI (zero_extend:TI (match_operand:DI 1 "nonimmediate_operand" "%0"))
 (zero_extend:TI (match_operand:DI 2 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{q}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "DI")])

;; We can't use this pattern in 64bit mode, since it results in two separate 32bit registers
(define_expand "umulsidi3"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (zero_extend:DI
      (match_operand:SI 1 "nonimmediate_operand" ""))
    (zero_extend:DI
      (match_operand:SI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "!TARGET_64BIT"
  "")
```

```
(define_insn "*umulsidi3_insn"
  [(set (match_operand:DI 0 "register_operand" "=A")
(mult:DI (zero_extend:DI (match_operand:SI 1 "nonimmediate_operand" "%0"))
 (zero_extend:DI (match_operand:SI 2 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{l}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "SI")])

(define_expand "mulditi3"
  [(parallel [(set (match_operand:TI 0 "register_operand" "")
   (mult:TI (sign_extend:TI
      (match_operand:DI 1 "nonimmediate_operand" ""))
    (sign_extend:TI
      (match_operand:DI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*mulditi3_insn"
  [(set (match_operand:TI 0 "register_operand" "=A")
(mult:TI (sign_extend:TI (match_operand:DI 1 "nonimmediate_operand" "%0"))
 (sign_extend:TI (match_operand:DI 2 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "imul{q}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "DI")])

(define_expand "mulsidi3"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (sign_extend:DI
      (match_operand:SI 1 "nonimmediate_operand" ""))
    (sign_extend:DI
      (match_operand:SI 2 "register_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
```

```
  "!TARGET_64BIT"
  "")

(define_insn "*mulsidi3_insn"
  [(set (match_operand:DI 0 "register_operand" "=A")
(mult:DI (sign_extend:DI (match_operand:SI 1 "nonimmediate_operand" "%0"))
 (sign_extend:DI (match_operand:SI 2 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "imul{l}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "SI")])

(define_expand "umuldi3_highpart"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (truncate:DI
     (lshiftrt:TI
       (mult:TI (zero_extend:TI
  (match_operand:DI 1 "nonimmediate_operand" ""))
(zero_extend:TI
  (match_operand:DI 2 "register_operand" "")))
       (const_int 64))))
      (clobber (match_scratch:DI 3 ""))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*umuldi3_highpart_rex64"
  [(set (match_operand:DI 0 "register_operand" "=d")
(truncate:DI
  (lshiftrt:TI
    (mult:TI (zero_extend:TI
       (match_operand:DI 1 "nonimmediate_operand" "%a"))
     (zero_extend:TI
       (match_operand:DI 2 "nonimmediate_operand" "rm")))
    (const_int 64))))
   (clobber (match_scratch:DI 3 "=1"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{q}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
```

```
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "DI")])

(define_expand "umulsi3_highpart"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (truncate:SI
     (lshiftrt:DI
       (mult:DI (zero_extend:DI
  (match_operand:SI 1 "nonimmediate_operand" ""))
(zero_extend:DI
  (match_operand:SI 2 "register_operand" "")))
       (const_int 32))))
      (clobber (match_scratch:SI 3 ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*umulsi3_highpart_insn"
  [(set (match_operand:SI 0 "register_operand" "=d")
(truncate:SI
  (lshiftrt:DI
    (mult:DI (zero_extend:DI
       (match_operand:SI 1 "nonimmediate_operand" "%a"))
     (zero_extend:DI
       (match_operand:SI 2 "nonimmediate_operand" "rm")))
    (const_int 32))))
   (clobber (match_scratch:SI 3 "=1"))
   (clobber (reg:CC FLAGS_REG))]
  "GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM"
  "mul{l}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "SI")])

(define_insn "*umulsi3_highpart_zext"
  [(set (match_operand:DI 0 "register_operand" "=d")
(zero_extend:DI (truncate:SI
  (lshiftrt:DI
    (mult:DI (zero_extend:DI
       (match_operand:SI 1 "nonimmediate_operand" "%a"))
     (zero_extend:DI
       (match_operand:SI 2 "nonimmediate_operand" "rm")))
    (const_int 32)))))
   (clobber (match_scratch:SI 3 "=1"))
```

```
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "mul{l}\t%2"
  [(set_attr "type" "imul")
   (set_attr "length_immediate" "0")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "SI")])

(define_expand "smuldi3_highpart"
  [(parallel [(set (match_operand:DI 0 "register_operand" "=d")
   (truncate:DI
     (lshiftrt:TI
       (mult:TI (sign_extend:TI
  (match_operand:DI 1 "nonimmediate_operand" ""))
(sign_extend:TI
  (match_operand:DI 2 "register_operand" "")))
       (const_int 64))))
      (clobber (match_scratch:DI 3 ""))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*smuldi3_highpart_rex64"
  [(set (match_operand:DI 0 "register_operand" "=d")
(truncate:DI
  (lshiftrt:TI
    (mult:TI (sign_extend:TI
       (match_operand:DI 1 "nonimmediate_operand" "%a"))
     (sign_extend:TI
       (match_operand:DI 2 "nonimmediate_operand" "rm")))
    (const_int 64))))
   (clobber (match_scratch:DI 3 "=1"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "imul{q}\t%2"
  [(set_attr "type" "imul")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "DI")])

(define_expand "smulsi3_highpart"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (truncate:SI
```

```
     (lshiftrt:DI
       (mult:DI (sign_extend:DI
  (match_operand:SI 1 "nonimmediate_operand" ""))
(sign_extend:DI
  (match_operand:SI 2 "register_operand" "")))
       (const_int 32))))
      (clobber (match_scratch:SI 3 ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*smulsi3_highpart_insn"
  [(set (match_operand:SI 0 "register_operand" "=d")
(truncate:SI
  (lshiftrt:DI
    (mult:DI (sign_extend:DI
       (match_operand:SI 1 "nonimmediate_operand" "%a"))
     (sign_extend:DI
       (match_operand:SI 2 "nonimmediate_operand" "rm")))
    (const_int 32))))
   (clobber (match_scratch:SI 3 "=1"))
   (clobber (reg:CC FLAGS_REG))]
  "GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM"
  "imul{l}\t%2"
  [(set_attr "type" "imul")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
   (set_attr "mode" "SI")])

(define_insn "*smulsi3_highpart_zext"
  [(set (match_operand:DI 0 "register_operand" "=d")
(zero_extend:DI (truncate:SI
  (lshiftrt:DI
    (mult:DI (sign_extend:DI
       (match_operand:SI 1 "nonimmediate_operand" "%a"))
     (sign_extend:DI
       (match_operand:SI 2 "nonimmediate_operand" "rm")))
    (const_int 32)))))
   (clobber (match_scratch:SI 3 "=1"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "imul{l}\t%2"
  [(set_attr "type" "imul")
   (set (attr "athlon_decode")
     (if_then_else (eq_attr "cpu" "athlon")
        (const_string "vector")
        (const_string "double")))
```

```
   (set_attr "mode" "SI")])

;; The patterns that match these are at the end of this file.

(define_expand "mulxf3"
  [(set (match_operand:XF 0 "register_operand" "")
(mult:XF (match_operand:XF 1 "register_operand" "")
 (match_operand:XF 2 "register_operand" "")))]
  "TARGET_80387"
  "")

(define_expand "muldf3"
  [(set (match_operand:DF 0 "register_operand" "")
(mult:DF (match_operand:DF 1 "register_operand" "")
 (match_operand:DF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "")

(define_expand "mulsf3"
  [(set (match_operand:SF 0 "register_operand" "")
(mult:SF (match_operand:SF 1 "register_operand" "")
 (match_operand:SF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "")

;; Divide instructions

(define_insn "divqi3"
  [(set (match_operand:QI 0 "register_operand" "=a")
(div:QI (match_operand:HI 1 "register_operand" "0")
(match_operand:QI 2 "nonimmediate_operand" "qm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "idiv{b}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "QI")])

(define_insn "udivqi3"
  [(set (match_operand:QI 0 "register_operand" "=a")
(udiv:QI (match_operand:HI 1 "register_operand" "0")
 (match_operand:QI 2 "nonimmediate_operand" "qm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "div{b}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "QI")])

;; The patterns that match these are at the end of this file.

(define_expand "divxf3"
```

```
  [(set (match_operand:XF 0 "register_operand" "")
(div:XF (match_operand:XF 1 "register_operand" "")
(match_operand:XF 2 "register_operand" "")))]
  "TARGET_80387"
  "")

(define_expand "divdf3"
  [(set (match_operand:DF 0 "register_operand" "")
  (div:DF (match_operand:DF 1 "register_operand" "")
  (match_operand:DF 2 "nonimmediate_operand" "")))]
   "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
   "")

(define_expand "divsf3"
  [(set (match_operand:SF 0 "register_operand" "")
(div:SF (match_operand:SF 1 "register_operand" "")
(match_operand:SF 2 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "")

;; Remainder instructions.

(define_expand "divmoddi4"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (div:DI (match_operand:DI 1 "register_operand" "")
   (match_operand:DI 2 "nonimmediate_operand" "")))
      (set (match_operand:DI 3 "register_operand" "")
   (mod:DI (match_dup 1) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

;; Allow to come the parameter in eax or edx to avoid extra moves.
;; Penalize eax case slightly because it results in worse scheduling
;; of code.
(define_insn "*divmoddi4_nocltd_rex64"
  [(set (match_operand:DI 0 "register_operand" "=&a,?a")
(div:DI (match_operand:DI 2 "register_operand" "1,0")
(match_operand:DI 3 "nonimmediate_operand" "rm,rm")))
   (set (match_operand:DI 1 "register_operand" "=&d,&d")
(mod:DI (match_dup 2) (match_dup 3)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && !optimize_size && !TARGET_USE_CLTD"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*divmoddi4_cltd_rex64"
  [(set (match_operand:DI 0 "register_operand" "=a")
(div:DI (match_operand:DI 2 "register_operand" "a")
(match_operand:DI 3 "nonimmediate_operand" "rm")))
```

```
   (set (match_operand:DI 1 "register_operand" "=&d")
(mod:DI (match_dup 2) (match_dup 3)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (optimize_size || TARGET_USE_CLTD)"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*divmoddi_noext_rex64"
  [(set (match_operand:DI 0 "register_operand" "=a")
(div:DI (match_operand:DI 1 "register_operand" "0")
(match_operand:DI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:DI 3 "register_operand" "=d")
(mod:DI (match_dup 1) (match_dup 2)))
   (use (match_operand:DI 4 "register_operand" "3"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "idiv{q}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(div:DI (match_operand:DI 1 "register_operand" "")
(match_operand:DI 2 "nonimmediate_operand" "")))
   (set (match_operand:DI 3 "register_operand" "")
(mod:DI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(parallel [(set (match_dup 3)
   (ashiftrt:DI (match_dup 4) (const_int 63)))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (match_dup 0)
            (div:DI (reg:DI 0) (match_dup 2)))
      (set (match_dup 3)
   (mod:DI (reg:DI 0) (match_dup 2)))
      (use (match_dup 3))
      (clobber (reg:CC FLAGS_REG))])]
{
  /* Avoid use of cltd in favor of a mov+shift.  */
  if (!TARGET_USE_CLTD && !optimize_size)
    {
      if (true_regnum (operands[1]))
        emit_move_insn (operands[0], operands[1]);
      else
emit_move_insn (operands[3], operands[1]);
      operands[4] = operands[3];
    }
  else
    {
      gcc_assert (!true_regnum (operands[1]));
```

```
      operands[4] = operands[1];
    }
})


(define_expand "divmodsi4"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (div:SI (match_operand:SI 1 "register_operand" "")
   (match_operand:SI 2 "nonimmediate_operand" "")))
      (set (match_operand:SI 3 "register_operand" "")
   (mod:SI (match_dup 1) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

;; Allow to come the parameter in eax or edx to avoid extra moves.
;; Penalize eax case slightly because it results in worse scheduling
;; of code.
(define_insn "*divmodsi4_nocltd"
  [(set (match_operand:SI 0 "register_operand" "=&a,?a")
(div:SI (match_operand:SI 2 "register_operand" "1,0")
(match_operand:SI 3 "nonimmediate_operand" "rm,rm")))
   (set (match_operand:SI 1 "register_operand" "=&d,&d")
(mod:SI (match_dup 2) (match_dup 3)))
   (clobber (reg:CC FLAGS_REG))]
  "!optimize_size && !TARGET_USE_CLTD"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*divmodsi4_cltd"
  [(set (match_operand:SI 0 "register_operand" "=a")
(div:SI (match_operand:SI 2 "register_operand" "a")
(match_operand:SI 3 "nonimmediate_operand" "rm")))
   (set (match_operand:SI 1 "register_operand" "=&d")
(mod:SI (match_dup 2) (match_dup 3)))
   (clobber (reg:CC FLAGS_REG))]
  "optimize_size || TARGET_USE_CLTD"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*divmodsi_noext"
  [(set (match_operand:SI 0 "register_operand" "=a")
(div:SI (match_operand:SI 1 "register_operand" "0")
(match_operand:SI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:SI 3 "register_operand" "=d")
(mod:SI (match_dup 1) (match_dup 2)))
   (use (match_operand:SI 4 "register_operand" "3"))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "idiv{l}\t%2"
```

```
  [(set_attr "type" "idiv")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(div:SI (match_operand:SI 1 "register_operand" "")
(match_operand:SI 2 "nonimmediate_operand" "")))
   (set (match_operand:SI 3 "register_operand" "")
(mod:SI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(parallel [(set (match_dup 3)
   (ashiftrt:SI (match_dup 4) (const_int 31)))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (match_dup 0)
           (div:SI (reg:SI 0) (match_dup 2)))
      (set (match_dup 3)
   (mod:SI (reg:SI 0) (match_dup 2)))
      (use (match_dup 3))
      (clobber (reg:CC FLAGS_REG))])]
{
  /* Avoid use of cltd in favor of a mov+shift.  */
  if (!TARGET_USE_CLTD && !optimize_size)
    {
      if (true_regnum (operands[1]))
        emit_move_insn (operands[0], operands[1]);
      else
emit_move_insn (operands[3], operands[1]);
      operands[4] = operands[3];
    }
  else
    {
      gcc_assert (!true_regnum (operands[1]));
      operands[4] = operands[1];
    }
})
;; %%% Split me.
(define_insn "divmodhi4"
  [(set (match_operand:HI 0 "register_operand" "=a")
(div:HI (match_operand:HI 1 "register_operand" "0")
(match_operand:HI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:HI 3 "register_operand" "=&d")
(mod:HI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "cwtd\;idiv{w}\t%2"
  [(set_attr "type" "multi")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "SI")])
```

```
(define_insn "udivmoddi4"
  [(set (match_operand:DI 0 "register_operand" "=a")
(udiv:DI (match_operand:DI 1 "register_operand" "0")
 (match_operand:DI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:DI 3 "register_operand" "=&d")
(umod:DI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "xor{q}\t%3, %3\;div{q}\t%2"
  [(set_attr "type" "multi")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "DI")])

(define_insn "*udivmoddi4_noext"
  [(set (match_operand:DI 0 "register_operand" "=a")
(udiv:DI (match_operand:DI 1 "register_operand" "0")
 (match_operand:DI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:DI 3 "register_operand" "=d")
(umod:DI (match_dup 1) (match_dup 2)))
   (use (match_dup 3))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "div{q}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(udiv:DI (match_operand:DI 1 "register_operand" "")
 (match_operand:DI 2 "nonimmediate_operand" "")))
   (set (match_operand:DI 3 "register_operand" "")
(umod:DI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(set (match_dup 3) (const_int 0))
   (parallel [(set (match_dup 0)
   (udiv:DI (match_dup 1) (match_dup 2)))
      (set (match_dup 3)
   (umod:DI (match_dup 1) (match_dup 2)))
      (use (match_dup 3))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_insn "udivmodsi4"
  [(set (match_operand:SI 0 "register_operand" "=a")
(udiv:SI (match_operand:SI 1 "register_operand" "0")
 (match_operand:SI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:SI 3 "register_operand" "=&d")
(umod:SI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
```

```
  ""
  "xor{l}\t%3, %3\;div{l}\t%2"
  [(set_attr "type" "multi")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "SI")])

(define_insn "*udivmodsi4_noext"
  [(set (match_operand:SI 0 "register_operand" "=a")
(udiv:SI (match_operand:SI 1 "register_operand" "0")
 (match_operand:SI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:SI 3 "register_operand" "=d")
(umod:SI (match_dup 1) (match_dup 2)))
   (use (match_dup 3))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "div{l}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(udiv:SI (match_operand:SI 1 "register_operand" "")
 (match_operand:SI 2 "nonimmediate_operand" "")))
   (set (match_operand:SI 3 "register_operand" "")
(umod:SI (match_dup 1) (match_dup 2)))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(set (match_dup 3) (const_int 0))
   (parallel [(set (match_dup 0)
   (udiv:SI (match_dup 1) (match_dup 2)))
      (set (match_dup 3)
   (umod:SI (match_dup 1) (match_dup 2)))
      (use (match_dup 3))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_expand "udivmodhi4"
  [(set (match_dup 4) (const_int 0))
   (parallel [(set (match_operand:HI 0 "register_operand" "")
   (udiv:HI (match_operand:HI 1 "register_operand" "")
      (match_operand:HI 2 "nonimmediate_operand" "")))
      (set (match_operand:HI 3 "register_operand" "")
       (umod:HI (match_dup 1) (match_dup 2)))
      (use (match_dup 4))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_HIMODE_MATH"
  "operands[4] = gen_reg_rtx (HImode);")

(define_insn "*udivmodhi_noext"
  [(set (match_operand:HI 0 "register_operand" "=a")
```

```
(udiv:HI (match_operand:HI 1 "register_operand" "0")
 (match_operand:HI 2 "nonimmediate_operand" "rm")))
   (set (match_operand:HI 3 "register_operand" "=d")
(umod:HI (match_dup 1) (match_dup 2)))
   (use (match_operand:HI 4 "register_operand" "3"))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "div{w}\t%2"
  [(set_attr "type" "idiv")
   (set_attr "mode" "HI")])

;; We cannot use div/idiv for double division, because it causes
;; "division by zero" on the overflow and that's not what we expect
;; from truncate.  Because true (non truncating) double division is
;; never generated, we can't create this insn anyway.
;
;(define_insn ""
;  [(set (match_operand:SI 0 "register_operand" "=a")
; (truncate:SI
;   (udiv:DI (match_operand:DI 1 "register_operand" "A")
;    (zero_extend:DI
;      (match_operand:SI 2 "nonimmediate_operand" "rm")))))
;   (set (match_operand:SI 3 "register_operand" "=d")
; (truncate:SI
;   (umod:DI (match_dup 1) (zero_extend:DI (match_dup 2)))))
;   (clobber (reg:CC FLAGS_REG))]
;  ""
;  "div{l}\t{%2, %0|%0, %2}"
;  [(set_attr "type" "idiv")])

;;- Logical AND instructions

;; On Pentium, "test imm, reg" is pairable only with eax, ax, and al.
;; Note that this excludes ah.

(define_insn "*testdi_1_rex64"
  [(set (reg FLAGS_REG)
(compare
  (and:DI (match_operand:DI 0 "nonimmediate_operand" "%!*a,r,!*a,r,rm")
  (match_operand:DI 1 "x86_64_szext_general_operand" "Z,Z,e,e,re"))
  (const_int 0)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
   test{l}\t{%k1, %k0|%k0, %k1}
   test{l}\t{%k1, %k0|%k0, %k1}
   test{q}\t{%1, %0|%0, %1}
   test{q}\t{%1, %0|%0, %1}
   test{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test")
```

```
   (set_attr "modrm" "0,1,0,1,1")
   (set_attr "mode" "SI,SI,DI,DI,DI")
   (set_attr "pent_pair" "uv,np,uv,np,uv")])

(define_insn "testsi_1"
  [(set (reg FLAGS_REG)
(compare
  (and:SI (match_operand:SI 0 "nonimmediate_operand" "%!*a,r,rm")
  (match_operand:SI 1 "general_operand" "in,in,rin"))
  (const_int 0)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "test{l}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test")
   (set_attr "modrm" "0,1,1")
   (set_attr "mode" "SI")
   (set_attr "pent_pair" "uv,np,uv")])

(define_expand "testsi_ccno_1"
  [(set (reg:CCNO FLAGS_REG)
(compare:CCNO
  (and:SI (match_operand:SI 0 "nonimmediate_operand" "")
  (match_operand:SI 1 "nonmemory_operand" ""))
  (const_int 0)))]
  ""
  "")

(define_insn "*testhi_1"
  [(set (reg FLAGS_REG)
        (compare (and:HI (match_operand:HI 0 "nonimmediate_operand" "%!*a,r,rm")
 (match_operand:HI 1 "general_operand" "n,n,rn"))
 (const_int 0)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "test{w}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test")
   (set_attr "modrm" "0,1,1")
   (set_attr "mode" "HI")
   (set_attr "pent_pair" "uv,np,uv")])

(define_expand "testqi_ccz_1"
  [(set (reg:CCZ FLAGS_REG)
        (compare:CCZ (and:QI (match_operand:QI 0 "nonimmediate_operand" "")
     (match_operand:QI 1 "nonmemory_operand" ""))
 (const_int 0)))]
  ""
  "")

(define_insn "*testqi_1_maybe_si"
  [(set (reg FLAGS_REG)
```

```
       (compare
  (and:QI
    (match_operand:QI 0 "nonimmediate_operand" "%!*a,q,qm,r")
    (match_operand:QI 1 "general_operand" "n,n,qn,n"))
  (const_int 0)))]
   "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
    && ix86_match_ccmode (insn,
   GET_CODE (operands[1]) == CONST_INT
   && INTVAL (operands[1]) >= 0 ? CCNOmode : CCZmode)"
{
  if (which_alternative == 3)
    {
      if (GET_CODE (operands[1]) == CONST_INT && INTVAL (operands[1]) < 0)
operands[1] = GEN_INT (INTVAL (operands[1]) & 0xff);
      return "test{l}\t{%1, %k0|%k0, %1}";
    }
  return "test{b}\t{%1, %0|%0, %1}";
}
  [(set_attr "type" "test")
   (set_attr "modrm" "0,1,1,1")
   (set_attr "mode" "QI,QI,QI,SI")
   (set_attr "pent_pair" "uv,np,uv,np")])

(define_insn "*testqi_1"
  [(set (reg FLAGS_REG)
       (compare
  (and:QI
    (match_operand:QI 0 "nonimmediate_operand" "%!*a,q,qm")
    (match_operand:QI 1 "general_operand" "n,n,qn"))
  (const_int 0)))]
  "(GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)
   && ix86_match_ccmode (insn, CCNOmode)"
  "test{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test")
   (set_attr "modrm" "0,1,1")
   (set_attr "mode" "QI")
   (set_attr "pent_pair" "uv,np,uv")])

(define_expand "testqi_ext_ccno_0"
  [(set (reg:CCNO FLAGS_REG)
(compare:CCNO
  (and:SI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "")
      (const_int 8)
      (const_int 8))
    (match_operand 1 "const_int_operand" ""))
  (const_int 0)))]
  ""
  "")
```

```
(define_insn "*testqi_ext_0"
  [(set (reg FLAGS_REG)
(compare
  (and:SI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8))
    (match_operand 1 "const_int_operand" "n"))
  (const_int 0)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "test{b}\t{%1, %h0|%h0, %1}"
  [(set_attr "type" "test")
   (set_attr "mode" "QI")
   (set_attr "length_immediate" "1")
   (set_attr "pent_pair" "np")])

(define_insn "*testqi_ext_1"
  [(set (reg FLAGS_REG)
(compare
  (and:SI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8))
    (zero_extend:SI
      (match_operand:QI 1 "general_operand" "Qm")))
  (const_int 0)))]
  "!TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "test{b}\t{%1, %h0|%h0, %1}"
  [(set_attr "type" "test")
   (set_attr "mode" "QI")])

(define_insn "*testqi_ext_1_rex64"
  [(set (reg FLAGS_REG)
(compare
  (and:SI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8))
    (zero_extend:SI
      (match_operand:QI 1 "register_operand" "Q")))
  (const_int 0)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  "test{b}\t{%1, %h0|%h0, %1}"
  [(set_attr "type" "test")
   (set_attr "mode" "QI")])
```

```
(define_insn "*testqi_ext_2"
  [(set (reg FLAGS_REG)
(compare
  (and:SI
    (zero_extract:SI
      (match_operand 0 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8))
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "Q")
      (const_int 8)
      (const_int 8)))
  (const_int 0)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "test{b}\t{%h1, %h0|%h0, %h1}"
  [(set_attr "type" "test")
   (set_attr "mode" "QI")])

;; Combine likes to form bit extractions for some tests.  Humor it.
(define_insn "*testqi_ext_3"
  [(set (reg FLAGS_REG)
        (compare (zero_extract:SI
   (match_operand 0 "nonimmediate_operand" "rm")
   (match_operand:SI 1 "const_int_operand" "")
   (match_operand:SI 2 "const_int_operand" ""))
 (const_int 0)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_MODE (operands[0]) == SImode
       || (TARGET_64BIT && GET_MODE (operands[0]) == DImode)
       || GET_MODE (operands[0]) == HImode
       || GET_MODE (operands[0]) == QImode)"
  "#")

(define_insn "*testqi_ext_3_rex64"
  [(set (reg FLAGS_REG)
        (compare (zero_extract:DI
   (match_operand 0 "nonimmediate_operand" "rm")
   (match_operand:DI 1 "const_int_operand" "")
   (match_operand:DI 2 "const_int_operand" ""))
 (const_int 0)))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCNOmode)
   /* The code below cannot deal with constants outside HOST_WIDE_INT.  */
   && INTVAL (operands[1]) + INTVAL (operands[2]) < HOST_BITS_PER_WIDE_INT
   /* Ensure that resulting mask is zero or sign extended operand.  */
   && (INTVAL (operands[1]) + INTVAL (operands[2]) <= 32
       || (INTVAL (operands[1]) + INTVAL (operands[2]) == 64
   && INTVAL (operands[1]) > 32))
   && (GET_MODE (operands[0]) == SImode
```

```
       || GET_MODE (operands[0]) == DImode
       || GET_MODE (operands[0]) == HImode
       || GET_MODE (operands[0]) == QImode)"
  "#")

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
        (match_operator 1 "compare_operator"
  [(zero_extract
     (match_operand 2 "nonimmediate_operand" "")
     (match_operand 3 "const_int_operand" "")
     (match_operand 4 "const_int_operand" ""))
   (const_int 0)]))]
  "ix86_match_ccmode (insn, CCNOmode)"
  [(set (match_dup 0) (match_op_dup 1 [(match_dup 2) (const_int 0)]))]
{
  rtx val = operands[2];
  HOST_WIDE_INT len = INTVAL (operands[3]);
  HOST_WIDE_INT pos = INTVAL (operands[4]);
  HOST_WIDE_INT mask;
  enum machine_mode mode, submode;

  mode = GET_MODE (val);
  if (GET_CODE (val) == MEM)
    {
      /* ??? Combine likes to put non-volatile mem extractions in QImode
 no matter the size of the test.  So find a mode that works.  */
      if (! MEM_VOLATILE_P (val))
{
  mode = smallest_mode_for_size (pos + len, MODE_INT);
  val = adjust_address (val, mode, 0);
}
    }
  else if (GET_CODE (val) == SUBREG
   && (submode = GET_MODE (SUBREG_REG (val)),
       GET_MODE_BITSIZE (mode) > GET_MODE_BITSIZE (submode))
   && pos + len <= GET_MODE_BITSIZE (submode))
    {
      /* Narrow a paradoxical subreg to prevent partial register stalls.  */
      mode = submode;
      val = SUBREG_REG (val);
    }
  else if (mode == HImode && pos + len <= 8)
    {
      /* Small HImode tests can be converted to QImode.  */
      mode = QImode;
      val = gen_lowpart (QImode, val);
    }

  mask  = ((HOST_WIDE_INT)1 << (pos + len)) - 1;
```

```
  mask &= ~(((HOST_WIDE_INT)1 << pos) - 1);

  operands[2] = gen_rtx_AND (mode, val, gen_int_mode (mask, mode));
})

;; Convert HImode/SImode test instructions with immediate to QImode ones.
;; i386 does not allow to encode test with 8bit sign extended immediate, so
;; this is relatively important trick.
;; Do the conversion only post-reload to avoid limiting of the register class
;; to QI regs.
(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and (match_operand 2 "register_operand" "")
        (match_operand 3 "const_int_operand" ""))
   (const_int 0)]))]
   "reload_completed
    && QI_REG_P (operands[2])
    && GET_MODE (operands[2]) != QImode
    && ((ix86_match_ccmode (insn, CCZmode)
      && !(INTVAL (operands[3]) & ~(255 << 8)))
|| (ix86_match_ccmode (insn, CCNOmode)
    && !(INTVAL (operands[3]) & ~(127 << 8))))"
  [(set (match_dup 0)
(match_op_dup 1
  [(and:SI (zero_extract:SI (match_dup 2) (const_int 8) (const_int 8))
   (match_dup 3))
   (const_int 0)]))]
  "operands[2] = gen_lowpart (SImode, operands[2]);
   operands[3] = gen_int_mode (INTVAL (operands[3]) >> 8, SImode);")

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and (match_operand 2 "nonimmediate_operand" "")
        (match_operand 3 "const_int_operand" ""))
   (const_int 0)]))]
   "reload_completed
    && GET_MODE (operands[2]) != QImode
    && (!REG_P (operands[2]) || ANY_QI_REG_P (operands[2]))
    && ((ix86_match_ccmode (insn, CCZmode)
 && !(INTVAL (operands[3]) & ~255))
|| (ix86_match_ccmode (insn, CCNOmode)
    && !(INTVAL (operands[3]) & ~127)))"
  [(set (match_dup 0)
(match_op_dup 1 [(and:QI (match_dup 2) (match_dup 3))
 (const_int 0)]))]
  "operands[2] = gen_lowpart (QImode, operands[2]);
   operands[3] = gen_lowpart (QImode, operands[3]);")
```

```
;; %%% This used to optimize known byte-wide and operations to memory,
;; and sometimes to QImode registers.  If this is considered useful,
;; it should be done with splitters.

(define_expand "anddi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(and:DI (match_operand:DI 1 "nonimmediate_operand" "")
(match_operand:DI 2 "x86_64_szext_general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (AND, DImode, operands); DONE;")

(define_insn "*anddi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=r,rm,r,r")
(and:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0,0,qm")
(match_operand:DI 2 "x86_64_szext_general_operand" "Z,re,rm,L")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (AND, DImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      {
enum machine_mode mode;

gcc_assert (GET_CODE (operands[2]) == CONST_INT);
        if (INTVAL (operands[2]) == 0xff)
  mode = QImode;
else
  {
    gcc_assert (INTVAL (operands[2]) == 0xffff);
    mode = HImode;
  }

operands[1] = gen_lowpart (mode, operands[1]);
if (mode == QImode)
  return "movz{bq|x}\t{%1,%0|%0, %1}";
else
  return "movz{wq|x}\t{%1,%0|%0, %1}";
      }

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      if (get_attr_mode (insn) == MODE_SI)
return "and{l}\t{%k2, %k0|%k0, %k2}";
      else
return "and{q}\t{%2, %0|%0, %2}";
    }
}
```

```
  [(set_attr "type" "alu,alu,alu,imovx")
   (set_attr "length_immediate" "*,*,*,0")
   (set_attr "mode" "SI,DI,DI,DI")])

(define_insn "*anddi_2"
  [(set (reg FLAGS_REG)
(compare (and:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0,0")
 (match_operand:DI 2 "x86_64_szext_general_operand" "Z,rem,re"))
 (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=r,r,rm")
(and:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (AND, DImode, operands)"
  "@
   and{l}\t{%k2, %k0|%k0, %k2}
   and{q}\t{%2, %0|%0, %2}
   and{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI,DI,DI")])

(define_expand "andsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(and:SI (match_operand:SI 1 "nonimmediate_operand" "")
(match_operand:SI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (AND, SImode, operands); DONE;")

(define_insn "*andsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r,r")
(and:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0,qm")
(match_operand:SI 2 "general_operand" "ri,rm,L")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (AND, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      {
enum machine_mode mode;

gcc_assert (GET_CODE (operands[2]) == CONST_INT);
        if (INTVAL (operands[2]) == 0xff)
  mode = QImode;
else
  {
    gcc_assert (INTVAL (operands[2]) == 0xffff);
    mode = HImode;
  }
```

```
operands[1] = gen_lowpart (mode, operands[1]);
if (mode == QImode)
  return "movz{bl|x}\t{%1,%0|%0, %1}";
else
  return "movz{wl|x}\t{%1,%0|%0, %1}";
      }

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      return "and{l}\t{%2, %0|%0, %2}";
    }
}
  [(set_attr "type" "alu,alu,imovx")
   (set_attr "length_immediate" "*,*,0")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand 0 "register_operand" "")
(and (match_dup 0)
     (const_int -65536)))
   (clobber (reg:CC FLAGS_REG))]
  "optimize_size || (TARGET_FAST_PREFIX && !TARGET_PARTIAL_REG_STALL)"
  [(set (strict_low_part (match_dup 1)) (const_int 0))]
  "operands[1] = gen_lowpart (HImode, operands[0]);")

(define_split
  [(set (match_operand 0 "ext_register_operand" "")
(and (match_dup 0)
     (const_int -256)))
   (clobber (reg:CC FLAGS_REG))]
  "(optimize_size || !TARGET_PARTIAL_REG_STALL) && reload_completed"
  [(set (strict_low_part (match_dup 1)) (const_int 0))]
  "operands[1] = gen_lowpart (QImode, operands[0]);")

(define_split
  [(set (match_operand 0 "ext_register_operand" "")
(and (match_dup 0)
     (const_int -65281)))
   (clobber (reg:CC FLAGS_REG))]
  "(optimize_size || !TARGET_PARTIAL_REG_STALL) && reload_completed"
  [(parallel [(set (zero_extract:SI (match_dup 0)
    (const_int 8)
    (const_int 8))
   (xor:SI
     (zero_extract:SI (match_dup 0)
      (const_int 8)
      (const_int 8))
     (zero_extract:SI (match_dup 0)
      (const_int 8)
      (const_int 8))))
```

```
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);")

;; See comment for addsi_1_zext why we do use nonimmediate_operand
(define_insn "*andsi_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (and:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
  (match_operand:SI 2 "general_operand" "rim"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (AND, SImode, operands)"
  "and{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*andsi_2"
  [(set (reg FLAGS_REG)
(compare (and:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
 (match_operand:SI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=r,rm")
(and:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (AND, SImode, operands)"
  "and{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
(define_insn "*andsi_2_zext"
  [(set (reg FLAGS_REG)
(compare (and:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand:SI 2 "general_operand" "rim"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (and:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (AND, SImode, operands)"
  "and{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_expand "andhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(and:HI (match_operand:HI 1 "nonimmediate_operand" "")
(match_operand:HI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (AND, HImode, operands); DONE;")
```

```
(define_insn "*andhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r,r")
(and:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0,qm")
(match_operand:HI 2 "general_operand" "ri,rm,L")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (AND, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOVX:
      gcc_assert (GET_CODE (operands[2]) == CONST_INT);
      gcc_assert (INTVAL (operands[2]) == 0xff);
      return "movz{bl|x}\t{%b1, %k0|%k0, %b1}";

    default:
      gcc_assert (rtx_equal_p (operands[0], operands[1]));

      return "and{w}\t{%2, %0|%0, %2}";
    }
}
  [(set_attr "type" "alu,alu,imovx")
   (set_attr "length_immediate" "*,*,0")
   (set_attr "mode" "HI,HI,SI")])

(define_insn "*andhi_2"
  [(set (reg FLAGS_REG)
(compare (and:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
 (match_operand:HI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=r,rm")
(and:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (AND, HImode, operands)"
  "and{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_expand "andqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(and:QI (match_operand:QI 1 "nonimmediate_operand" "")
(match_operand:QI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (AND, QImode, operands); DONE;")

;; %%% Potential partial reg stall on alternative 2.  What to do?
(define_insn "*andqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,q,r")
(and:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0,0")
(match_operand:QI 2 "general_operand" "qi,qmi,ri")))
```

```
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (AND, QImode, operands)"
  "@
   and{b}\t{%2, %0|%0, %2}
   and{b}\t{%2, %0|%0, %2}
   and{l}\t{%k2, %k0|%k0, %k2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI,QI,SI")])

(define_insn "*andqi_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,q"))
(and:QI (match_dup 0)
(match_operand:QI 1 "general_operand" "qi,qmi")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "and{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "*andqi_2_maybe_si"
  [(set (reg FLAGS_REG)
(compare (and:QI
      (match_operand:QI 1 "nonimmediate_operand" "%0,0,0")
      (match_operand:QI 2 "general_operand" "qim,qi,i"))
 (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=q,qm,*r")
(and:QI (match_dup 1) (match_dup 2)))]
  "ix86_binary_operator_ok (AND, QImode, operands)
   && ix86_match_ccmode (insn,
 GET_CODE (operands[2]) == CONST_INT
 && INTVAL (operands[2]) >= 0 ? CCNOmode : CCZmode)"
{
  if (which_alternative == 2)
    {
      if (GET_CODE (operands[2]) == CONST_INT && INTVAL (operands[2]) < 0)
        operands[2] = GEN_INT (INTVAL (operands[2]) & 0xff);
      return "and{l}\t{%2, %k0|%k0, %2}";
    }
  return "and{b}\t{%2, %0|%0, %2}";
}
  [(set_attr "type" "alu")
   (set_attr "mode" "QI,QI,SI")])

(define_insn "*andqi_2"
  [(set (reg FLAGS_REG)
(compare (and:QI
   (match_operand:QI 1 "nonimmediate_operand" "%0,0")
   (match_operand:QI 2 "general_operand" "qim,qi"))
 (const_int 0)))
```

```
   (set (match_operand:QI 0 "nonimmediate_operand" "=q,qm")
(and:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (AND, QImode, operands)"
  "and{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*andqi_2_slp"
  [(set (reg FLAGS_REG)
(compare (and:QI
   (match_operand:QI 0 "nonimmediate_operand" "+q,qm")
   (match_operand:QI 1 "nonimmediate_operand" "qmi,qi"))
 (const_int 0)))
   (set (strict_low_part (match_dup 0))
(and:QI (match_dup 0) (match_dup 1)))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "and{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

;; ??? A bug in recog prevents it from recognizing a const_int as an
;; operand to zero_extend in andqi_ext_1.  It was checking explicitly
;; for a QImode operand, which of course failed.

(define_insn "andqi_ext_0"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(and:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (match_operand 2 "const_int_operand" "n")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "and{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "1")
   (set_attr "mode" "QI")])

;; Generated by peephole translating test to and.  This shows up
;; often in fp comparisons.

(define_insn "*andqi_ext_0_cc"
  [(set (reg FLAGS_REG)
(compare
```

```
  (and:SI
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "0")
      (const_int 8)
      (const_int 8))
    (match_operand 2 "const_int_operand" "n"))
  (const_int 0)))
   (set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(and:SI
  (zero_extract:SI
    (match_dup 1)
    (const_int 8)
    (const_int 8))
  (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  "and{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "1")
   (set_attr "mode" "QI")])

(define_insn "*andqi_ext_1"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(and:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
    (match_operand:QI 2 "general_operand" "Qm"))))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "and{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*andqi_ext_1_rex64"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(and:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
```

```
    (match_operand 2 "ext_register_operand" "Q"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "and{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*andqi_ext_2"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(and:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "%0")
    (const_int 8)
    (const_int 8))
  (zero_extract:SI
    (match_operand 2 "ext_register_operand" "Q")
    (const_int 8)
    (const_int 8))))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "and{b}\t{%h2, %h0|%h0, %h2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

;; Convert wide AND instructions with immediate operand to shorter QImode
;; equivalents when possible.
;; Don't do the splitting with memory operands, since it introduces risk
;; of memory mismatch stalls.  We may want to do the splitting for optimizing
;; for size, but that can (should?) be handled by generic code instead.
(define_split
  [(set (match_operand 0 "register_operand" "")
(and (match_operand 1 "register_operand" "")
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(~INTVAL (operands[2]) & ~(255 << 8))
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (zero_extract:SI (match_dup 0) (const_int 8) (const_int 8))
   (and:SI (zero_extract:SI (match_dup 1)
    (const_int 8) (const_int 8))
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);
```

```
   operands[2] = gen_int_mode ((INTVAL (operands[2]) >> 8) & 0xff, SImode);")

;; Since AND can be encoded with sign extended immediate, this is only
;; profitable when 7th bit is not set.
(define_split
  [(set (match_operand 0 "register_operand" "")
(and (match_operand 1 "general_operand" "")
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && ANY_QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(~INTVAL (operands[2]) & ~255)
    && !(INTVAL (operands[2]) & 128)
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (strict_low_part (match_dup 0))
   (and:QI (match_dup 1)
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (QImode, operands[0]);
   operands[1] = gen_lowpart (QImode, operands[1]);
   operands[2] = gen_lowpart (QImode, operands[2]);")

;; Logical inclusive OR instructions

;; %%% This used to optimize known byte-wide and operations to memory.
;; If this is considered useful, it should be done with splitters.

(define_expand "iordi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(ior:DI (match_operand:DI 1 "nonimmediate_operand" "")
(match_operand:DI 2 "x86_64_general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (IOR, DImode, operands); DONE;")

(define_insn "*iordi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(ior:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
(match_operand:DI 2 "x86_64_general_operand" "re,rme")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && ix86_binary_operator_ok (IOR, DImode, operands)"
  "or{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "*iordi_2_rex64"
  [(set (reg FLAGS_REG)
(compare (ior:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
```

```
 (match_operand:DI 2 "x86_64_general_operand" "rem,re"))
 (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=r,rm")
(ior:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, DImode, operands)"
  "or{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])

(define_insn "*iordi_3_rex64"
  [(set (reg FLAGS_REG)
(compare (ior:DI (match_operand:DI 1 "nonimmediate_operand" "%0")
 (match_operand:DI 2 "x86_64_general_operand" "rem"))
 (const_int 0)))
   (clobber (match_scratch:DI 0 "=r"))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, DImode, operands)"
  "or{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])


(define_expand "iorsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(ior:SI (match_operand:SI 1 "nonimmediate_operand" "")
(match_operand:SI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (IOR, SImode, operands); DONE;")

(define_insn "*iorsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
(match_operand:SI 2 "general_operand" "ri,rmi")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (IOR, SImode, operands)"
  "or{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
(define_insn "*iorsi_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=rm")
(zero_extend:DI
  (ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
  (match_operand:SI 2 "general_operand" "rim"))))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "TARGET_64BIT && ix86_binary_operator_ok (IOR, SImode, operands)"
  "or{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*iorsi_1_zext_imm"
  [(set (match_operand:DI 0 "register_operand" "=rm")
(ior:DI (zero_extend:DI (match_operand:SI 1 "register_operand" "%0"))
(match_operand:DI 2 "x86_64_zext_immediate_operand" "Z")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "or{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*iorsi_2"
  [(set (reg FLAGS_REG)
(compare (ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
 (match_operand:SI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=r,rm")
(ior:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, SImode, operands)"
  "or{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
;; ??? Special case for immediate operand is missing - it is tricky.
(define_insn "*iorsi_2_zext"
  [(set (reg FLAGS_REG)
(compare (ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand:SI 2 "general_operand" "rim"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (ior:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, SImode, operands)"
  "or{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*iorsi_2_zext_imm"
  [(set (reg FLAGS_REG)
(compare (ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand 2 "x86_64_zext_immediate_operand" "Z"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(ior:DI (zero_extend:DI (match_dup 1)) (match_dup 2)))]
```

```
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, SImode, operands)"
  "or{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*iorsi_3"
  [(set (reg FLAGS_REG)
(compare (ior:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand:SI 2 "general_operand" "rim"))
 (const_int 0)))
   (clobber (match_scratch:SI 0 "=r"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "or{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_expand "iorhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(ior:HI (match_operand:HI 1 "nonimmediate_operand" "")
(match_operand:HI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (IOR, HImode, operands); DONE;")

(define_insn "*iorhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=r,m")
(ior:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
(match_operand:HI 2 "general_operand" "rmi,ri")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (IOR, HImode, operands)"
  "or{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*iorhi_2"
  [(set (reg FLAGS_REG)
(compare (ior:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
 (match_operand:HI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=r,rm")
(ior:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, HImode, operands)"
  "or{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*iorhi_3"
```

```
  [(set (reg FLAGS_REG)
(compare (ior:HI (match_operand:HI 1 "nonimmediate_operand" "%0")
 (match_operand:HI 2 "general_operand" "rim"))
 (const_int 0)))
   (clobber (match_scratch:HI 0 "=r"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "or{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_expand "iorqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(ior:QI (match_operand:QI 1 "nonimmediate_operand" "")
(match_operand:QI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (IOR, QImode, operands); DONE;")

;; %%% Potential partial reg stall on alternative 2.  What to do?
(define_insn "*iorqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=q,m,r")
(ior:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0,0")
(match_operand:QI 2 "general_operand" "qmi,qi,ri")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (IOR, QImode, operands)"
  "@
   or{b}\t{%2, %0|%0, %2}
   or{b}\t{%2, %0|%0, %2}
   or{l}\t{%k2, %k0|%k0, %k2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI,QI,SI")])

(define_insn "*iorqi_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+q,m"))
(ior:QI (match_dup 0)
(match_operand:QI 1 "general_operand" "qmi,qi")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "or{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "*iorqi_2"
  [(set (reg FLAGS_REG)
(compare (ior:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0")
 (match_operand:QI 2 "general_operand" "qim,qi"))
 (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=q,qm")
```

```
(ior:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (IOR, QImode, operands)"
  "or{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*iorqi_2_slp"
  [(set (reg FLAGS_REG)
(compare (ior:QI (match_operand:QI 0 "nonimmediate_operand" "+q,qm")
 (match_operand:QI 1 "general_operand" "qim,qi"))
 (const_int 0)))
   (set (strict_low_part (match_dup 0))
(ior:QI (match_dup 0) (match_dup 1)))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "or{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "*iorqi_3"
  [(set (reg FLAGS_REG)
(compare (ior:QI (match_operand:QI 1 "nonimmediate_operand" "%0")
 (match_operand:QI 2 "general_operand" "qim"))
 (const_int 0)))
   (clobber (match_scratch:QI 0 "=q"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "or{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "iorqi_ext_0"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(ior:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (match_operand 2 "const_int_operand" "n")))
   (clobber (reg:CC FLAGS_REG))]
  "(!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "or{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "1")
   (set_attr "mode" "QI")])
```

```
(define_insn "*iorqi_ext_1"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(ior:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
    (match_operand:QI 2 "general_operand" "Qm"))))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "or{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*iorqi_ext_1_rex64"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(ior:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
    (match_operand 2 "ext_register_operand" "Q"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "or{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*iorqi_ext_2"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(ior:SI
  (zero_extract:SI (match_operand 1 "ext_register_operand" "0")
      (const_int 8)
   (const_int 8))
  (zero_extract:SI (match_operand 2 "ext_register_operand" "Q")
      (const_int 8)
   (const_int 8))))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "(!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "ior{b}\t{%h2, %h0|%h0, %h2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_split
  [(set (match_operand 0 "register_operand" "")
(ior (match_operand 1 "register_operand" "")
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(INTVAL (operands[2]) & ~(255 << 8))
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (zero_extract:SI (match_dup 0) (const_int 8) (const_int 8))
   (ior:SI (zero_extract:SI (match_dup 1)
    (const_int 8) (const_int 8))
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);
   operands[2] = gen_int_mode ((INTVAL (operands[2]) >> 8) & 0xff, SImode);")

;; Since OR can be encoded with sign extended immediate, this is only
;; profitable when 7th bit is set.
(define_split
  [(set (match_operand 0 "register_operand" "")
(ior (match_operand 1 "general_operand" "")
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && ANY_QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(INTVAL (operands[2]) & ~255)
    && (INTVAL (operands[2]) & 128)
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (strict_low_part (match_dup 0))
   (ior:QI (match_dup 1)
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (QImode, operands[0]);
   operands[1] = gen_lowpart (QImode, operands[1]);
   operands[2] = gen_lowpart (QImode, operands[2]);")

;; Logical XOR instructions

;; %%% This used to optimize known byte-wide and operations to memory.
;; If this is considered useful, it should be done with splitters.
```

```
(define_expand "xordi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(xor:DI (match_operand:DI 1 "nonimmediate_operand" "")
(match_operand:DI 2 "x86_64_general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (XOR, DImode, operands); DONE;")

(define_insn "*xordi_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(xor:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
(match_operand:DI 2 "x86_64_general_operand" "re,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && ix86_binary_operator_ok (XOR, DImode, operands)"
  "@
   xor{q}\t{%2, %0|%0, %2}
   xor{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI,DI")])

(define_insn "*xordi_2_rex64"
  [(set (reg FLAGS_REG)
(compare (xor:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
 (match_operand:DI 2 "x86_64_general_operand" "rem,re"))
 (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=r,rm")
(xor:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, DImode, operands)"
  "@
   xor{q}\t{%2, %0|%0, %2}
   xor{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI,DI")])

(define_insn "*xordi_3_rex64"
  [(set (reg FLAGS_REG)
(compare (xor:DI (match_operand:DI 1 "nonimmediate_operand" "%0")
 (match_operand:DI 2 "x86_64_general_operand" "rem"))
 (const_int 0)))
   (clobber (match_scratch:DI 0 "=r"))]
  "TARGET_64BIT
   && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, DImode, operands)"
  "xor{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "DI")])
```

```
(define_expand "xorsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(xor:SI (match_operand:SI 1 "nonimmediate_operand" "")
(match_operand:SI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (XOR, SImode, operands); DONE;")

(define_insn "*xorsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
(match_operand:SI 2 "general_operand" "ri,rm")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
;; Add speccase for immediates
(define_insn "*xorsi_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
  (match_operand:SI 2 "general_operand" "rim"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*xorsi_1_zext_imm"
  [(set (match_operand:DI 0 "register_operand" "=r")
(xor:DI (zero_extend:DI (match_operand:SI 1 "register_operand" "%0"))
(match_operand:DI 2 "x86_64_zext_immediate_operand" "Z")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*xorsi_2"
  [(set (reg FLAGS_REG)
(compare (xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0")
 (match_operand:SI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=r,rm")
(xor:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
```

```
   && ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

;; See comment for addsi_1_zext why we do use nonimmediate_operand
;; ??? Special case for immediate operand is missing - it is tricky.
(define_insn "*xorsi_2_zext"
  [(set (reg FLAGS_REG)
(compare (xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand:SI 2 "general_operand" "rim"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (xor:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*xorsi_2_zext_imm"
  [(set (reg FLAGS_REG)
(compare (xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand 2 "x86_64_zext_immediate_operand" "Z"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(xor:DI (zero_extend:DI (match_dup 1)) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, SImode, operands)"
  "xor{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_insn "*xorsi_3"
  [(set (reg FLAGS_REG)
(compare (xor:SI (match_operand:SI 1 "nonimmediate_operand" "%0")
 (match_operand:SI 2 "general_operand" "rim"))
 (const_int 0)))
   (clobber (match_scratch:SI 0 "=r"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "xor{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "SI")])

(define_expand "xorhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(xor:HI (match_operand:HI 1 "nonimmediate_operand" "")
(match_operand:HI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (XOR, HImode, operands); DONE;")

(define_insn "*xorhi_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=r,m")
(xor:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
(match_operand:HI 2 "general_operand" "rmi,ri")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (XOR, HImode, operands)"
  "xor{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*xorhi_2"
  [(set (reg FLAGS_REG)
(compare (xor:HI (match_operand:HI 1 "nonimmediate_operand" "%0,0")
 (match_operand:HI 2 "general_operand" "rim,ri"))
 (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=r,rm")
(xor:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, HImode, operands)"
  "xor{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_insn "*xorhi_3"
  [(set (reg FLAGS_REG)
(compare (xor:HI (match_operand:HI 1 "nonimmediate_operand" "%0")
 (match_operand:HI 2 "general_operand" "rim"))
 (const_int 0)))
   (clobber (match_scratch:HI 0 "=r"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "xor{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "HI")])

(define_expand "xorqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(xor:QI (match_operand:QI 1 "nonimmediate_operand" "")
(match_operand:QI 2 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (XOR, QImode, operands); DONE;")

;; %%% Potential partial reg stall on alternative 2.  What to do?
(define_insn "*xorqi_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=q,m,r")
(xor:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0,0")
```

```
(match_operand:QI 2 "general_operand" "qmi,qi,ri")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (XOR, QImode, operands)"
  "@
   xor{b}\t{%2, %0|%0, %2}
   xor{b}\t{%2, %0|%0, %2}
   xor{l}\t{%k2, %k0|%k0, %k2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI,QI,SI")])

(define_insn "*xorqi_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,q"))
(xor:QI (match_dup 0)
(match_operand:QI 1 "general_operand" "qi,qmi")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "xor{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "xorqi_ext_0"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (match_operand 2 "const_int_operand" "n")))
   (clobber (reg:CC FLAGS_REG))]
  "(!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "xor{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "1")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_ext_1"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
    (match_operand:QI 2 "general_operand" "Qm"))))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "!TARGET_64BIT
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "xor{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_ext_1_rex64"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI
    (match_operand 1 "ext_register_operand" "0")
    (const_int 8)
    (const_int 8))
  (zero_extend:SI
    (match_operand 2 "ext_register_operand" "Q"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT
   && (!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "xor{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_ext_2"
  [(set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI (match_operand 1 "ext_register_operand" "0")
      (const_int 8)
   (const_int 8))
  (zero_extract:SI (match_operand 2 "ext_register_operand" "Q")
      (const_int 8)
   (const_int 8))))
   (clobber (reg:CC FLAGS_REG))]
  "(!TARGET_PARTIAL_REG_STALL || optimize_size)"
  "xor{b}\t{%h2, %h0|%h0, %h2}"
  [(set_attr "type" "alu")
   (set_attr "length_immediate" "0")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_cc_1"
  [(set (reg FLAGS_REG)
(compare
  (xor:QI (match_operand:QI 1 "nonimmediate_operand" "%0,0")
  (match_operand:QI 2 "general_operand" "qim,qi"))
  (const_int 0)))
```

```
   (set (match_operand:QI 0 "nonimmediate_operand" "=q,qm")
(xor:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_binary_operator_ok (XOR, QImode, operands)"
  "xor{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_2_slp"
  [(set (reg FLAGS_REG)
(compare (xor:QI (match_operand:QI 0 "nonimmediate_operand" "+q,qm")
 (match_operand:QI 1 "general_operand" "qim,qi"))
 (const_int 0)))
   (set (strict_low_part (match_dup 0))
(xor:QI (match_dup 0) (match_dup 1)))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "xor{b}\t{%1, %0|%0, %1}"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_cc_2"
  [(set (reg FLAGS_REG)
(compare
  (xor:QI (match_operand:QI 1 "nonimmediate_operand" "%0")
  (match_operand:QI 2 "general_operand" "qim"))
  (const_int 0)))
   (clobber (match_scratch:QI 0 "=q"))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "xor{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_cc_ext_1"
  [(set (reg FLAGS_REG)
(compare
  (xor:SI
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "0")
      (const_int 8)
      (const_int 8))
    (match_operand:QI 2 "general_operand" "qmn"))
  (const_int 0)))
   (set (zero_extract:SI (match_operand 0 "ext_register_operand" "=q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI (match_dup 1) (const_int 8) (const_int 8))
```

```
  (match_dup 2)))]
  "!TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  "xor{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_insn "*xorqi_cc_ext_1_rex64"
  [(set (reg FLAGS_REG)
(compare
  (xor:SI
    (zero_extract:SI
      (match_operand 1 "ext_register_operand" "0")
      (const_int 8)
      (const_int 8))
    (match_operand:QI 2 "nonmemory_operand" "Qn"))
  (const_int 0)))
   (set (zero_extract:SI (match_operand 0 "ext_register_operand" "=Q")
 (const_int 8)
 (const_int 8))
(xor:SI
  (zero_extract:SI (match_dup 1) (const_int 8) (const_int 8))
  (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  "xor{b}\t{%2, %h0|%h0, %2}"
  [(set_attr "type" "alu")
   (set_attr "mode" "QI")])

(define_expand "xorqi_cc_ext_1"
  [(parallel [
     (set (reg:CCNO FLAGS_REG)
  (compare:CCNO
    (xor:SI
      (zero_extract:SI
(match_operand 1 "ext_register_operand" "")
(const_int 8)
(const_int 8))
      (match_operand:QI 2 "general_operand" ""))
    (const_int 0)))
     (set (zero_extract:SI (match_operand 0 "ext_register_operand" "")
   (const_int 8)
   (const_int 8))
  (xor:SI
    (zero_extract:SI (match_dup 1) (const_int 8) (const_int 8))
    (match_dup 2)))])]
  ""
  "")

(define_split
  [(set (match_operand 0 "register_operand" "")
(xor (match_operand 1 "register_operand" "")
```

```
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(INTVAL (operands[2]) & ~(255 << 8))
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (zero_extract:SI (match_dup 0) (const_int 8) (const_int 8))
   (xor:SI (zero_extract:SI (match_dup 1)
    (const_int 8) (const_int 8))
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);
   operands[2] = gen_int_mode ((INTVAL (operands[2]) >> 8) & 0xff, SImode);")

;; Since XOR can be encoded with sign extended immediate, this is only
;; profitable when 7th bit is set.
(define_split
  [(set (match_operand 0 "register_operand" "")
(xor (match_operand 1 "general_operand" "")
     (match_operand 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
   "reload_completed
    && ANY_QI_REG_P (operands[0])
    && (!TARGET_PARTIAL_REG_STALL || optimize_size)
    && !(INTVAL (operands[2]) & ~255)
    && (INTVAL (operands[2]) & 128)
    && GET_MODE (operands[0]) != QImode"
  [(parallel [(set (strict_low_part (match_dup 0))
   (xor:QI (match_dup 1)
   (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (QImode, operands[0]);
   operands[1] = gen_lowpart (QImode, operands[1]);
   operands[2] = gen_lowpart (QImode, operands[2]);")

;; Negation instructions

(define_expand "negti2"
  [(parallel [(set (match_operand:TI 0 "nonimmediate_operand" "")
   (neg:TI (match_operand:TI 1 "nonimmediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "ix86_expand_unary_operator (NEG, TImode, operands); DONE;")

(define_insn "*negti2_1"
  [(set (match_operand:TI 0 "nonimmediate_operand" "=ro")
(neg:TI (match_operand:TI 1 "general_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "TARGET_64BIT
   && ix86_unary_operator_ok (NEG, TImode, operands)"
  "#")

(define_split
  [(set (match_operand:TI 0 "nonimmediate_operand" "")
(neg:TI (match_operand:TI 1 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(parallel
    [(set (reg:CCZ FLAGS_REG)
  (compare:CCZ (neg:DI (match_dup 2)) (const_int 0)))
     (set (match_dup 0) (neg:DI (match_dup 2)))])
   (parallel
    [(set (match_dup 1)
  (plus:DI (plus:DI (ltu:DI (reg:CC FLAGS_REG) (const_int 0))
    (match_dup 3))
   (const_int 0)))
     (clobber (reg:CC FLAGS_REG))])
   (parallel
    [(set (match_dup 1)
  (neg:DI (match_dup 1)))
     (clobber (reg:CC FLAGS_REG))])]
  "split_ti (operands+1, 1, operands+2, operands+3);
   split_ti (operands+0, 1, operands+0, operands+1);")

(define_expand "negdi2"
  [(parallel [(set (match_operand:DI 0 "nonimmediate_operand" "")
   (neg:DI (match_operand:DI 1 "nonimmediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_unary_operator (NEG, DImode, operands); DONE;")

(define_insn "*negdi2_1"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=ro")
(neg:DI (match_operand:DI 1 "general_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT
   && ix86_unary_operator_ok (NEG, DImode, operands)"
  "#")

(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(neg:DI (match_operand:DI 1 "general_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && reload_completed"
  [(parallel
    [(set (reg:CCZ FLAGS_REG)
  (compare:CCZ (neg:SI (match_dup 2)) (const_int 0)))
     (set (match_dup 0) (neg:SI (match_dup 2)))])
```

```
   (parallel
    [(set (match_dup 1)
  (plus:SI (plus:SI (ltu:SI (reg:CC FLAGS_REG) (const_int 0))
    (match_dup 3))
   (const_int 0)))
     (clobber (reg:CC FLAGS_REG))])
   (parallel
    [(set (match_dup 1)
  (neg:SI (match_dup 1)))
     (clobber (reg:CC FLAGS_REG))])]
  "split_di (operands+1, 1, operands+2, operands+3);
   split_di (operands+0, 1, operands+0, operands+1);")

(define_insn "*negdi2_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(neg:DI (match_operand:DI 1 "nonimmediate_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_unary_operator_ok (NEG, DImode, operands)"
  "neg{q}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "DI")])

;; The problem with neg is that it does not perform (compare x 0),
;; it really performs (compare 0 x), which leaves us with the zero
;; flag being the only useful item.

(define_insn "*negdi2_cmpz_rex64"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (neg:DI (match_operand:DI 1 "nonimmediate_operand" "0"))
     (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(neg:DI (match_dup 1)))]
  "TARGET_64BIT && ix86_unary_operator_ok (NEG, DImode, operands)"
  "neg{q}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "DI")])


(define_expand "negsi2"
  [(parallel [(set (match_operand:SI 0 "nonimmediate_operand" "")
   (neg:SI (match_operand:SI 1 "nonimmediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_unary_operator (NEG, SImode, operands); DONE;")

(define_insn "*negsi2_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(neg:SI (match_operand:SI 1 "nonimmediate_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_unary_operator_ok (NEG, SImode, operands)"
```

```
  "neg{l}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

;; Combine is quite creative about this pattern.
(define_insn "*negsi2_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (neg:DI (ashift:DI (match_operand:DI 1 "register_operand" "0")
(const_int 32)))
     (const_int 32)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_unary_operator_ok (NEG, SImode, operands)"
  "neg{l}\t%k0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

;; The problem with neg is that it does not perform (compare x 0),
;; it really performs (compare 0 x), which leaves us with the zero
;; flag being the only useful item.

(define_insn "*negsi2_cmpz"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (neg:SI (match_operand:SI 1 "nonimmediate_operand" "0"))
     (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(neg:SI (match_dup 1)))]
  "ix86_unary_operator_ok (NEG, SImode, operands)"
  "neg{l}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

(define_insn "*negsi2_cmpz_zext"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (lshiftrt:DI
       (neg:DI (ashift:DI
 (match_operand:DI 1 "register_operand" "0")
 (const_int 32)))
       (const_int 32))
     (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (neg:DI (ashift:DI (match_dup 1)
(const_int 32)))
     (const_int 32)))]
  "TARGET_64BIT && ix86_unary_operator_ok (NEG, SImode, operands)"
  "neg{l}\t%k0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

(define_expand "neghi2"
  [(parallel [(set (match_operand:HI 0 "nonimmediate_operand" "")
```

```
   (neg:HI (match_operand:HI 1 "nonimmediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_HIMODE_MATH"
  "ix86_expand_unary_operator (NEG, HImode, operands); DONE;")

(define_insn "*neghi2_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(neg:HI (match_operand:HI 1 "nonimmediate_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_unary_operator_ok (NEG, HImode, operands)"
  "neg{w}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "HI")])

(define_insn "*neghi2_cmpz"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (neg:HI (match_operand:HI 1 "nonimmediate_operand" "0"))
     (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(neg:HI (match_dup 1)))]
  "ix86_unary_operator_ok (NEG, HImode, operands)"
  "neg{w}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "HI")])

(define_expand "negqi2"
  [(parallel [(set (match_operand:QI 0 "nonimmediate_operand" "")
   (neg:QI (match_operand:QI 1 "nonimmediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_QIMODE_MATH"
  "ix86_expand_unary_operator (NEG, QImode, operands); DONE;")

(define_insn "*negqi2_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(neg:QI (match_operand:QI 1 "nonimmediate_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_unary_operator_ok (NEG, QImode, operands)"
  "neg{b}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "QI")])

(define_insn "*negqi2_cmpz"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (neg:QI (match_operand:QI 1 "nonimmediate_operand" "0"))
     (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(neg:QI (match_dup 1)))]
  "ix86_unary_operator_ok (NEG, QImode, operands)"
  "neg{b}\t%0"
  [(set_attr "type" "negnot")
```

```
   (set_attr "mode" "QI")])

;; Changing of sign for FP values is doable using integer unit too.

(define_expand "negsf2"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
(neg:SF (match_operand:SF 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_fp_absneg_operator (NEG, SFmode, operands); DONE;")

(define_expand "abssf2"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
(abs:SF (match_operand:SF 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_fp_absneg_operator (ABS, SFmode, operands); DONE;")

(define_insn "*absnegsf2_mixed"
  [(set (match_operand:SF 0 "nonimmediate_operand"    "=x#f,x#f,f#x,rm")
(match_operator:SF 3 "absneg_operator"
  [(match_operand:SF 1 "nonimmediate_operand" "0   ,x#f,0  ,0")]))
   (use (match_operand:V4SF 2 "nonimmediate_operand"  "xm  ,0  ,X  ,X"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_SSE_MATH && TARGET_MIX_SSE_I387
   && ix86_unary_operator_ok (GET_CODE (operands[3]), SFmode, operands)"
  "#")

(define_insn "*absnegsf2_sse"
  [(set (match_operand:SF 0 "nonimmediate_operand"    "=x,x,rm")
(match_operator:SF 3 "absneg_operator"
  [(match_operand:SF 1 "nonimmediate_operand" "0 ,x,0")]))
   (use (match_operand:V4SF 2 "nonimmediate_operand"  "xm,0,X"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_SSE_MATH
   && ix86_unary_operator_ok (GET_CODE (operands[3]), SFmode, operands)"
  "#")

(define_insn "*absnegsf2_i387"
  [(set (match_operand:SF 0 "nonimmediate_operand" "=f,rm")
(match_operator:SF 3 "absneg_operator"
  [(match_operand:SF 1 "nonimmediate_operand" "0,0")]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_80387 && !TARGET_SSE_MATH
   && ix86_unary_operator_ok (GET_CODE (operands[3]), SFmode, operands)"
  "#")

(define_expand "copysignsf3"
  [(match_operand:SF 0 "register_operand" "")
   (match_operand:SF 1 "nonmemory_operand" "")
   (match_operand:SF 2 "register_operand" "")]
```

```
  "TARGET_SSE_MATH"
{
  ix86_expand_copysign (operands);
  DONE;
})

(define_insn_and_split "copysignsf3_const"
  [(set (match_operand:SF 0 "register_operand"          "=x")
(unspec:SF
  [(match_operand:V4SF 1 "vector_move_operand"  "xmC")
   (match_operand:SF 2 "register_operand"       "0")
   (match_operand:V4SF 3 "nonimmediate_operand" "xm")]
  UNSPEC_COPYSIGN))]
  "TARGET_SSE_MATH"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  ix86_split_copysign_const (operands);
  DONE;
})

(define_insn "copysignsf3_var"
  [(set (match_operand:SF 0 "register_operand"          "=x, x, x, x,x")
(unspec:SF
  [(match_operand:SF 2 "register_operand"       " x, 0, 0, x,x")
   (match_operand:SF 3 "register_operand"       " 1, 1, x, 1,x")
   (match_operand:V4SF 4 "nonimmediate_operand" " X,xm,xm, 0,0")
   (match_operand:V4SF 5 "nonimmediate_operand" " 0,xm, 1,xm,1")]
  UNSPEC_COPYSIGN))
   (clobber (match_scratch:V4SF 1 "=x, x, x, x,x"))]
  "TARGET_SSE_MATH"
  "#")

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(unspec:SF
  [(match_operand:SF 2 "register_operand" "")
   (match_operand:SF 3 "register_operand" "")
   (match_operand:V4SF 4 "" "")
   (match_operand:V4SF 5 "" "")]
  UNSPEC_COPYSIGN))
   (clobber (match_scratch:V4SF 1 ""))]
  "TARGET_SSE_MATH && reload_completed"
  [(const_int 0)]
{
  ix86_split_copysign_var (operands);
  DONE;
})
```

```
(define_expand "negdf2"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
(neg:DF (match_operand:DF 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "ix86_expand_fp_absneg_operator (NEG, DFmode, operands); DONE;")

(define_expand "absdf2"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
(abs:DF (match_operand:DF 1 "nonimmediate_operand" "")))]
  "TARGET_80387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "ix86_expand_fp_absneg_operator (ABS, DFmode, operands); DONE;")

(define_insn "*absnegdf2_mixed"
  [(set (match_operand:DF 0 "nonimmediate_operand"    "=Y#f,Y#f,f#Y,rm")
(match_operator:DF 3 "absneg_operator"
  [(match_operand:DF 1 "nonimmediate_operand" "0   ,Y#f,0  ,0")]))
   (use (match_operand:V2DF 2 "nonimmediate_operand"  "Ym  ,0  ,X  ,X"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_SSE2 && TARGET_SSE_MATH && TARGET_MIX_SSE_I387
   && ix86_unary_operator_ok (GET_CODE (operands[3]), DFmode, operands)"
  "#")

(define_insn "*absnegdf2_sse"
  [(set (match_operand:DF 0 "nonimmediate_operand"    "=Y,Y,rm")
(match_operator:DF 3 "absneg_operator"
  [(match_operand:DF 1 "nonimmediate_operand" "0 ,Y,0")]))
   (use (match_operand:V2DF 2 "nonimmediate_operand"  "Ym,0,X"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_SSE2 && TARGET_SSE_MATH
   && ix86_unary_operator_ok (GET_CODE (operands[3]), DFmode, operands)"
  "#")

(define_insn "*absnegdf2_i387"
  [(set (match_operand:DF 0 "nonimmediate_operand" "=f,rm")
(match_operator:DF 3 "absneg_operator"
  [(match_operand:DF 1 "nonimmediate_operand" "0,0")]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_80387 && !(TARGET_SSE2 && TARGET_SSE_MATH)
   && ix86_unary_operator_ok (GET_CODE (operands[3]), DFmode, operands)"
  "#")

(define_expand "copysigndf3"
  [(match_operand:DF 0 "register_operand" "")
   (match_operand:DF 1 "nonmemory_operand" "")
   (match_operand:DF 2 "register_operand" "")]
  "TARGET_SSE2 && TARGET_SSE_MATH"
{
  ix86_expand_copysign (operands);
  DONE;
```

```
})

(define_insn_and_split "copysigndf3_const"
  [(set (match_operand:DF 0 "register_operand"          "=x")
(unspec:DF
  [(match_operand:V2DF 1 "vector_move_operand"  "xmC")
   (match_operand:DF 2 "register_operand"       "0")
   (match_operand:V2DF 3 "nonimmediate_operand" "xm")]
  UNSPEC_COPYSIGN))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  ix86_split_copysign_const (operands);
  DONE;
})

(define_insn "copysigndf3_var"
  [(set (match_operand:DF 0 "register_operand"          "=x, x, x, x,x")
(unspec:DF
  [(match_operand:DF 2 "register_operand"       " x, 0, 0, x,x")
   (match_operand:DF 3 "register_operand"       " 1, 1, x, 1,x")
   (match_operand:V2DF 4 "nonimmediate_operand" " X,xm,xm, 0,0")
   (match_operand:V2DF 5 "nonimmediate_operand" " 0,xm, 1,xm,1")]
  UNSPEC_COPYSIGN))
   (clobber (match_scratch:V2DF 1 "=x, x, x, x,x"))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "#")

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(unspec:DF
  [(match_operand:DF 2 "register_operand" "")
   (match_operand:DF 3 "register_operand" "")
   (match_operand:V2DF 4 "" "")
   (match_operand:V2DF 5 "" "")]
  UNSPEC_COPYSIGN))
   (clobber (match_scratch:V2DF 1 ""))]
  "TARGET_SSE2 && TARGET_SSE_MATH && reload_completed"
  [(const_int 0)]
{
  ix86_split_copysign_var (operands);
  DONE;
})

(define_expand "negxf2"
  [(set (match_operand:XF 0 "nonimmediate_operand" "")
(neg:XF (match_operand:XF 1 "nonimmediate_operand" "")))]
  "TARGET_80387"
```

```
  "ix86_expand_fp_absneg_operator (NEG, XFmode, operands); DONE;")

(define_expand "absxf2"
  [(set (match_operand:XF 0 "nonimmediate_operand" "")
(neg:XF (match_operand:XF 1 "nonimmediate_operand" "")))]
  "TARGET_80387"
  "ix86_expand_fp_absneg_operator (ABS, XFmode, operands); DONE;")

(define_insn "*absnegxf2_i387"
  [(set (match_operand:XF 0 "nonimmediate_operand" "=f,?rm")
(match_operator:XF 3 "absneg_operator"
  [(match_operand:XF 1 "nonimmediate_operand" "0,0")]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_80387
   && ix86_unary_operator_ok (GET_CODE (operands[3]), XFmode, operands)"
  "#")

;; Splitters for fp abs and neg.

(define_split
  [(set (match_operand 0 "fp_register_operand" "")
(match_operator 1 "absneg_operator" [(match_dup 0)]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(set (match_dup 0) (match_op_dup 1 [(match_dup 0)]))])

(define_split
  [(set (match_operand 0 "register_operand" "")
(match_operator 3 "absneg_operator"
  [(match_operand 1 "register_operand" "")]))
   (use (match_operand 2 "nonimmediate_operand" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed && SSE_REG_P (operands[0])"
  [(set (match_dup 0) (match_dup 3))]
{
  enum machine_mode mode = GET_MODE (operands[0]);
  enum machine_mode vmode = GET_MODE (operands[2]);
  rtx tmp;

  operands[0] = simplify_gen_subreg (vmode, operands[0], mode, 0);
  operands[1] = simplify_gen_subreg (vmode, operands[1], mode, 0);
  if (operands_match_p (operands[0], operands[2]))
    {
      tmp = operands[1];
      operands[1] = operands[2];
      operands[2] = tmp;
    }
  if (GET_CODE (operands[3]) == ABS)
```

```
    tmp = gen_rtx_AND (vmode, operands[1], operands[2]);
  else
    tmp = gen_rtx_XOR (vmode, operands[1], operands[2]);
  operands[3] = tmp;
})

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(match_operator:SF 1 "absneg_operator" [(match_dup 0)]))
   (use (match_operand:V4SF 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (match_dup 1))
      (clobber (reg:CC FLAGS_REG))])]
{
  rtx tmp;
  operands[0] = gen_lowpart (SImode, operands[0]);
  if (GET_CODE (operands[1]) == ABS)
    {
      tmp = gen_int_mode (0x7fffffff, SImode);
      tmp = gen_rtx_AND (SImode, operands[0], tmp);
    }
  else
    {
      tmp = gen_int_mode (0x80000000, SImode);
      tmp = gen_rtx_XOR (SImode, operands[0], tmp);
    }
  operands[1] = tmp;
})

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(match_operator:DF 1 "absneg_operator" [(match_dup 0)]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (match_dup 1))
      (clobber (reg:CC FLAGS_REG))])]
{
  rtx tmp;
  if (TARGET_64BIT)
    {
      tmp = gen_lowpart (DImode, operands[0]);
      tmp = gen_rtx_ZERO_EXTRACT (DImode, tmp, const1_rtx, GEN_INT (63));
      operands[0] = tmp;

      if (GET_CODE (operands[1]) == ABS)
tmp = const0_rtx;
      else
tmp = gen_rtx_NOT (DImode, tmp);
```

```
    }
  else
    {
      operands[0] = gen_highpart (SImode, operands[0]);
      if (GET_CODE (operands[1]) == ABS)
{
  tmp = gen_int_mode (0x7fffffff, SImode);
  tmp = gen_rtx_AND (SImode, operands[0], tmp);
}
      else
{
  tmp = gen_int_mode (0x80000000, SImode);
  tmp = gen_rtx_XOR (SImode, operands[0], tmp);
}
    }
  operands[1] = tmp;
})

(define_split
  [(set (match_operand:XF 0 "register_operand" "")
(match_operator:XF 1 "absneg_operator" [(match_dup 0)]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (match_dup 1))
      (clobber (reg:CC FLAGS_REG))])]
{
  rtx tmp;
  operands[0] = gen_rtx_REG (SImode,
     true_regnum (operands[0])
     + (TARGET_64BIT ? 1 : 2));
  if (GET_CODE (operands[1]) == ABS)
    {
      tmp = GEN_INT (0x7fff);
      tmp = gen_rtx_AND (SImode, operands[0], tmp);
    }
  else
    {
      tmp = GEN_INT (0x8000);
      tmp = gen_rtx_XOR (SImode, operands[0], tmp);
    }
  operands[1] = tmp;
})

(define_split
  [(set (match_operand 0 "memory_operand" "")
(match_operator 1 "absneg_operator" [(match_dup 0)]))
   (use (match_operand 2 "" ""))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed"
```

```
  [(parallel [(set (match_dup 0) (match_dup 1))
      (clobber (reg:CC FLAGS_REG))])]
{
  enum machine_mode mode = GET_MODE (operands[0]);
  int size = mode == XFmode ? 10 : GET_MODE_SIZE (mode);
  rtx tmp;

  operands[0] = adjust_address (operands[0], QImode, size - 1);
  if (GET_CODE (operands[1]) == ABS)
    {
      tmp = gen_int_mode (0x7f, QImode);
      tmp = gen_rtx_AND (QImode, operands[0], tmp);
    }
  else
    {
      tmp = gen_int_mode (0x80, QImode);
      tmp = gen_rtx_XOR (QImode, operands[0], tmp);
    }
  operands[1] = tmp;
})

;; Conditionalize these after reload. If they match before reload, we
;; lose the clobber and ability to use integer instructions.

(define_insn "*negsf2_1"
  [(set (match_operand:SF 0 "register_operand" "=f")
(neg:SF (match_operand:SF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "SF")])

(define_insn "*negdf2_1"
  [(set (match_operand:DF 0 "register_operand" "=f")
(neg:DF (match_operand:DF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "DF")])

(define_insn "*negxf2_1"
  [(set (match_operand:XF 0 "register_operand" "=f")
(neg:XF (match_operand:XF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "XF")])

(define_insn "*abssf2_1"
  [(set (match_operand:SF 0 "register_operand" "=f")
```

```
(abs:SF (match_operand:SF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "SF")])

(define_insn "*absdf2_1"
  [(set (match_operand:DF 0 "register_operand" "=f")
(abs:DF (match_operand:DF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "DF")])

(define_insn "*absxf2_1"
  [(set (match_operand:XF 0 "register_operand" "=f")
(abs:XF (match_operand:XF 1 "register_operand" "0")))]
  "TARGET_80387 && reload_completed"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "DF")])

(define_insn "*negextendsfdf2"
  [(set (match_operand:DF 0 "register_operand" "=f")
(neg:DF (float_extend:DF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_80387 && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "DF")])

(define_insn "*negextenddfxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(neg:XF (float_extend:XF
  (match_operand:DF 1 "register_operand" "0"))))]
  "TARGET_80387"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "XF")])

(define_insn "*negextendsfxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(neg:XF (float_extend:XF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_80387"
  "fchs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "XF")])

(define_insn "*absextendsfdf2"
```

```
  [(set (match_operand:DF 0 "register_operand" "=f")
(abs:DF (float_extend:DF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_80387 && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "DF")])

(define_insn "*absextenddfxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(abs:XF (float_extend:XF
  (match_operand:DF 1 "register_operand" "0"))))]
  "TARGET_80387"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "XF")])

(define_insn "*absextendsfxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(abs:XF (float_extend:XF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_80387"
  "fabs"
  [(set_attr "type" "fsgn")
   (set_attr "mode" "XF")])

;; One complement instructions

(define_expand "one_cmpldi2"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(not:DI (match_operand:DI 1 "nonimmediate_operand" "")))]
  "TARGET_64BIT"
  "ix86_expand_unary_operator (NOT, DImode, operands); DONE;")

(define_insn "*one_cmpldi2_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(not:DI (match_operand:DI 1 "nonimmediate_operand" "0")))]
  "TARGET_64BIT && ix86_unary_operator_ok (NOT, DImode, operands)"
  "not{q}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "DI")])

(define_insn "*one_cmpldi2_2_rex64"
  [(set (reg FLAGS_REG)
(compare (not:DI (match_operand:DI 1 "nonimmediate_operand" "0"))
 (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(not:DI (match_dup 1)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_unary_operator_ok (NOT, DImode, operands)"
```

```
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(not:DI (match_operand:DI 3 "nonimmediate_operand" ""))
   (const_int 0)]))
   (set (match_operand:DI 1 "nonimmediate_operand" "")
(not:DI (match_dup 3)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2
     [(xor:DI (match_dup 3) (const_int -1))
      (const_int 0)]))
      (set (match_dup 1)
   (xor:DI (match_dup 3) (const_int -1)))])]
  "")

(define_expand "one_cmplsi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(not:SI (match_operand:SI 1 "nonimmediate_operand" "")))]
  ""
  "ix86_expand_unary_operator (NOT, SImode, operands); DONE;")

(define_insn "*one_cmplsi2_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(not:SI (match_operand:SI 1 "nonimmediate_operand" "0")))]
  "ix86_unary_operator_ok (NOT, SImode, operands)"
  "not{l}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

;; ??? Currently never generated - xor is used instead.
(define_insn "*one_cmplsi2_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (not:SI (match_operand:SI 1 "register_operand" "0"))))]
  "TARGET_64BIT && ix86_unary_operator_ok (NOT, SImode, operands)"
  "not{l}\t%k0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "SI")])

(define_insn "*one_cmplsi2_2"
  [(set (reg FLAGS_REG)
(compare (not:SI (match_operand:SI 1 "nonimmediate_operand" "0"))
 (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(not:SI (match_dup 1)))]
  "ix86_match_ccmode (insn, CCNOmode)
```

```
   && ix86_unary_operator_ok (NOT, SImode, operands)"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(not:SI (match_operand:SI 3 "nonimmediate_operand" ""))
   (const_int 0)]))
   (set (match_operand:SI 1 "nonimmediate_operand" "")
(not:SI (match_dup 3)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2 [(xor:SI (match_dup 3) (const_int -1))
    (const_int 0)]))
      (set (match_dup 1)
   (xor:SI (match_dup 3) (const_int -1)))])]
  "")

;; ??? Currently never generated - xor is used instead.
(define_insn "*one_cmplsi2_2_zext"
  [(set (reg FLAGS_REG)
(compare (not:SI (match_operand:SI 1 "register_operand" "0"))
 (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (not:SI (match_dup 1))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)
   && ix86_unary_operator_ok (NOT, SImode, operands)"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "SI")])

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(not:SI (match_operand:SI 3 "register_operand" ""))
   (const_int 0)]))
   (set (match_operand:DI 1 "register_operand" "")
(zero_extend:DI (not:SI (match_dup 3))))]
  "ix86_match_ccmode (insn, CCNOmode)"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2 [(xor:SI (match_dup 3) (const_int -1))
    (const_int 0)]))
      (set (match_dup 1)
   (zero_extend:DI (xor:SI (match_dup 3) (const_int -1))))])]
  "")

(define_expand "one_cmplhi2"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
```

```
(not:HI (match_operand:HI 1 "nonimmediate_operand" "")))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_unary_operator (NOT, HImode, operands); DONE;")

(define_insn "*one_cmplhi2_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(not:HI (match_operand:HI 1 "nonimmediate_operand" "0")))]
  "ix86_unary_operator_ok (NOT, HImode, operands)"
  "not{w}\t%0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "HI")])

(define_insn "*one_cmplhi2_2"
  [(set (reg FLAGS_REG)
(compare (not:HI (match_operand:HI 1 "nonimmediate_operand" "0"))
 (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(not:HI (match_dup 1)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_unary_operator_ok (NEG, HImode, operands)"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "HI")])

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(not:HI (match_operand:HI 3 "nonimmediate_operand" ""))
   (const_int 0)]))
   (set (match_operand:HI 1 "nonimmediate_operand" "")
(not:HI (match_dup 3)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2 [(xor:HI (match_dup 3) (const_int -1))
           (const_int 0)]))
      (set (match_dup 1)
   (xor:HI (match_dup 3) (const_int -1)))])]
  "")

;; %%% Potential partial reg stall on alternative 1.  What to do?
(define_expand "one_cmplqi2"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(not:QI (match_operand:QI 1 "nonimmediate_operand" "")))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_unary_operator (NOT, QImode, operands); DONE;")

(define_insn "*one_cmplqi2_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,r")
(not:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")))]
  "ix86_unary_operator_ok (NOT, QImode, operands)"
```

```
  "@
   not{b}\t%0
   not{l}\t%k0"
  [(set_attr "type" "negnot")
   (set_attr "mode" "QI,SI")])

(define_insn "*one_cmplqi2_2"
  [(set (reg FLAGS_REG)
(compare (not:QI (match_operand:QI 1 "nonimmediate_operand" "0"))
 (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(not:QI (match_dup 1)))]
  "ix86_match_ccmode (insn, CCNOmode)
   && ix86_unary_operator_ok (NOT, QImode, operands)"
  "#"
  [(set_attr "type" "alu1")
   (set_attr "mode" "QI")])

(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(not:QI (match_operand:QI 3 "nonimmediate_operand" ""))
   (const_int 0)]))
   (set (match_operand:QI 1 "nonimmediate_operand" "")
(not:QI (match_dup 3)))]
  "ix86_match_ccmode (insn, CCNOmode)"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2 [(xor:QI (match_dup 3) (const_int -1))
            (const_int 0)]))
      (set (match_dup 1)
   (xor:QI (match_dup 3) (const_int -1)))])]
  "")

;; Arithmetic shift instructions

;; DImode shifts are implemented using the i386 "shift double" opcode,
;; which is written as "sh[lr]d[lw] imm,reg,reg/mem".  If the shift count
;; is variable, then the count is in %cl and the "imm" operand is dropped
;; from the assembler input.
;;
;; This instruction shifts the target reg/mem as usual, but instead of
;; shifting in zeros, bits are shifted in from reg operand.  If the insn
;; is a left shift double, bits are taken from the high order bits of
;; reg, else if the insn is a shift right double, bits are taken from the
;; low order bits of reg.  So if %eax is "1234" and %edx is "5678",
;; "shldl $8,%edx,%eax" leaves %edx unchanged and sets %eax to "2345".
;;
;; Since sh[lr]d does not change the `reg' operand, that is done
;; separately, making all shifts emit pairs of shift double and normal
;; shift.  Since sh[lr]d does not shift more than 31 bits, and we wish to
```

```
;; support a 63 bit shift, each shift where the count is in a reg expands
;; to a pair of shifts, a branch, a shift by 32 and a label.
;;
;; If the shift count is a constant, we need never emit more than one
;; shift pair, instead using moves and sign extension for counts greater
;; than 31.

(define_expand "ashlti3"
  [(parallel [(set (match_operand:TI 0 "register_operand" "")
   (ashift:TI (match_operand:TI 1 "register_operand" "")
      (match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
{
  if (! immediate_operand (operands[2], QImode))
    {
      emit_insn (gen_ashlti3_1 (operands[0], operands[1], operands[2]));
      DONE;
    }
  ix86_expand_binary_operator (ASHIFT, TImode, operands);
  DONE;
})

(define_insn "ashlti3_1"
  [(set (match_operand:TI 0 "register_operand" "=r")
(ashift:TI (match_operand:TI 1 "register_operand" "0")
   (match_operand:QI 2 "register_operand" "c")))
   (clobber (match_scratch:DI 3 "=&r"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*ashlti3_2"
  [(set (match_operand:TI 0 "register_operand" "=r")
(ashift:TI (match_operand:TI 1 "register_operand" "0")
   (match_operand:QI 2 "immediate_operand" "O")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(ashift:TI (match_operand:TI 1 "nonmemory_operand" "")
   (match_operand:QI 2 "register_operand" "")))
   (clobber (match_scratch:DI 3 ""))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
```

```
  "ix86_split_ashl (operands, operands[3], TImode); DONE;")

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(ashift:TI (match_operand:TI 1 "register_operand" "")
   (match_operand:QI 2 "immediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
  "ix86_split_ashl (operands, NULL_RTX, TImode); DONE;")

(define_insn "x86_64_shld"
  [(set (match_operand:DI 0 "nonimmediate_operand" "+r*m,r*m")
        (ior:DI (ashift:DI (match_dup 0)
  (match_operand:QI 2 "nonmemory_operand" "J,c"))
(lshiftrt:DI (match_operand:DI 1 "register_operand" "r,r")
  (minus:QI (const_int 64) (match_dup 2)))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "@
   shld{q}\t{%2, %1, %0|%0, %1, %2}
   shld{q}\t{%s2%1, %0|%0, %1, %2}"
  [(set_attr "type" "ishift")
   (set_attr "prefix_0f" "1")
   (set_attr "mode" "DI")
   (set_attr "athlon_decode" "vector")])

(define_expand "x86_64_shift_adj"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (and:QI (match_operand:QI 2 "register_operand" "")
     (const_int 64))
     (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "")
        (if_then_else:DI (ne (reg:CCZ FLAGS_REG) (const_int 0))
 (match_operand:DI 1 "register_operand" "")
 (match_dup 0)))
   (set (match_dup 1)
(if_then_else:DI (ne (reg:CCZ FLAGS_REG) (const_int 0))
 (match_operand:DI 3 "register_operand" "r")
 (match_dup 1)))]
  "TARGET_64BIT"
  "")

(define_expand "ashldi3"
  [(set (match_operand:DI 0 "shiftdi_operand" "")
(ashift:DI (match_operand:DI 1 "ashldi_input_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))]
  ""
  "ix86_expand_binary_operator (ASHIFT, DImode, operands); DONE;")
```

```
(define_insn "*ashldi3_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
(ashift:DI (match_operand:DI 1 "nonimmediate_operand" "0,l")
   (match_operand:QI 2 "nonmemory_operand" "cJ,M")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFT, DImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      return "add{q}\t{%0, %0|%0, %0}";

    case TYPE_LEA:
      gcc_assert (GET_CODE (operands[2]) == CONST_INT);
      gcc_assert ((unsigned HOST_WIDE_INT) INTVAL (operands[2]) <= 3);
      operands[1] = gen_rtx_MULT (DImode, operands[1],
  GEN_INT (1 << INTVAL (operands[2])));
      return "lea{q}\t{%a1, %0|%0, %a1}";

    default:
      if (REG_P (operands[2]))
return "sal{q}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{q}\t%0";
      else
return "sal{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "1")
      (const_string "lea")
            (and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "DI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(ashift:DI (match_operand:DI 1 "index_register_operand" "")
   (match_operand:QI 2 "immediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed
```

```
   && true_regnum (operands[0]) != true_regnum (operands[1])"
  [(set (match_dup 0)
(mult:DI (match_dup 1)
 (match_dup 2)))]
  "operands[2] = gen_int_mode (1 << INTVAL (operands[2]), DImode);")

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashldi3_cmp_rex64"
  [(set (reg FLAGS_REG)
(compare
  (ashift:DI (match_operand:DI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "immediate_operand" "e"))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(ashift:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFT, DImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{q}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{q}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{q}\t%0";
      else
return "sal{q}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
       (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "DI")])

(define_insn "*ashldi3_1"
  [(set (match_operand:DI 0 "register_operand" "=&r,r")
(ashift:DI (match_operand:DI 1 "reg_or_pm1_operand" "n,0")
   (match_operand:QI 2 "nonmemory_operand" "Jc,Jc")))
```

```
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

;; By default we don't ask for a scratch register, because when DImode
;; values are manipulated, registers are already at a premium.  But if
;; we have one handy, we won't turn it away.
(define_peephole2
  [(match_scratch:SI 3 "r")
   (parallel [(set (match_operand:DI 0 "register_operand" "")
   (ashift:DI (match_operand:DI 1 "nonmemory_operand" "")
      (match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_dup 3)]
  "!TARGET_64BIT && TARGET_CMOVE"
  [(const_int 0)]
  "ix86_split_ashl (operands, operands[3], DImode); DONE;")

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(ashift:DI (match_operand:DI 1 "nonmemory_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && (flag_peephole2 ? flow2_completed : reload_completed)"
  [(const_int 0)]
  "ix86_split_ashl (operands, NULL_RTX, DImode); DONE;")

(define_insn "x86_shld_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "+r*m,r*m")
        (ior:SI (ashift:SI (match_dup 0)
  (match_operand:QI 2 "nonmemory_operand" "I,c"))
(lshiftrt:SI (match_operand:SI 1 "register_operand" "r,r")
  (minus:QI (const_int 32) (match_dup 2)))))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "@
   shld{l}\t{%2, %1, %0|%0, %1, %2}
   shld{l}\t{%s2%1, %0|%0, %1, %2}"
  [(set_attr "type" "ishift")
   (set_attr "prefix_0f" "1")
   (set_attr "mode" "SI")
   (set_attr "pent_pair" "np")
   (set_attr "athlon_decode" "vector")])

(define_expand "x86_shift_adj_1"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (and:QI (match_operand:QI 2 "register_operand" "")
     (const_int 32))
     (const_int 0)))
```

```
   (set (match_operand:SI 0 "register_operand" "")
        (if_then_else:SI (ne (reg:CCZ FLAGS_REG) (const_int 0))
 (match_operand:SI 1 "register_operand" "")
 (match_dup 0)))
   (set (match_dup 1)
(if_then_else:SI (ne (reg:CCZ FLAGS_REG) (const_int 0))
 (match_operand:SI 3 "register_operand" "r")
 (match_dup 1)))]
  "TARGET_CMOVE"
  "")

(define_expand "x86_shift_adj_2"
  [(use (match_operand:SI 0 "register_operand" ""))
   (use (match_operand:SI 1 "register_operand" ""))
   (use (match_operand:QI 2 "register_operand" ""))]
  ""
{
  rtx label = gen_label_rtx ();
  rtx tmp;

  emit_insn (gen_testqi_ccz_1 (operands[2], GEN_INT (32)));

  tmp = gen_rtx_REG (CCZmode, FLAGS_REG);
  tmp = gen_rtx_EQ (VOIDmode, tmp, const0_rtx);
  tmp = gen_rtx_IF_THEN_ELSE (VOIDmode, tmp,
      gen_rtx_LABEL_REF (VOIDmode, label),
      pc_rtx);
  tmp = emit_jump_insn (gen_rtx_SET (VOIDmode, pc_rtx, tmp));
  JUMP_LABEL (tmp) = label;

  emit_move_insn (operands[0], operands[1]);
  ix86_expand_clear (operands[1]);

  emit_label (label);
  LABEL_NUSES (label) = 1;

  DONE;
})

(define_expand "ashlsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(ashift:SI (match_operand:SI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (ASHIFT, SImode, operands); DONE;")

(define_insn "*ashlsi3_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,r")
(ashift:SI (match_operand:SI 1 "nonimmediate_operand" "0,l")
```

```
   (match_operand:QI 2 "nonmemory_operand" "cI,M")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFT, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      gcc_assert (rtx_equal_p (operands[0], operands[1]));
      return "add{l}\t{%0, %0|%0, %0}";

    case TYPE_LEA:
      return "#";

    default:
      if (REG_P (operands[2]))
return "sal{l}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{l}\t%0";
      else
return "sal{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "1")
      (const_string "lea")
            (and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "SI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand 0 "register_operand" "")
(ashift (match_operand 1 "index_register_operand" "")
                (match_operand:QI 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])
   && GET_MODE_SIZE (GET_MODE (operands[0])) <= 4"
  [(const_int 0)]
{
  rtx pat;
  enum machine_mode mode = GET_MODE (operands[0]);
```

```
  if (GET_MODE_SIZE (mode) < 4)
    operands[0] = gen_lowpart (SImode, operands[0]);
  if (mode != Pmode)
    operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[2] = gen_int_mode (1 << INTVAL (operands[2]), Pmode);

  pat = gen_rtx_MULT (Pmode, operands[1], operands[2]);
  if (Pmode != SImode)
    pat = gen_rtx_SUBREG (SImode, pat, 0);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0], pat));
  DONE;
})

;; Rare case of shifting RSP is handled by generating move and shift
(define_split
  [(set (match_operand 0 "register_operand" "")
(ashift (match_operand 1 "register_operand" "")
                (match_operand:QI 2 "const_int_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])"
  [(const_int 0)]
{
  rtx pat, clob;
  emit_move_insn (operands[1], operands[0]);
  pat = gen_rtx_SET (VOIDmode, operands[0],
     gen_rtx_ASHIFT (GET_MODE (operands[0]),
     operands[0], operands[2]));
  clob = gen_rtx_CLOBBER (VOIDmode, gen_rtx_REG (CCmode, FLAGS_REG));
  emit_insn (gen_rtx_PARALLEL (VOIDmode, gen_rtvec (2, pat, clob)));
  DONE;
})

(define_insn "*ashlsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI (ashift:SI (match_operand:SI 1 "register_operand" "0,l")
(match_operand:QI 2 "nonmemory_operand" "cI,M"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFT, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{l}\t{%k0, %k0|%k0, %k0}";

    case TYPE_LEA:
      return "#";

    default:
```

```
      if (REG_P (operands[2]))
return "sal{l}\t{%b2, %k0|%k0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{l}\t%k0";
      else
return "sal{l}\t{%2, %k0|%k0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "1")
      (const_string "lea")
            (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
     (const_int 0))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "SI")])

;; Convert lea to the lea pattern to avoid flags dependency.
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(zero_extend:DI (ashift (match_operand 1 "register_operand" "")
(match_operand:QI 2 "const_int_operand" ""))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed
   && true_regnum (operands[0]) != true_regnum (operands[1])"
  [(set (match_dup 0) (zero_extend:DI
(subreg:SI (mult:SI (match_dup 1)
    (match_dup 2)) 0)))]
{
  operands[1] = gen_lowpart (Pmode, operands[1]);
  operands[2] = gen_int_mode (1 << INTVAL (operands[2]), Pmode);
})

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashlsi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashift:SI (match_operand:SI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(ashift:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFT, SImode, operands)"
{
```

```
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{l}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{l}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{l}\t%0";
      else
return "sal{l}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "SI")])

(define_insn "*ashlsi3_cmp_zext"
  [(set (reg FLAGS_REG)
(compare
  (ashift:SI (match_operand:SI 1 "register_operand" "0")
     (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (ashift:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFT, SImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{l}\t{%k0, %k0|%k0, %k0}";

    default:
      if (REG_P (operands[2]))
return "sal{l}\t{%b2, %k0|%k0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{l}\t%k0";
      else
```

```
return "sal{l}\t{%2, %k0|%k0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
     (const_int 0))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "SI")])

(define_expand "ashlhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(ashift:HI (match_operand:HI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (ASHIFT, HImode, operands); DONE;")

(define_insn "*ashlhi3_1_lea"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,r")
(ashift:HI (match_operand:HI 1 "nonimmediate_operand" "0,l")
   (match_operand:QI 2 "nonmemory_operand" "cI,M")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (ASHIFT, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      return "#";
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{w}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{w}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{w}\t%0";
      else
return "sal{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "1")
      (const_string "lea")
            (and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
```

```
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "HI,SI")])

(define_insn "*ashlhi3_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(ashift:HI (match_operand:HI 1 "nonimmediate_operand" "0")
   (match_operand:QI 2 "nonmemory_operand" "cI")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (ASHIFT, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{w}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{w}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{w}\t%0";
      else
return "sal{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "HI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashlhi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashift:HI (match_operand:HI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const_1_to_31_operand" "I"))
```

```
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(ashift:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFT, HImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{w}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{w}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{w}\t%0";
      else
return "sal{w}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "HI")])

(define_expand "ashlqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(ashift:QI (match_operand:QI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (ASHIFT, QImode, operands); DONE;")

;; %%% Potential partial reg stall on alternative 2.  What to do?

(define_insn "*ashlqi3_1_lea"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,r,r")
(ashift:QI (match_operand:QI 1 "nonimmediate_operand" "0,0,l")
   (match_operand:QI 2 "nonmemory_operand" "cI,cI,M")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (ASHIFT, QImode, operands)"
{
```

```
  switch (get_attr_type (insn))
    {
    case TYPE_LEA:
      return "#";
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      if (REG_P (operands[1]) && !ANY_QI_REG_P (operands[1]))
        return "add{l}\t{%k0, %k0|%k0, %k0}";
      else
        return "add{b}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t{%b2, %k0|%k0, %b2}";
  else
    return "sal{b}\t{%b2, %0|%0, %b2}";
}
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t%0";
  else
    return "sal{b}\t%0";
}
      else
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t{%2, %k0|%k0, %2}";
  else
    return "sal{b}\t{%2, %0|%0, %2}";
}
    }
}
  [(set (attr "type")
     (cond [(eq_attr "alternative" "2")
      (const_string "lea")
            (and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
         (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "QI,SI,SI")])

(define_insn "*ashlqi3_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,r")
```

```
(ashift:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
   (match_operand:QI 2 "nonmemory_operand" "cI,cI")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_PARTIAL_REG_STALL
   && ix86_binary_operator_ok (ASHIFT, QImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      if (REG_P (operands[1]) && !ANY_QI_REG_P (operands[1]))
        return "add{l}\t{%k0, %k0|%k0, %k0}";
      else
        return "add{b}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t{%b2, %k0|%k0, %b2}";
  else
    return "sal{b}\t{%b2, %0|%0, %b2}";
}
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t%0";
  else
    return "sal{b}\t%0";
}
      else
{
  if (get_attr_mode (insn) == MODE_SI)
    return "sal{l}\t{%2, %k0|%k0, %2}";
  else
    return "sal{b}\t{%2, %0|%0, %2}";
}
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "QI,SI")])
```

```
;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashlqi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashift:QI (match_operand:QI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(ashift:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFT, QImode, operands)"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      gcc_assert (operands[2] == const1_rtx);
      return "add{b}\t{%0, %0|%0, %0}";

    default:
      if (REG_P (operands[2]))
return "sal{b}\t{%b2, %0|%0, %b2}";
      else if (operands[2] == const1_rtx
       && (TARGET_SHIFT1 || optimize_size))
return "sal{b}\t%0";
      else
return "sal{b}\t{%2, %0|%0, %2}";
    }
}
  [(set (attr "type")
     (cond [(and (and (ne (symbol_ref "TARGET_DOUBLE_WITH_ADD")
          (const_int 0))
      (match_operand 0 "register_operand" ""))
 (match_operand 2 "const1_operand" ""))
      (const_string "alu")
   ]
   (const_string "ishift")))
   (set_attr "mode" "QI")])

;; See comment above `ashldi3' about how this works.

(define_expand "ashrti3"
  [(parallel [(set (match_operand:TI 0 "register_operand" "")
   (ashiftrt:TI (match_operand:TI 1 "register_operand" "")
(match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
{
  if (! immediate_operand (operands[2], QImode))
```

```
    {
      emit_insn (gen_ashrti3_1 (operands[0], operands[1], operands[2]));
      DONE;
    }
  ix86_expand_binary_operator (ASHIFTRT, TImode, operands);
  DONE;
})

(define_insn "ashrti3_1"
  [(set (match_operand:TI 0 "register_operand" "=r")
(ashiftrt:TI (match_operand:TI 1 "register_operand" "0")
     (match_operand:QI 2 "register_operand" "c")))
   (clobber (match_scratch:DI 3 "=&r"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*ashrti3_2"
  [(set (match_operand:TI 0 "register_operand" "=r")
(ashiftrt:TI (match_operand:TI 1 "register_operand" "0")
     (match_operand:QI 2 "immediate_operand" "O")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(ashiftrt:TI (match_operand:TI 1 "register_operand" "")
     (match_operand:QI 2 "register_operand" "")))
   (clobber (match_scratch:DI 3 ""))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
  "ix86_split_ashr (operands, operands[3], TImode); DONE;")

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(ashiftrt:TI (match_operand:TI 1 "register_operand" "")
     (match_operand:QI 2 "immediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
  "ix86_split_ashr (operands, NULL_RTX, TImode); DONE;")

(define_insn "x86_64_shrd"
  [(set (match_operand:DI 0 "nonimmediate_operand" "+r*m,r*m")
        (ior:DI (ashiftrt:DI (match_dup 0)
  (match_operand:QI 2 "nonmemory_operand" "J,c"))
```

```
(ashift:DI (match_operand:DI 1 "register_operand" "r,r")
  (minus:QI (const_int 64) (match_dup 2)))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "@
   shrd{q}\t{%2, %1, %0|%0, %1, %2}
   shrd{q}\t{%s2%1, %0|%0, %1, %2}"
  [(set_attr "type" "ishift")
   (set_attr "prefix_0f" "1")
   (set_attr "mode" "DI")
   (set_attr "athlon_decode" "vector")])

(define_expand "ashrdi3"
  [(set (match_operand:DI 0 "shiftdi_operand" "")
(ashiftrt:DI (match_operand:DI 1 "shiftdi_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))]
  ""
  "ix86_expand_binary_operator (ASHIFTRT, DImode, operands); DONE;")

(define_insn "*ashrdi3_63_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=*d,rm")
(ashiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "*a,0")
     (match_operand:DI 2 "const_int_operand" "i,i")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && INTVAL (operands[2]) == 63
   && (TARGET_USE_CLTD || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, DImode, operands)"
  "@
   {cqto|cqo}
   sar{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "imovx,ishift")
   (set_attr "prefix_0f" "0,*")
   (set_attr "length_immediate" "0,*")
   (set_attr "modrm" "0,1")
   (set_attr "mode" "DI")])

(define_insn "*ashrdi3_1_one_bit_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(ashiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFTRT, DImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{q}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])
```

```
(define_insn "*ashrdi3_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,rm")
(ashiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "J,c")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFTRT, DImode, operands)"
  "@
   sar{q}\t{%2, %0|%0, %2}
   sar{q}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "DI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrdi3_one_bit_cmp_rex64"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(ashiftrt:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, DImode, operands)"
  "sar{q}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrdi3_cmp_rex64"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_int_operand" "n"))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(ashiftrt:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFTRT, DImode, operands)"
  "sar{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "DI")])
```

```
(define_insn "*ashrdi3_1"
  [(set (match_operand:DI 0 "register_operand" "=r")
(ashiftrt:DI (match_operand:DI 1 "register_operand" "0")
     (match_operand:QI 2 "nonmemory_operand" "Jc")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

;; By default we don't ask for a scratch register, because when DImode
;; values are manipulated, registers are already at a premium.  But if
;; we have one handy, we won't turn it away.
(define_peephole2
  [(match_scratch:SI 3 "r")
   (parallel [(set (match_operand:DI 0 "register_operand" "")
   (ashiftrt:DI (match_operand:DI 1 "register_operand" "")
        (match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_dup 3)]
  "!TARGET_64BIT && TARGET_CMOVE"
  [(const_int 0)]
  "ix86_split_ashr (operands, operands[3], DImode); DONE;")

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(ashiftrt:DI (match_operand:DI 1 "register_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && (flag_peephole2 ? flow2_completed : reload_completed)"
  [(const_int 0)]
  "ix86_split_ashr (operands, NULL_RTX, DImode); DONE;")

(define_insn "x86_shrd_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "+r*m,r*m")
        (ior:SI (ashiftrt:SI (match_dup 0)
  (match_operand:QI 2 "nonmemory_operand" "I,c"))
(ashift:SI (match_operand:SI 1 "register_operand" "r,r")
  (minus:QI (const_int 32) (match_dup 2)))))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "@
   shrd{l}\t{%2, %1, %0|%0, %1, %2}
   shrd{l}\t{%s2%1, %0|%0, %1, %2}"
  [(set_attr "type" "ishift")
   (set_attr "prefix_0f" "1")
   (set_attr "pent_pair" "np")
   (set_attr "mode" "SI")])

(define_expand "x86_shift_adj_3"
  [(use (match_operand:SI 0 "register_operand" ""))
```

```
   (use (match_operand:SI 1 "register_operand" ""))
   (use (match_operand:QI 2 "register_operand" ""))]
  ""
{
  rtx label = gen_label_rtx ();
  rtx tmp;

  emit_insn (gen_testqi_ccz_1 (operands[2], GEN_INT (32)));

  tmp = gen_rtx_REG (CCZmode, FLAGS_REG);
  tmp = gen_rtx_EQ (VOIDmode, tmp, const0_rtx);
  tmp = gen_rtx_IF_THEN_ELSE (VOIDmode, tmp,
      gen_rtx_LABEL_REF (VOIDmode, label),
      pc_rtx);
  tmp = emit_jump_insn (gen_rtx_SET (VOIDmode, pc_rtx, tmp));
  JUMP_LABEL (tmp) = label;

  emit_move_insn (operands[0], operands[1]);
  emit_insn (gen_ashrsi3_31 (operands[1], operands[1], GEN_INT (31)));

  emit_label (label);
  LABEL_NUSES (label) = 1;

  DONE;
})

(define_insn "ashrsi3_31"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=*d,rm")
(ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "*a,0")
     (match_operand:SI 2 "const_int_operand" "i,i")))
   (clobber (reg:CC FLAGS_REG))]
  "INTVAL (operands[2]) == 31 && (TARGET_USE_CLTD || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "@
   {cltd|cdq}
   sar{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "imovx,ishift")
   (set_attr "prefix_0f" "0,*")
   (set_attr "length_immediate" "0,*")
   (set_attr "modrm" "0,1")
   (set_attr "mode" "SI")])

(define_insn "*ashrsi3_31_zext"
  [(set (match_operand:DI 0 "register_operand" "=*d,r")
(zero_extend:DI (ashiftrt:SI (match_operand:SI 1 "register_operand" "*a,0")
     (match_operand:SI 2 "const_int_operand" "i,i"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (TARGET_USE_CLTD || optimize_size)
   && INTVAL (operands[2]) == 31
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
```

```
  "@
   {cltd|cdq}
   sar{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "imovx,ishift")
   (set_attr "prefix_0f" "0,*")
   (set_attr "length_immediate" "0,*")
   (set_attr "modrm" "0,1")
   (set_attr "mode" "SI")])

(define_expand "ashrsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (ASHIFTRT, SImode, operands); DONE;")

(define_insn "*ashrsi3_1_one_bit"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, SImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{l}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*ashrsi3_1_one_bit_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (ashiftrt:SI (match_operand:SI 1 "register_operand" "0")
     (match_operand:QI 2 "const1_operand" ""))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{l}\t%k0"
  [(set_attr "type" "ishift")
   (set_attr "length" "2")])

(define_insn "*ashrsi3_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,rm")
(ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "@
   sar{l}\t{%2, %0|%0, %2}
```

```
   sar{l}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_insn "*ashrsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI (ashiftrt:SI (match_operand:SI 1 "register_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "@
   sar{l}\t{%2, %k0|%k0, %2}
   sar{l}\t{%b2, %k0|%k0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrsi3_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(ashiftrt:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "sar{l}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*ashrsi3_one_bit_cmp_zext"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:SI (match_operand:SI 1 "register_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (ashiftrt:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "sar{l}\t%k0"
  [(set_attr "type" "ishift")
```

```
   (set_attr "length" "2")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrsi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(ashiftrt:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "sar{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_insn "*ashrsi3_cmp_zext"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:SI (match_operand:SI 1 "register_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI (ashiftrt:SI (match_dup 1) (match_dup 2))))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (ASHIFTRT, SImode, operands)"
  "sar{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_expand "ashrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(ashiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (ASHIFTRT, HImode, operands); DONE;")

(define_insn "*ashrhi3_1_one_bit"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(ashiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{w}\t%0"
  [(set_attr "type" "ishift")
```

```
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*ashrhi3_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,rm")
(ashiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, HImode, operands)"
  "@
   sar{w}\t{%2, %0|%0, %2}
   sar{w}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "HI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrhi3_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(ashiftrt:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, HImode, operands)"
  "sar{w}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrhi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(ashiftrt:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
```

```
   && ix86_binary_operator_ok (ASHIFTRT, HImode, operands)"
  "sar{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "HI")])

(define_expand "ashrqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(ashiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (ASHIFTRT, QImode, operands); DONE;")

(define_insn "*ashrqi3_1_one_bit"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(ashiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, QImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{b}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*ashrqi3_1_one_bit_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm"))
(ashiftrt:QI (match_dup 0)
     (match_operand:QI 1 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, QImode, operands)
   && (! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (TARGET_SHIFT1 || optimize_size)"
  "sar{b}\t%0"
  [(set_attr "type" "ishift1")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*ashrqi3_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,qm")
(ashiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ASHIFTRT, QImode, operands)"
  "@
   sar{b}\t{%2, %0|%0, %2}
```

```
   sar{b}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "QI")])

(define_insn "*ashrqi3_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,qm"))
(ashiftrt:QI (match_dup 0)
     (match_operand:QI 1 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
   sar{b}\t{%1, %0|%0, %1}
   sar{b}\t{%b1, %0|%0, %b1}"
  [(set_attr "type" "ishift1")
   (set_attr "mode" "QI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrqi3_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" "I"))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(ashiftrt:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (ASHIFTRT, QImode, operands)"
  "sar{b}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*ashrqi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (ashiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(ashiftrt:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
```

```
   && ix86_binary_operator_ok (ASHIFTRT, QImode, operands)"
  "sar{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "QI")])

;; Logical shift instructions

;; See comment above `ashldi3' about how this works.

(define_expand "lshrti3"
  [(parallel [(set (match_operand:TI 0 "register_operand" "")
   (lshiftrt:TI (match_operand:TI 1 "register_operand" "")
        (match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
{
  if (! immediate_operand (operands[2], QImode))
    {
      emit_insn (gen_lshrti3_1 (operands[0], operands[1], operands[2]));
      DONE;
    }
  ix86_expand_binary_operator (LSHIFTRT, TImode, operands);
  DONE;
})

(define_insn "lshrti3_1"
  [(set (match_operand:TI 0 "register_operand" "=r")
(lshiftrt:TI (match_operand:TI 1 "register_operand" "0")
     (match_operand:QI 2 "register_operand" "c")))
   (clobber (match_scratch:DI 3 "=&r"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_insn "*lshrti3_2"
  [(set (match_operand:TI 0 "register_operand" "=r")
(lshiftrt:TI (match_operand:TI 1 "register_operand" "0")
     (match_operand:QI 2 "immediate_operand" "O")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(lshiftrt:TI (match_operand:TI 1 "register_operand" "")
     (match_operand:QI 2 "register_operand" "")))
   (clobber (match_scratch:DI 3 ""))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
  "ix86_split_lshr (operands, operands[3], TImode); DONE;")

(define_split
  [(set (match_operand:TI 0 "register_operand" "")
(lshiftrt:TI (match_operand:TI 1 "register_operand" "")
     (match_operand:QI 2 "immediate_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && reload_completed"
  [(const_int 0)]
  "ix86_split_lshr (operands, NULL_RTX, TImode); DONE;")

(define_expand "lshrdi3"
  [(set (match_operand:DI 0 "shiftdi_operand" "")
(lshiftrt:DI (match_operand:DI 1 "shiftdi_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))]
  ""
  "ix86_expand_binary_operator (LSHIFTRT, DImode, operands); DONE;")

(define_insn "*lshrdi3_1_one_bit_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(lshiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{q}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrdi3_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,rm")
(lshiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "J,c")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "@
   shr{q}\t{%2, %0|%0, %2}
   shr{q}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "DI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrdi3_cmp_one_bit_rex64"
```

```
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(lshiftrt:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{q}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrdi3_cmp_rex64"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:DI (match_operand:DI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_int_operand" "e"))
  (const_int 0)))
   (set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(lshiftrt:DI (match_dup 1) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "DI")])

(define_insn "*lshrdi3_1"
  [(set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (match_operand:DI 1 "register_operand" "0")
     (match_operand:QI 2 "nonmemory_operand" "Jc")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "#"
  [(set_attr "type" "multi")])

;; By default we don't ask for a scratch register, because when DImode
;; values are manipulated, registers are already at a premium.  But if
;; we have one handy, we won't turn it away.
(define_peephole2
  [(match_scratch:SI 3 "r")
   (parallel [(set (match_operand:DI 0 "register_operand" "")
   (lshiftrt:DI (match_operand:DI 1 "register_operand" "")
```

```
        (match_operand:QI 2 "nonmemory_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_dup 3)]
  "!TARGET_64BIT && TARGET_CMOVE"
  [(const_int 0)]
  "ix86_split_lshr (operands, operands[3], DImode); DONE;")

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(lshiftrt:DI (match_operand:DI 1 "register_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && (flag_peephole2 ? flow2_completed : reload_completed)"
  [(const_int 0)]
  "ix86_split_lshr (operands, NULL_RTX, DImode); DONE;")

(define_expand "lshrsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (LSHIFTRT, SImode, operands); DONE;")

(define_insn "*lshrsi3_1_one_bit"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{l}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrsi3_1_one_bit_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (zero_extend:DI (match_operand:SI 1 "register_operand" "0"))
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{l}\t%k0"
  [(set_attr "type" "ishift")
   (set_attr "length" "2")])

(define_insn "*lshrsi3_1"
```

```
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,rm")
(lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "@
   shr{l}\t{%2, %0|%0, %2}
   shr{l}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_insn "*lshrsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI
  (lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
       (match_operand:QI 2 "nonmemory_operand" "I,c"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "@
   shr{l}\t{%2, %k0|%k0, %2}
   shr{l}\t{%b2, %k0|%k0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrsi3_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(lshiftrt:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{l}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrsi3_cmp_one_bit_zext"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:SI (match_operand:SI 1 "register_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
```

```
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (zero_extend:DI (match_dup 1)) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{l}\t%k0"
  [(set_attr "type" "ishift")
   (set_attr "length" "2")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrsi3_cmp"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:SI (match_operand:SI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(lshiftrt:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_insn "*lshrsi3_cmp_zext"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:SI (match_operand:SI 1 "register_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(lshiftrt:DI (zero_extend:DI (match_dup 1)) (match_dup 2)))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{l}\t{%2, %k0|%k0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "SI")])

(define_expand "lshrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(lshiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (LSHIFTRT, HImode, operands); DONE;")

(define_insn "*lshrhi3_1_one_bit"
```

```
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(lshiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{w}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrhi3_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,rm")
(lshiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "@
   shr{w}\t{%2, %0|%0, %2}
   shr{w}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "HI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrhi3_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(lshiftrt:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{w}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrhi3_cmp"
```

```
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:HI (match_operand:HI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(lshiftrt:HI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (LSHIFTRT, HImode, operands)"
  "shr{w}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "HI")])

(define_expand "lshrqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(lshiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (LSHIFTRT, QImode, operands); DONE;")

(define_insn "*lshrqi3_1_one_bit"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(lshiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, QImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{b}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrqi3_1_one_bit_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm"))
(lshiftrt:QI (match_dup 0)
     (match_operand:QI 1 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (TARGET_SHIFT1 || optimize_size)"
  "shr{b}\t%0"
  [(set_attr "type" "ishift1")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*lshrqi3_1"
```

```
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,qm")
(lshiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (LSHIFTRT, QImode, operands)"
  "@
   shr{b}\t{%2, %0|%0, %2}
   shr{b}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "QI")])

(define_insn "*lshrqi3_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,qm"))
(lshiftrt:QI (match_dup 0)
     (match_operand:QI 1 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
   shr{b}\t{%1, %0|%0, %1}
   shr{b}\t{%b1, %0|%0, %b1}"
  [(set_attr "type" "ishift1")
   (set_attr "mode" "QI")])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrqi2_one_bit_cmp"
  [(set (reg FLAGS_REG)
(compare
  (lshiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(lshiftrt:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && (TARGET_SHIFT1 || optimize_size)
   && ix86_binary_operator_ok (LSHIFTRT, QImode, operands)"
  "shr{b}\t%0"
  [(set_attr "type" "ishift")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

;; This pattern can't accept a variable shift count, since shifts by
;; zero don't affect the flags.  We assume that shifts by constant
;; zero are optimized away.
(define_insn "*lshrqi2_cmp"
  [(set (reg FLAGS_REG)
```

```
(compare
  (lshiftrt:QI (match_operand:QI 1 "nonimmediate_operand" "0")
       (match_operand:QI 2 "const_1_to_31_operand" "I"))
  (const_int 0)))
   (set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(lshiftrt:QI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCGOCmode)
   && ix86_binary_operator_ok (LSHIFTRT, QImode, operands)"
  "shr{b}\t{%2, %0|%0, %2}"
  [(set_attr "type" "ishift")
   (set_attr "mode" "QI")])

;; Rotate instructions

(define_expand "rotldi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(rotate:DI (match_operand:DI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (ROTATE, DImode, operands); DONE;")

(define_insn "*rotlsi3_1_one_bit_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(rotate:DI (match_operand:DI 1 "nonimmediate_operand" "0")
   (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATE, DImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{q}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotldi3_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,rm")
(rotate:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
   (match_operand:QI 2 "nonmemory_operand" "e,c")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATE, DImode, operands)"
  "@
   rol{q}\t{%2, %0|%0, %2}
   rol{q}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "DI")])

(define_expand "rotlsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
```

```
(rotate:SI (match_operand:SI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (ROTATE, SImode, operands); DONE;")

(define_insn "*rotlsi3_1_one_bit"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(rotate:SI (match_operand:SI 1 "nonimmediate_operand" "0")
   (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, SImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{l}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotlsi3_1_one_bit_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (rotate:SI (match_operand:SI 1 "register_operand" "0")
     (match_operand:QI 2 "const1_operand" ""))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATE, SImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{l}\t%k0"
  [(set_attr "type" "rotate")
   (set_attr "length" "2")])

(define_insn "*rotlsi3_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,rm")
(rotate:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
   (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, SImode, operands)"
  "@
   rol{l}\t{%2, %0|%0, %2}
   rol{l}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "SI")])

(define_insn "*rotlsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI
  (rotate:SI (match_operand:SI 1 "register_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c"))))
   (clobber (reg:CC FLAGS_REG))]
```

```
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATE, SImode, operands)"
  "@
   rol{l}\t{%2, %k0|%k0, %2}
   rol{l}\t{%b2, %k0|%k0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "SI")])

(define_expand "rotlhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(rotate:HI (match_operand:HI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (ROTATE, HImode, operands); DONE;")

(define_insn "*rotlhi3_1_one_bit"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(rotate:HI (match_operand:HI 1 "nonimmediate_operand" "0")
   (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{w}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotlhi3_1"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,rm")
(rotate:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
   (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, HImode, operands)"
  "@
   rol{w}\t{%2, %0|%0, %2}
   rol{w}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "HI")])

(define_expand "rotlqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(rotate:QI (match_operand:QI 1 "nonimmediate_operand" "")
   (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (ROTATE, QImode, operands); DONE;")

(define_insn "*rotlqi3_1_one_bit_slp"
```

```
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm"))
(rotate:QI (match_dup 0)
   (match_operand:QI 1 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{b}\t%0"
  [(set_attr "type" "rotate1")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotlqi3_1_one_bit"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(rotate:QI (match_operand:QI 1 "nonimmediate_operand" "0")
   (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, QImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "rol{b}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotlqi3_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,qm"))
(rotate:QI (match_dup 0)
   (match_operand:QI 1 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
   rol{b}\t{%1, %0|%0, %1}
   rol{b}\t{%b1, %0|%0, %b1}"
  [(set_attr "type" "rotate1")
   (set_attr "mode" "QI")])

(define_insn "*rotlqi3_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,qm")
(rotate:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
   (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATE, QImode, operands)"
  "@
   rol{b}\t{%2, %0|%0, %2}
   rol{b}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
```

```
   (set_attr "mode" "QI")])

(define_expand "rotrdi3"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(rotatert:DI (match_operand:DI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "ix86_expand_binary_operator (ROTATERT, DImode, operands); DONE;")

(define_insn "*rotrdi3_1_one_bit_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm")
(rotatert:DI (match_operand:DI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATERT, DImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{q}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand:DI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrdi3_1_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,rm")
(rotatert:DI (match_operand:DI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "J,c")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATERT, DImode, operands)"
  "@
   ror{q}\t{%2, %0|%0, %2}
   ror{q}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "DI")])

(define_expand "rotrsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(rotatert:SI (match_operand:SI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "ix86_expand_binary_operator (ROTATERT, SImode, operands); DONE;")

(define_insn "*rotrsi3_1_one_bit"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm")
(rotatert:SI (match_operand:SI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, SImode, operands)
```

```
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{l}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrsi3_1_one_bit_zext"
  [(set (match_operand:DI 0 "register_operand" "=r")
(zero_extend:DI
  (rotatert:SI (match_operand:SI 1 "register_operand" "0")
       (match_operand:QI 2 "const1_operand" ""))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATERT, SImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{l}\t%k0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand:SI 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrsi3_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=rm,rm")
(rotatert:SI (match_operand:SI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, SImode, operands)"
  "@
   ror{l}\t{%2, %0|%0, %2}
   ror{l}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "SI")])

(define_insn "*rotrsi3_1_zext"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(zero_extend:DI
  (rotatert:SI (match_operand:SI 1 "register_operand" "0,0")
       (match_operand:QI 2 "nonmemory_operand" "I,c"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (ROTATERT, SImode, operands)"
  "@
   ror{l}\t{%2, %k0|%k0, %2}
   ror{l}\t{%b2, %k0|%k0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "SI")])

(define_expand "rotrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
```

```
(rotatert:HI (match_operand:HI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_HIMODE_MATH"
  "ix86_expand_binary_operator (ROTATERT, HImode, operands); DONE;")

(define_insn "*rotrhi3_one_bit"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm")
(rotatert:HI (match_operand:HI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, HImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{w}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "=rm,rm")
(rotatert:HI (match_operand:HI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, HImode, operands)"
  "@
   ror{w}\t{%2, %0|%0, %2}
   ror{w}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "HI")])

(define_expand "rotrqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(rotatert:QI (match_operand:QI 1 "nonimmediate_operand" "")
     (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_QIMODE_MATH"
  "ix86_expand_binary_operator (ROTATERT, QImode, operands); DONE;")

(define_insn "*rotrqi3_1_one_bit"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(rotatert:QI (match_operand:QI 1 "nonimmediate_operand" "0")
     (match_operand:QI 2 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, QImode, operands)
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{b}\t%0"
  [(set_attr "type" "rotate")
   (set (attr "length")
```

```
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrqi3_1_one_bit_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm"))
(rotatert:QI (match_dup 0)
     (match_operand:QI 1 "const1_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (TARGET_SHIFT1 || optimize_size)"
  "ror{b}\t%0"
  [(set_attr "type" "rotate1")
   (set (attr "length")
     (if_then_else (match_operand 0 "register_operand" "")
(const_string "2")
(const_string "*")))])

(define_insn "*rotrqi3_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm,qm")
(rotatert:QI (match_operand:QI 1 "nonimmediate_operand" "0,0")
     (match_operand:QI 2 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "ix86_binary_operator_ok (ROTATERT, QImode, operands)"
  "@
   ror{b}\t{%2, %0|%0, %2}
   ror{b}\t{%b2, %0|%0, %b2}"
  [(set_attr "type" "rotate")
   (set_attr "mode" "QI")])

(define_insn "*rotrqi3_1_slp"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm,qm"))
(rotatert:QI (match_dup 0)
     (match_operand:QI 1 "nonmemory_operand" "I,c")))
   (clobber (reg:CC FLAGS_REG))]
  "(! TARGET_PARTIAL_REG_STALL || optimize_size)
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
   ror{b}\t{%1, %0|%0, %1}
   ror{b}\t{%b1, %0|%0, %b1}"
  [(set_attr "type" "rotate1")
   (set_attr "mode" "QI")])

;; Bit set / bit test instructions

(define_expand "extv"
  [(set (match_operand:SI 0 "register_operand" "")
(sign_extract:SI (match_operand:SI 1 "register_operand" "")
 (match_operand:SI 2 "immediate_operand" "")
 (match_operand:SI 3 "immediate_operand" "")))]
```

```
  ""
{
  /* Handle extractions from %ah et al.  */
  if (INTVAL (operands[2]) != 8 || INTVAL (operands[3]) != 8)
    FAIL;

  /* From mips.md: extract_bit_field doesn't verify that our source
     matches the predicate, so check it again here.  */
  if (! ext_register_operand (operands[1], VOIDmode))
    FAIL;
})

(define_expand "extzv"
  [(set (match_operand:SI 0 "register_operand" "")
(zero_extract:SI (match_operand 1 "ext_register_operand" "")
 (match_operand:SI 2 "immediate_operand" "")
 (match_operand:SI 3 "immediate_operand" "")))]
  ""
{
  /* Handle extractions from %ah et al.  */
  if (INTVAL (operands[2]) != 8 || INTVAL (operands[3]) != 8)
    FAIL;

  /* From mips.md: extract_bit_field doesn't verify that our source
     matches the predicate, so check it again here.  */
  if (! ext_register_operand (operands[1], VOIDmode))
    FAIL;
})

(define_expand "insv"
  [(set (zero_extract (match_operand 0 "ext_register_operand" "")
      (match_operand 1 "immediate_operand" "")
      (match_operand 2 "immediate_operand" ""))
        (match_operand 3 "register_operand" ""))]
  ""
{
  /* Handle extractions from %ah et al.  */
  if (INTVAL (operands[1]) != 8 || INTVAL (operands[2]) != 8)
    FAIL;

  /* From mips.md: insert_bit_field doesn't verify that our source
     matches the predicate, so check it again here.  */
  if (! ext_register_operand (operands[0], VOIDmode))
    FAIL;

  if (TARGET_64BIT)
    emit_insn (gen_movdi_insv_1_rex64 (operands[0], operands[3]));
  else
    emit_insn (gen_movsi_insv_1 (operands[0], operands[3]));
```

```
  DONE;
})

;; %%% bts, btr, btc, bt.
;; In general these instructions are *slow* when applied to memory,
;; since they enforce atomic operation.  When applied to registers,
;; it depends on the cpu implementation.  They're never faster than
;; the corresponding and/ior/xor operations, so with 32-bit there's
;; no point.  But in 64-bit, we can't hold the relevant immediates
;; within the instruction itself, so operating on bits in the high
;; 32-bits of a register becomes easier.
;;
;; These are slow on Nocona, but fast on Athlon64.  We do require the use
;; of btrq and btcq for corner cases of post-reload expansion of absdf and
;; negdf respectively, so they can never be disabled entirely.

(define_insn "*btsq"
  [(set (zero_extract:DI (match_operand:DI 0 "register_operand" "+r")
 (const_int 1)
 (match_operand:DI 1 "const_0_to_63_operand" ""))
(const_int 1))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (TARGET_USE_BT || reload_completed)"
  "bts{q} %1,%0"
  [(set_attr "type" "alu1")])

(define_insn "*btrq"
  [(set (zero_extract:DI (match_operand:DI 0 "register_operand" "+r")
 (const_int 1)
 (match_operand:DI 1 "const_0_to_63_operand" ""))
(const_int 0))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (TARGET_USE_BT || reload_completed)"
  "btr{q} %1,%0"
  [(set_attr "type" "alu1")])

(define_insn "*btcq"
  [(set (zero_extract:DI (match_operand:DI 0 "register_operand" "+r")
 (const_int 1)
 (match_operand:DI 1 "const_0_to_63_operand" ""))
(not:DI (zero_extract:DI (match_dup 0) (const_int 1) (match_dup 1))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && (TARGET_USE_BT || reload_completed)"
  "btc{q} %1,%0"
  [(set_attr "type" "alu1")])

;; Allow Nocona to avoid these instructions if a register is available.

(define_peephole2
  [(match_scratch:DI 2 "r")
```

```
   (parallel [(set (zero_extract:DI
     (match_operand:DI 0 "register_operand" "")
     (const_int 1)
     (match_operand:DI 1 "const_0_to_63_operand" ""))
   (const_int 1))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT && !TARGET_USE_BT"
  [(const_int 0)]
{
  HOST_WIDE_INT i = INTVAL (operands[1]), hi, lo;
  rtx op1;

  if (HOST_BITS_PER_WIDE_INT >= 64)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else if (i < HOST_BITS_PER_WIDE_INT)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else
    lo = 0, hi = (HOST_WIDE_INT)1 << (i - HOST_BITS_PER_WIDE_INT);

  op1 = immed_double_const (lo, hi, DImode);
  if (i >= 31)
    {
      emit_move_insn (operands[2], op1);
      op1 = operands[2];
    }

  emit_insn (gen_iordi3 (operands[0], operands[0], op1));
  DONE;
})

(define_peephole2
  [(match_scratch:DI 2 "r")
   (parallel [(set (zero_extract:DI
     (match_operand:DI 0 "register_operand" "")
     (const_int 1)
     (match_operand:DI 1 "const_0_to_63_operand" ""))
   (const_int 0))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT && !TARGET_USE_BT"
  [(const_int 0)]
{
  HOST_WIDE_INT i = INTVAL (operands[1]), hi, lo;
  rtx op1;

  if (HOST_BITS_PER_WIDE_INT >= 64)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else if (i < HOST_BITS_PER_WIDE_INT)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else
    lo = 0, hi = (HOST_WIDE_INT)1 << (i - HOST_BITS_PER_WIDE_INT);
```

```
  op1 = immed_double_const (~lo, ~hi, DImode);
  if (i >= 32)
    {
      emit_move_insn (operands[2], op1);
      op1 = operands[2];
    }

  emit_insn (gen_anddi3 (operands[0], operands[0], op1));
  DONE;
})

(define_peephole2
  [(match_scratch:DI 2 "r")
   (parallel [(set (zero_extract:DI
     (match_operand:DI 0 "register_operand" "")
     (const_int 1)
     (match_operand:DI 1 "const_0_to_63_operand" ""))
      (not:DI (zero_extract:DI
(match_dup 0) (const_int 1) (match_dup 1))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT && !TARGET_USE_BT"
  [(const_int 0)]
{
  HOST_WIDE_INT i = INTVAL (operands[1]), hi, lo;
  rtx op1;

  if (HOST_BITS_PER_WIDE_INT >= 64)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else if (i < HOST_BITS_PER_WIDE_INT)
    lo = (HOST_WIDE_INT)1 << i, hi = 0;
  else
    lo = 0, hi = (HOST_WIDE_INT)1 << (i - HOST_BITS_PER_WIDE_INT);

  op1 = immed_double_const (lo, hi, DImode);
  if (i >= 31)
    {
      emit_move_insn (operands[2], op1);
      op1 = operands[2];
    }

  emit_insn (gen_xordi3 (operands[0], operands[0], op1));
  DONE;
})

;; Store-flag instructions.

;; For all sCOND expanders, also expand the compare or test insn that
;; generates cc0.  Generate an equality comparison if `seq' or `sne'.
```

```
;; %%% Do the expansion to SImode.  If PII, do things the xor+setcc way
;; to avoid partial register stalls.  Otherwise do things the setcc+movzx
;; way, which can later delete the movzx if only QImode is needed.

(define_expand "seq"
  [(set (match_operand:QI 0 "register_operand" "")
        (eq:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (EQ, operands[0])) DONE; else FAIL;")

(define_expand "sne"
  [(set (match_operand:QI 0 "register_operand" "")
        (ne:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (NE, operands[0])) DONE; else FAIL;")

(define_expand "sgt"
  [(set (match_operand:QI 0 "register_operand" "")
        (gt:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (GT, operands[0])) DONE; else FAIL;")

(define_expand "sgtu"
  [(set (match_operand:QI 0 "register_operand" "")
        (gtu:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (GTU, operands[0])) DONE; else FAIL;")

(define_expand "slt"
  [(set (match_operand:QI 0 "register_operand" "")
        (lt:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (LT, operands[0])) DONE; else FAIL;")

(define_expand "sltu"
  [(set (match_operand:QI 0 "register_operand" "")
        (ltu:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (LTU, operands[0])) DONE; else FAIL;")

(define_expand "sge"
  [(set (match_operand:QI 0 "register_operand" "")
        (ge:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (GE, operands[0])) DONE; else FAIL;")

(define_expand "sgeu"
  [(set (match_operand:QI 0 "register_operand" "")
        (geu:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
```

```
  "if (ix86_expand_setcc (GEU, operands[0])) DONE; else FAIL;")

(define_expand "sle"
  [(set (match_operand:QI 0 "register_operand" "")
        (le:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (LE, operands[0])) DONE; else FAIL;")

(define_expand "sleu"
  [(set (match_operand:QI 0 "register_operand" "")
        (leu:QI (reg:CC FLAGS_REG) (const_int 0)))]
  ""
  "if (ix86_expand_setcc (LEU, operands[0])) DONE; else FAIL;")

(define_expand "sunordered"
  [(set (match_operand:QI 0 "register_operand" "")
        (unordered:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNORDERED, operands[0])) DONE; else FAIL;")

(define_expand "sordered"
  [(set (match_operand:QI 0 "register_operand" "")
        (ordered:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387"
  "if (ix86_expand_setcc (ORDERED, operands[0])) DONE; else FAIL;")

(define_expand "suneq"
  [(set (match_operand:QI 0 "register_operand" "")
        (uneq:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNEQ, operands[0])) DONE; else FAIL;")

(define_expand "sunge"
  [(set (match_operand:QI 0 "register_operand" "")
        (unge:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNGE, operands[0])) DONE; else FAIL;")

(define_expand "sungt"
  [(set (match_operand:QI 0 "register_operand" "")
        (ungt:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNGT, operands[0])) DONE; else FAIL;")

(define_expand "sunle"
  [(set (match_operand:QI 0 "register_operand" "")
        (unle:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNLE, operands[0])) DONE; else FAIL;")
```

```
(define_expand "sunlt"
  [(set (match_operand:QI 0 "register_operand" "")
        (unlt:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (UNLT, operands[0])) DONE; else FAIL;")

(define_expand "sltgt"
  [(set (match_operand:QI 0 "register_operand" "")
        (ltgt:QI (reg:CC FLAGS_REG) (const_int 0)))]
  "TARGET_80387 || TARGET_SSE"
  "if (ix86_expand_setcc (LTGT, operands[0])) DONE; else FAIL;")

(define_insn "*setcc_1"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=qm")
(match_operator:QI 1 "ix86_comparison_operator"
  [(reg FLAGS_REG) (const_int 0)]))]
  ""
  "set%C1\t%0"
  [(set_attr "type" "setcc")
   (set_attr "mode" "QI")])

(define_insn "*setcc_2"
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" "+qm"))
(match_operator:QI 1 "ix86_comparison_operator"
  [(reg FLAGS_REG) (const_int 0)]))]
  ""
  "set%C1\t%0"
  [(set_attr "type" "setcc")
   (set_attr "mode" "QI")])

;; In general it is not safe to assume too much about CCmode registers,
;; so simplify-rtx stops when it sees a second one.  Under certain
;; conditions this is safe on x86, so help combine not create
;;
;; seta %al
;; testb %al, %al
;; sete %al

(define_split
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(ne:QI (match_operator 1 "ix86_comparison_operator"
         [(reg FLAGS_REG) (const_int 0)])
    (const_int 0)))]
  ""
  [(set (match_dup 0) (match_dup 1))]
{
  PUT_MODE (operands[1], QImode);
})

(define_split
```

```
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" ""))
(ne:QI (match_operator 1 "ix86_comparison_operator"
         [(reg FLAGS_REG) (const_int 0)])
    (const_int 0)))]
  ""
  [(set (match_dup 0) (match_dup 1))]
{
  PUT_MODE (operands[1], QImode);
})

(define_split
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(eq:QI (match_operator 1 "ix86_comparison_operator"
         [(reg FLAGS_REG) (const_int 0)])
    (const_int 0)))]
  ""
  [(set (match_dup 0) (match_dup 1))]
{
  rtx new_op1 = copy_rtx (operands[1]);
  operands[1] = new_op1;
  PUT_MODE (new_op1, QImode);
  PUT_CODE (new_op1, ix86_reverse_condition (GET_CODE (new_op1),
     GET_MODE (XEXP (new_op1, 0))));

  /* Make sure that (a) the CCmode we have for the flags is strong
     enough for the reversed compare or (b) we have a valid FP compare.  */
  if (! ix86_comparison_operator (new_op1, VOIDmode))
    FAIL;
})

(define_split
  [(set (strict_low_part (match_operand:QI 0 "nonimmediate_operand" ""))
(eq:QI (match_operator 1 "ix86_comparison_operator"
         [(reg FLAGS_REG) (const_int 0)])
    (const_int 0)))]
  ""
  [(set (match_dup 0) (match_dup 1))]
{
  rtx new_op1 = copy_rtx (operands[1]);
  operands[1] = new_op1;
  PUT_MODE (new_op1, QImode);
  PUT_CODE (new_op1, ix86_reverse_condition (GET_CODE (new_op1),
     GET_MODE (XEXP (new_op1, 0))));

  /* Make sure that (a) the CCmode we have for the flags is strong
     enough for the reversed compare or (b) we have a valid FP compare.  */
  if (! ix86_comparison_operator (new_op1, VOIDmode))
    FAIL;
})
```

```
;; The SSE store flag instructions saves 0 or 0xffffffff to the result.
;; subsequent logical operations are used to imitate conditional moves.
;; 0xffffffff is NaN, but not in normalized form, so we can't represent
;; it directly.

(define_insn "*sse_setccsf"
  [(set (match_operand:SF 0 "register_operand" "=x")
(match_operator:SF 1 "sse_comparison_operator"
  [(match_operand:SF 2 "register_operand" "0")
   (match_operand:SF 3 "nonimmediate_operand" "xm")]))]
  "TARGET_SSE"
  "cmp%D1ss\t{%3, %0|%0, %3}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "SF")])

(define_insn "*sse_setccdf"
  [(set (match_operand:DF 0 "register_operand" "=Y")
(match_operator:DF 1 "sse_comparison_operator"
  [(match_operand:DF 2 "register_operand" "0")
   (match_operand:DF 3 "nonimmediate_operand" "Ym")]))]
  "TARGET_SSE2"
  "cmp%D1sd\t{%3, %0|%0, %3}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "DF")])

;; Basic conditional jump instructions.
;; We ignore the overflow flag for signed branch instructions.

;; For all bCOND expanders, also expand the compare or test insn that
;; generates reg FLAGS_REG.  Generate an equality comparison if `beq' or `bne'.

(define_expand "beq"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (EQ, operands[0]); DONE;")

(define_expand "bne"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (NE, operands[0]); DONE;")

(define_expand "bgt"
  [(set (pc)
(if_then_else (match_dup 1)
```

```
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (GT, operands[0]); DONE;")

(define_expand "bgtu"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (GTU, operands[0]); DONE;")

(define_expand "blt"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (LT, operands[0]); DONE;")

(define_expand "bltu"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (LTU, operands[0]); DONE;")

(define_expand "bge"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (GE, operands[0]); DONE;")

(define_expand "bgeu"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (GEU, operands[0]); DONE;")

(define_expand "ble"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
```

```
  ""
  "ix86_expand_branch (LE, operands[0]); DONE;")

(define_expand "bleu"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "ix86_expand_branch (LEU, operands[0]); DONE;")

(define_expand "bunordered"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNORDERED, operands[0]); DONE;")

(define_expand "bordered"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (ORDERED, operands[0]); DONE;")

(define_expand "buneq"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNEQ, operands[0]); DONE;")

(define_expand "bunge"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNGE, operands[0]); DONE;")

(define_expand "bungt"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNGT, operands[0]); DONE;")
```

```
(define_expand "bunle"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNLE, operands[0]); DONE;")

(define_expand "bunlt"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (UNLT, operands[0]); DONE;")

(define_expand "bltgt"
  [(set (pc)
(if_then_else (match_dup 1)
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  "TARGET_80387 || TARGET_SSE_MATH"
  "ix86_expand_branch (LTGT, operands[0]); DONE;")

(define_insn "*jcc_1"
  [(set (pc)
(if_then_else (match_operator 1 "ix86_comparison_operator"
      [(reg FLAGS_REG) (const_int 0)])
      (label_ref (match_operand 0 "" ""))
      (pc)))]
  ""
  "%+j%C1\t%l0"
  [(set_attr "type" "ibr")
   (set_attr "modrm" "0")
   (set (attr "length")
   (if_then_else (and (ge (minus (match_dup 0) (pc))
  (const_int -126))
      (lt (minus (match_dup 0) (pc))
  (const_int 128)))
     (const_int 2)
     (const_int 6)))])

(define_insn "*jcc_2"
  [(set (pc)
(if_then_else (match_operator 1 "ix86_comparison_operator"
      [(reg FLAGS_REG) (const_int 0)])
      (pc)
      (label_ref (match_operand 0 "" ""))))]
  ""
```

```
  "%+j%c1\t%l0"
  [(set_attr "type" "ibr")
   (set_attr "modrm" "0")
   (set (attr "length")
   (if_then_else (and (ge (minus (match_dup 0) (pc))
  (const_int -126))
      (lt (minus (match_dup 0) (pc))
  (const_int 128)))
     (const_int 2)
     (const_int 6)))])

;; In general it is not safe to assume too much about CCmode registers,
;; so simplify-rtx stops when it sees a second one.  Under certain
;; conditions this is safe on x86, so help combine not create
;;
;; seta %al
;; testb %al, %al
;; je Lfoo

(define_split
  [(set (pc)
(if_then_else (ne (match_operator 0 "ix86_comparison_operator"
      [(reg FLAGS_REG) (const_int 0)])
  (const_int 0))
      (label_ref (match_operand 1 "" ""))
      (pc)))]
  ""
  [(set (pc)
(if_then_else (match_dup 0)
      (label_ref (match_dup 1))
      (pc)))]
{
  PUT_MODE (operands[0], VOIDmode);
})

(define_split
  [(set (pc)
(if_then_else (eq (match_operator 0 "ix86_comparison_operator"
      [(reg FLAGS_REG) (const_int 0)])
  (const_int 0))
      (label_ref (match_operand 1 "" ""))
      (pc)))]
  ""
  [(set (pc)
(if_then_else (match_dup 0)
      (label_ref (match_dup 1))
      (pc)))]
{
  rtx new_op0 = copy_rtx (operands[0]);
  operands[0] = new_op0;
```

```
  PUT_MODE (new_op0, VOIDmode);
  PUT_CODE (new_op0, ix86_reverse_condition (GET_CODE (new_op0),
     GET_MODE (XEXP (new_op0, 0))));

  /* Make sure that (a) the CCmode we have for the flags is strong
     enough for the reversed compare or (b) we have a valid FP compare.  */
  if (! ix86_comparison_operator (new_op0, VOIDmode))
    FAIL;
})

;; Define combination compare-and-branch fp compare instructions to use
;; during early optimization.  Splitting the operation apart early makes
;; for bad code when we want to reverse the operation.

(define_insn "*fp_jcc_1_mixed"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f#x,x#f")
 (match_operand 2 "nonimmediate_operand" "f#x,xm#f")])
  (label_ref (match_operand 3 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_MIX_SSE_I387
   && SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_1_sse"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "x")
 (match_operand 2 "nonimmediate_operand" "xm")])
  (label_ref (match_operand 3 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_SSE_MATH
   && SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_1_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "register_operand" "f")])
  (label_ref (match_operand 3 "" ""))
```

```
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_CMOVE && TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_2_mixed"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f#x,x#f")
 (match_operand 2 "nonimmediate_operand" "f#x,xm#f")])
  (pc)
  (label_ref (match_operand 3 "" ""))))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_MIX_SSE_I387
   && SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_2_sse"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "x")
 (match_operand 2 "nonimmediate_operand" "xm")])
  (pc)
  (label_ref (match_operand 3 "" ""))))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_SSE_MATH
   && SSE_FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_2_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "register_operand" "f")])
  (pc)
  (label_ref (match_operand 3 "" ""))))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "TARGET_CMOVE && TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
```

```
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_3_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "nonimmediate_operand" "fm")])
  (label_ref (match_operand 3 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 4 "=a"))]
  "TARGET_80387
   && (GET_MODE (operands[1]) == SFmode || GET_MODE (operands[1]) == DFmode)
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && !ix86_use_fcomi_compare (GET_CODE (operands[0]))
   && SELECT_CC_MODE (GET_CODE (operands[0]),
      operands[1], operands[2]) == CCFPmode
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_4_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "nonimmediate_operand" "fm")])
  (pc)
  (label_ref (match_operand 3 "" ""))))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 4 "=a"))]
  "TARGET_80387
   && (GET_MODE (operands[1]) == SFmode || GET_MODE (operands[1]) == DFmode)
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && !ix86_use_fcomi_compare (GET_CODE (operands[0]))
   && SELECT_CC_MODE (GET_CODE (operands[0]),
      operands[1], operands[2]) == CCFPmode
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_5_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "register_operand" "f")])
  (label_ref (match_operand 3 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
```

```
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 4 "=a"))]
  "TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_6_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "register_operand" "f")])
  (pc)
  (label_ref (match_operand 3 "" ""))))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 4 "=a"))]
  "TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

(define_insn "*fp_jcc_7_387"
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "f")
 (match_operand 2 "const0_operand" "X")])
  (label_ref (match_operand 3 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 4 "=a"))]
  "TARGET_80387
   && FLOAT_MODE_P (GET_MODE (operands[1]))
   && GET_MODE (operands[1]) == GET_MODE (operands[2])
   && !ix86_use_fcomi_compare (GET_CODE (operands[0]))
   && SELECT_CC_MODE (GET_CODE (operands[0]),
      operands[1], operands[2]) == CCFPmode
   && ix86_fp_jump_nontrivial_p (GET_CODE (operands[0]))"
  "#")

;; The order of operands in *fp_jcc_8_387 is forced by combine in
;; simplify_comparison () function. Float operator is treated as RTX_OBJ
;; with a precedence over other operators and is always put in the first
;; place. Swap condition and operands to match ficom instruction.

(define_insn "*fp_jcc_8<mode>_387"
  [(set (pc)
```

```
(if_then_else (match_operator 0 "comparison_operator"
[(match_operator 1 "float_operator"
   [(match_operand:X87MODEI12 2 "nonimmediate_operand" "m,?r")])
   (match_operand 3 "register_operand" "f,f")])
  (label_ref (match_operand 4 "" ""))
  (pc)))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 5 "=a,a"))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP
   && FLOAT_MODE_P (GET_MODE (operands[3]))
   && GET_MODE (operands[1]) == GET_MODE (operands[3])
   && !ix86_use_fcomi_compare (swap_condition (GET_CODE (operands[0])))
   && ix86_fp_compare_mode (swap_condition (GET_CODE (operands[0]))) == CCFPmode
   && ix86_fp_jump_nontrivial_p (swap_condition (GET_CODE (operands[0])))"
  "#")

(define_split
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "")
 (match_operand 2 "nonimmediate_operand" "")])
  (match_operand 3 "" "")
  (match_operand 4 "" "")))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))]
  "reload_completed"
  [(const_int 0)]
{
  ix86_split_fp_branch (GET_CODE (operands[0]), operands[1], operands[2],
                operands[3], operands[4], NULL_RTX, NULL_RTX);
  DONE;
})

(define_split
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operand 1 "register_operand" "")
 (match_operand 2 "general_operand" "")])
  (match_operand 3 "" "")
  (match_operand 4 "" "")))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 5 "=a"))]
  "reload_completed"
  [(const_int 0)]
{
  ix86_split_fp_branch (GET_CODE (operands[0]), operands[1], operands[2],
      operands[3], operands[4], operands[5], NULL_RTX);
  DONE;
```

```
})

(define_split
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operator 1 "float_operator"
   [(match_operand:X87MODEI12 2 "memory_operand" "")])
   (match_operand 3 "register_operand" "")])
  (match_operand 4 "" "")
  (match_operand 5 "" "")))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 6 "=a"))]
  "reload_completed"
  [(const_int 0)]
{
  operands[7] = gen_rtx_FLOAT (GET_MODE (operands[1]), operands[2]);
  ix86_split_fp_branch (swap_condition (GET_CODE (operands[0])),
operands[3], operands[7],
operands[4], operands[5], operands[6], NULL_RTX);
  DONE;
})

;; %%% Kill this when reload knows how to do it.
(define_split
  [(set (pc)
(if_then_else (match_operator 0 "comparison_operator"
[(match_operator 1 "float_operator"
   [(match_operand:X87MODEI12 2 "register_operand" "")])
   (match_operand 3 "register_operand" "")])
  (match_operand 4 "" "")
  (match_operand 5 "" "")))
   (clobber (reg:CCFP FPSR_REG))
   (clobber (reg:CCFP FLAGS_REG))
   (clobber (match_scratch:HI 6 "=a"))]
  "reload_completed"
  [(const_int 0)]
{
  operands[7] = ix86_force_to_memory (GET_MODE (operands[2]), operands[2]);
  operands[7] = gen_rtx_FLOAT (GET_MODE (operands[1]), operands[7]);
  ix86_split_fp_branch (swap_condition (GET_CODE (operands[0])),
operands[3], operands[7],
operands[4], operands[5], operands[6], operands[2]);
  DONE;
})

;; Unconditional and other jump instructions

(define_insn "jump"
  [(set (pc)
```

```
(label_ref (match_operand 0 "" "")))]
  ""
  "jmp\t%l0"
  [(set_attr "type" "ibr")
   (set (attr "length")
   (if_then_else (and (ge (minus (match_dup 0) (pc))
  (const_int -126))
      (lt (minus (match_dup 0) (pc))
  (const_int 128)))
     (const_int 2)
     (const_int 5)))
   (set_attr "modrm" "0")])

(define_expand "indirect_jump"
  [(set (pc) (match_operand 0 "nonimmediate_operand" "rm"))]
  ""
  "")

(define_insn "*indirect_jump"
  [(set (pc) (match_operand:SI 0 "nonimmediate_operand" "rm"))]
  "!TARGET_64BIT"
  "jmp\t%A0"
  [(set_attr "type" "ibr")
   (set_attr "length_immediate" "0")])

(define_insn "*indirect_jump_rtx64"
  [(set (pc) (match_operand:DI 0 "nonimmediate_operand" "rm"))]
  "TARGET_64BIT"
  "jmp\t%A0"
  [(set_attr "type" "ibr")
   (set_attr "length_immediate" "0")])

(define_expand "tablejump"
  [(parallel [(set (pc) (match_operand 0 "nonimmediate_operand" "rm"))
      (use (label_ref (match_operand 1 "" "")))])]
  ""
{
  /* In PIC mode, the table entries are stored GOT (32-bit) or PC (64-bit)
     relative.  Convert the relative address to an absolute address.  */
  if (flag_pic)
    {
      rtx op0, op1;
      enum rtx_code code;

      if (TARGET_64BIT)
{
  code = PLUS;
  op0 = operands[0];
  op1 = gen_rtx_LABEL_REF (Pmode, operands[1]);
}
```

```
      else if (TARGET_MACHO || HAVE_AS_GOTOFF_IN_DATA)
{
  code = PLUS;
  op0 = operands[0];
  op1 = pic_offset_table_rtx;
}
      else
{
  code = MINUS;
  op0 = pic_offset_table_rtx;
  op1 = operands[0];
}

      operands[0] = expand_simple_binop (Pmode, code, op0, op1, NULL_RTX, 0,
 OPTAB_DIRECT);
    }
})

(define_insn "*tablejump_1"
  [(set (pc) (match_operand:SI 0 "nonimmediate_operand" "rm"))
   (use (label_ref (match_operand 1 "" "")))]
  "!TARGET_64BIT"
  "jmp\t%A0"
  [(set_attr "type" "ibr")
   (set_attr "length_immediate" "0")])

(define_insn "*tablejump_1_rtx64"
  [(set (pc) (match_operand:DI 0 "nonimmediate_operand" "rm"))
   (use (label_ref (match_operand 1 "" "")))]
  "TARGET_64BIT"
  "jmp\t%A0"
  [(set_attr "type" "ibr")
   (set_attr "length_immediate" "0")])

;; Convert setcc + movzbl to xor + setcc if operands don't overlap.

(define_peephole2
  [(set (reg FLAGS_REG) (match_operand 0 "" ""))
   (set (match_operand:QI 1 "register_operand" "")
(match_operator:QI 2 "ix86_comparison_operator"
  [(reg FLAGS_REG) (const_int 0)]))
   (set (match_operand 3 "q_regs_operand" "")
(zero_extend (match_dup 1)))]
  "(peep2_reg_dead_p (3, operands[1])
    || operands_match_p (operands[1], operands[3]))
   && ! reg_overlap_mentioned_p (operands[3], operands[0])"
  [(set (match_dup 4) (match_dup 0))
   (set (strict_low_part (match_dup 5))
(match_dup 2))]
{
```

```
  operands[4] = gen_rtx_REG (GET_MODE (operands[0]), FLAGS_REG);
  operands[5] = gen_lowpart (QImode, operands[3]);
  ix86_expand_clear (operands[3]);
})

;; Similar, but match zero_extendhisi2_and, which adds a clobber.

(define_peephole2
  [(set (reg FLAGS_REG) (match_operand 0 "" ""))
   (set (match_operand:QI 1 "register_operand" "")
(match_operator:QI 2 "ix86_comparison_operator"
  [(reg FLAGS_REG) (const_int 0)]))
   (parallel [(set (match_operand 3 "q_regs_operand" "")
   (zero_extend (match_dup 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "(peep2_reg_dead_p (3, operands[1])
    || operands_match_p (operands[1], operands[3]))
   && ! reg_overlap_mentioned_p (operands[3], operands[0])"
  [(set (match_dup 4) (match_dup 0))
   (set (strict_low_part (match_dup 5))
(match_dup 2))]
{
  operands[4] = gen_rtx_REG (GET_MODE (operands[0]), FLAGS_REG);
  operands[5] = gen_lowpart (QImode, operands[3]);
  ix86_expand_clear (operands[3]);
})

;; Call instructions.

;; The predicates normally associated with named expanders are not properly
;; checked for calls.  This is a bug in the generic code, but it isn't that
;; easy to fix.  Ignore it for now and be prepared to fix things up.

;; Call subroutine returning no value.

(define_expand "call_pop"
  [(parallel [(call (match_operand:QI 0 "" "")
    (match_operand:SI 1 "" ""))
      (set (reg:SI SP_REG)
   (plus:SI (reg:SI SP_REG)
    (match_operand:SI 3 "" "")))])]
  "!TARGET_64BIT"
{
  ix86_expand_call (NULL, operands[0], operands[1], operands[2], operands[3], 0);
  DONE;
})

(define_insn "*call_pop_0"
  [(call (mem:QI (match_operand:SI 0 "constant_call_address_operand" ""))
 (match_operand:SI 1 "" ""))
```

```
   (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG)
    (match_operand:SI 2 "immediate_operand" "")))]
  "!TARGET_64BIT"
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P0";
  else
    return "call\t%P0";
}
  [(set_attr "type" "call")])

(define_insn "*call_pop_1"
  [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
 (match_operand:SI 1 "" ""))
   (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG)
    (match_operand:SI 2 "immediate_operand" "i")))]
  "!TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], Pmode))
    {
      if (SIBLING_CALL_P (insn))
return "jmp\t%P0";
      else
return "call\t%P0";
    }
  if (SIBLING_CALL_P (insn))
    return "jmp\t%A0";
  else
    return "call\t%A0";
}
  [(set_attr "type" "call")])

(define_expand "call"
  [(call (match_operand:QI 0 "" "")
 (match_operand 1 "" ""))
   (use (match_operand 2 "" ""))]
  ""
{
  ix86_expand_call (NULL, operands[0], operands[1], operands[2], NULL, 0);
  DONE;
})

(define_expand "sibcall"
  [(call (match_operand:QI 0 "" "")
 (match_operand 1 "" ""))
   (use (match_operand 2 "" ""))]
  ""
{
  ix86_expand_call (NULL, operands[0], operands[1], operands[2], NULL, 1);
  DONE;
```

```
})

(define_insn "*call_0"
  [(call (mem:QI (match_operand 0 "constant_call_address_operand" ""))
 (match_operand 1 "" ""))]
  ""
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P0";
  else
    return "call\t%P0";
}
  [(set_attr "type" "call")])

(define_insn "*call_1"
  [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
 (match_operand 1 "" ""))]
  "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], Pmode))
    return "call\t%P0";
  return "call\t%A0";
}
  [(set_attr "type" "call")])

(define_insn "*sibcall_1"
  [(call (mem:QI (match_operand:SI 0 "sibcall_insn_operand" "s,c,d,a"))
 (match_operand 1 "" ""))]
  "SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], Pmode))
    return "jmp\t%P0";
  return "jmp\t%A0";
}
  [(set_attr "type" "call")])

(define_insn "*call_1_rex64"
  [(call (mem:QI (match_operand:DI 0 "call_insn_operand" "rsm"))
 (match_operand 1 "" ""))]
  "!SIBLING_CALL_P (insn) && TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], Pmode))
    return "call\t%P0";
  return "call\t%A0";
}
  [(set_attr "type" "call")])

(define_insn "*sibcall_1_rex64"
  [(call (mem:QI (match_operand:DI 0 "constant_call_address_operand" ""))
 (match_operand 1 "" ""))]
```

```
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t%P0"
  [(set_attr "type" "call")])

(define_insn "*sibcall_1_rex64_v"
  [(call (mem:QI (reg:DI 40))
 (match_operand 0 "" ""))]
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t*%%r11"
  [(set_attr "type" "call")])


;; Call subroutine, returning value in operand 0

(define_expand "call_value_pop"
  [(parallel [(set (match_operand 0 "" "")
   (call (match_operand:QI 1 "" "")
 (match_operand:SI 2 "" "")))
      (set (reg:SI SP_REG)
   (plus:SI (reg:SI SP_REG)
    (match_operand:SI 4 "" "")))])]
  "!TARGET_64BIT"
{
  ix86_expand_call (operands[0], operands[1], operands[2],
    operands[3], operands[4], 0);
  DONE;
})

(define_expand "call_value"
  [(set (match_operand 0 "" "")
(call (match_operand:QI 1 "" "")
      (match_operand:SI 2 "" "")))
   (use (match_operand:SI 3 "" ""))]
  ;; Operand 2 not used on the i386.
  ""
{
  ix86_expand_call (operands[0], operands[1], operands[2], operands[3], NULL, 0);
  DONE;
})

(define_expand "sibcall_value"
  [(set (match_operand 0 "" "")
(call (match_operand:QI 1 "" "")
      (match_operand:SI 2 "" "")))
   (use (match_operand:SI 3 "" ""))]
  ;; Operand 2 not used on the i386.
  ""
{
  ix86_expand_call (operands[0], operands[1], operands[2], operands[3], NULL, 1);
  DONE;
```

```
})

;; Call subroutine returning any type.

(define_expand "untyped_call"
  [(parallel [(call (match_operand 0 "" "")
    (const_int 0))
      (match_operand 1 "" "")
      (match_operand 2 "" "")])]
  ""
{
  int i;

  /* In order to give reg-stack an easier job in validating two
     coprocessor registers as containing a possible return value,
     simply pretend the untyped call returns a complex long double
     value.  */

  ix86_expand_call ((TARGET_FLOAT_RETURNS_IN_80387
     ? gen_rtx_REG (XCmode, FIRST_FLOAT_REG) : NULL),
    operands[0], const0_rtx, GEN_INT (SSE_REGPARM_MAX - 1),
    NULL, 0);

  for (i = 0; i < XVECLEN (operands[2], 0); i++)
    {
      rtx set = XVECEXP (operands[2], 0, i);
      emit_move_insn (SET_DEST (set), SET_SRC (set));
    }

  /* The optimizer does not know that the call sets the function value
     registers we stored in the result block.  We avoid problems by
     claiming that all hard registers are used and clobbered at this
     point.  */
  emit_insn (gen_blockage (const0_rtx));

  DONE;
})

;; Prologue and epilogue instructions

;; UNSPEC_VOLATILE is considered to use and clobber all hard registers and
;; all of memory.  This blocks insns from being moved across this point.

(define_insn "blockage"
  [(unspec_volatile [(match_operand 0 "" "")] UNSPECV_BLOCKAGE)]
  ""
  ""
  [(set_attr "length" "0")])

;; Insn emitted into the body of a function to return from a function.
```

```
;; This is only done if the function's epilogue is known to be simple.
;; See comments for ix86_can_use_return_insn_p in i386.c.

(define_expand "return"
  [(return)]
  "ix86_can_use_return_insn_p ()"
{
  if (current_function_pops_args)
    {
      rtx popc = GEN_INT (current_function_pops_args);
      emit_jump_insn (gen_return_pop_internal (popc));
      DONE;
    }
})

(define_insn "return_internal"
  [(return)]
  "reload_completed"
  "ret"
  [(set_attr "length" "1")
   (set_attr "length_immediate" "0")
   (set_attr "modrm" "0")])

;; Used by x86_machine_dependent_reorg to avoid penalty on single byte RET
;; instruction Athlon and K8 have.

(define_insn "return_internal_long"
  [(return)
   (unspec [(const_int 0)] UNSPEC_REP)]
  "reload_completed"
  "rep {;} ret"
  [(set_attr "length" "1")
   (set_attr "length_immediate" "0")
   (set_attr "prefix_rep" "1")
   (set_attr "modrm" "0")])

(define_insn "return_pop_internal"
  [(return)
   (use (match_operand:SI 0 "const_int_operand" ""))]
  "reload_completed"
  "ret\t%0"
  [(set_attr "length" "3")
   (set_attr "length_immediate" "2")
   (set_attr "modrm" "0")])

(define_insn "return_indirect_internal"
  [(return)
   (use (match_operand:SI 0 "register_operand" "r"))]
  "reload_completed"
  "jmp\t%A0"
```

```
  [(set_attr "type" "ibr")
   (set_attr "length_immediate" "0")])

(define_insn "nop"
  [(const_int 0)]
  ""
  "nop"
  [(set_attr "length" "1")
   (set_attr "length_immediate" "0")
   (set_attr "modrm" "0")])

;; Align to 16-byte boundary, max skip in op0.  Used to avoid
;; branch prediction penalty for the third jump in a 16-byte
;; block on K8.

(define_insn "align"
  [(unspec_volatile [(match_operand 0 "" "")] UNSPECV_ALIGN)]
  ""
{
#ifdef ASM_OUTPUT_MAX_SKIP_ALIGN
  ASM_OUTPUT_MAX_SKIP_ALIGN (asm_out_file, 4, (int)INTVAL (operands[0]));
#else
  /* It is tempting to use ASM_OUTPUT_ALIGN here, but we don't want to do that.
     The align insn is used to avoid 3 jump instructions in the row to improve
     branch prediction and the benefits hardly outweight the cost of extra 8
     nops on the average inserted by full alignment pseudo operation.  */
#endif
  return "";
}
  [(set_attr "length" "16")])

(define_expand "prologue"
  [(const_int 1)]
  ""
  "ix86_expand_prologue (); DONE;")

(define_insn "set_got"
  [(set (match_operand:SI 0 "register_operand" "=r")
(unspec:SI [(const_int 0)] UNSPEC_SET_GOT))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  { return output_set_got (operands[0]); }
  [(set_attr "type" "multi")
   (set_attr "length" "12")])

(define_insn "set_got_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(unspec:DI [(const_int 0)] UNSPEC_SET_GOT))]
  "TARGET_64BIT"
  "lea{q}\t_GLOBAL_OFFSET_TABLE_(%%rip), %0"
```

```
  [(set_attr "type" "lea")
   (set_attr "length" "6")])

(define_expand "epilogue"
  [(const_int 1)]
  ""
  "ix86_expand_epilogue (1); DONE;")

(define_expand "sibcall_epilogue"
  [(const_int 1)]
  ""
  "ix86_expand_epilogue (0); DONE;")

(define_expand "eh_return"
  [(use (match_operand 0 "register_operand" ""))]
  ""
{
  rtx tmp, sa = EH_RETURN_STACKADJ_RTX, ra = operands[0];

  /* Tricky bit: we write the address of the handler to which we will
     be returning into someone else's stack frame, one word below the
     stack address we wish to restore.  */
  tmp = gen_rtx_PLUS (Pmode, arg_pointer_rtx, sa);
  tmp = plus_constant (tmp, -UNITS_PER_WORD);
  tmp = gen_rtx_MEM (Pmode, tmp);
  emit_move_insn (tmp, ra);

  if (Pmode == SImode)
    emit_jump_insn (gen_eh_return_si (sa));
  else
    emit_jump_insn (gen_eh_return_di (sa));
  emit_barrier ();
  DONE;
})

(define_insn_and_split "eh_return_si"
  [(set (pc)
        (unspec [(match_operand:SI 0 "register_operand" "c")]
         UNSPEC_EH_RETURN))]
  "!TARGET_64BIT"
  "#"
  "reload_completed"
  [(const_int 1)]
  "ix86_expand_epilogue (2); DONE;")

(define_insn_and_split "eh_return_di"
  [(set (pc)
        (unspec [(match_operand:DI 0 "register_operand" "c")]
         UNSPEC_EH_RETURN))]
  "TARGET_64BIT"
```

```
  "#"
  "reload_completed"
  [(const_int 1)]
  "ix86_expand_epilogue (2); DONE;")

(define_insn "leave"
  [(set (reg:SI SP_REG) (plus:SI (reg:SI BP_REG) (const_int 4)))
   (set (reg:SI BP_REG) (mem:SI (reg:SI BP_REG)))
   (clobber (mem:BLK (scratch)))]
  "!TARGET_64BIT"
  "leave"
  [(set_attr "type" "leave")])

(define_insn "leave_rex64"
  [(set (reg:DI SP_REG) (plus:DI (reg:DI BP_REG) (const_int 8)))
   (set (reg:DI BP_REG) (mem:DI (reg:DI BP_REG)))
   (clobber (mem:BLK (scratch)))]
  "TARGET_64BIT"
  "leave"
  [(set_attr "type" "leave")])

(define_expand "ffssi2"
  [(parallel
     [(set (match_operand:SI 0 "register_operand" "")
   (ffs:SI (match_operand:SI 1 "nonimmediate_operand" "")))
      (clobber (match_scratch:SI 2 ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn_and_split "*ffs_cmove"
  [(set (match_operand:SI 0 "register_operand" "=r")
(ffs:SI (match_operand:SI 1 "nonimmediate_operand" "rm")))
   (clobber (match_scratch:SI 2 "=&r"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_CMOVE"
  "#"
  "&& reload_completed"
  [(set (match_dup 2) (const_int -1))
   (parallel [(set (reg:CCZ FLAGS_REG) (compare:CCZ (match_dup 1) (const_int 0)))
      (set (match_dup 0) (ctz:SI (match_dup 1)))])
   (set (match_dup 0) (if_then_else:SI
(eq (reg:CCZ FLAGS_REG) (const_int 0))
(match_dup 2)
(match_dup 0)))
   (parallel [(set (match_dup 0) (plus:SI (match_dup 0) (const_int 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_insn_and_split "*ffs_no_cmove"
```

```
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r")
(ffs:SI (match_operand:SI 1 "nonimmediate_operand" "rm")))
   (clobber (match_scratch:SI 2 "=&q"))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "#"
  "reload_completed"
  [(parallel [(set (reg:CCZ FLAGS_REG) (compare:CCZ (match_dup 1) (const_int 0)))
      (set (match_dup 0) (ctz:SI (match_dup 1)))])
   (set (strict_low_part (match_dup 3))
(eq:QI (reg:CCZ FLAGS_REG) (const_int 0)))
   (parallel [(set (match_dup 2) (neg:SI (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (match_dup 0) (ior:SI (match_dup 0) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (match_dup 0) (plus:SI (match_dup 0) (const_int 1)))
      (clobber (reg:CC FLAGS_REG))])]
{
  operands[3] = gen_lowpart (QImode, operands[2]);
  ix86_expand_clear (operands[2]);
})

(define_insn "*ffssi_1"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (match_operand:SI 1 "nonimmediate_operand" "rm")
     (const_int 0)))
   (set (match_operand:SI 0 "register_operand" "=r")
(ctz:SI (match_dup 1)))]
  ""
  "bsf{l}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

(define_expand "ffsdi2"
  [(parallel
     [(set (match_operand:DI 0 "register_operand" "")
   (ffs:DI (match_operand:DI 1 "nonimmediate_operand" "")))
      (clobber (match_scratch:DI 2 ""))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT && TARGET_CMOVE"
  "")

(define_insn_and_split "*ffs_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(ffs:DI (match_operand:DI 1 "nonimmediate_operand" "rm")))
   (clobber (match_scratch:DI 2 "=&r"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && TARGET_CMOVE"
  "#"
  "&& reload_completed"
  [(set (match_dup 2) (const_int -1))
```

```
   (parallel [(set (reg:CCZ FLAGS_REG)
   (compare:CCZ (match_dup 1) (const_int 0)))
      (set (match_dup 0) (ctz:DI (match_dup 1)))])
   (set (match_dup 0) (if_then_else:DI
(eq (reg:CCZ FLAGS_REG) (const_int 0))
(match_dup 2)
(match_dup 0)))
   (parallel [(set (match_dup 0) (plus:DI (match_dup 0) (const_int 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_insn "*ffsdi_1"
  [(set (reg:CCZ FLAGS_REG)
(compare:CCZ (match_operand:DI 1 "nonimmediate_operand" "rm")
     (const_int 0)))
   (set (match_operand:DI 0 "register_operand" "=r")
(ctz:DI (match_dup 1)))]
  "TARGET_64BIT"
  "bsf{q}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

(define_insn "ctzsi2"
  [(set (match_operand:SI 0 "register_operand" "=r")
(ctz:SI (match_operand:SI 1 "nonimmediate_operand" "rm")))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "bsf{l}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

(define_insn "ctzdi2"
  [(set (match_operand:DI 0 "register_operand" "=r")
(ctz:DI (match_operand:DI 1 "nonimmediate_operand" "rm")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "bsf{q}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

(define_expand "clzsi2"
  [(parallel
     [(set (match_operand:SI 0 "register_operand" "")
   (minus:SI (const_int 31)
     (clz:SI (match_operand:SI 1 "nonimmediate_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])
   (parallel
     [(set (match_dup 0) (xor:SI (match_dup 0) (const_int 31)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*bsr"
```

```
  [(set (match_operand:SI 0 "register_operand" "=r")
(minus:SI (const_int 31)
  (clz:SI (match_operand:SI 1 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "bsr{l}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

(define_expand "clzdi2"
  [(parallel
     [(set (match_operand:DI 0 "register_operand" "")
   (minus:DI (const_int 63)
     (clz:DI (match_operand:DI 1 "nonimmediate_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])
   (parallel
     [(set (match_dup 0) (xor:DI (match_dup 0) (const_int 63)))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT"
  "")

(define_insn "*bsr_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(minus:DI (const_int 63)
  (clz:DI (match_operand:DI 1 "nonimmediate_operand" "rm"))))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "bsr{q}\t{%1, %0|%0, %1}"
  [(set_attr "prefix_0f" "1")])

;; Thread-local storage patterns for ELF.
;;
;; Note that these code sequences must appear exactly as shown
;; in order to allow linker relaxation.

(define_insn "*tls_global_dynamic_32_gnu"
  [(set (match_operand:SI 0 "register_operand" "=a")
(unspec:SI [(match_operand:SI 1 "register_operand" "b")
    (match_operand:SI 2 "tls_symbolic_operand" "")
    (match_operand:SI 3 "call_insn_operand" "")]
    UNSPEC_TLS_GD))
   (clobber (match_scratch:SI 4 "=d"))
   (clobber (match_scratch:SI 5 "=c"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && TARGET_GNU_TLS"
  "lea{l}\t{%a2@TLSGD(,%1,1), %0|%0, %a2@TLSGD[%1*1]}\;call\t%P3"
  [(set_attr "type" "multi")
   (set_attr "length" "12")])

(define_insn "*tls_global_dynamic_32_sun"
  [(set (match_operand:SI 0 "register_operand" "=a")
```

```
(unspec:SI [(match_operand:SI 1 "register_operand" "b")
    (match_operand:SI 2 "tls_symbolic_operand" "")
    (match_operand:SI 3 "call_insn_operand" "")]
    UNSPEC_TLS_GD))
   (clobber (match_scratch:SI 4 "=d"))
   (clobber (match_scratch:SI 5 "=c"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && TARGET_SUN_TLS"
  "lea{l}\t{%a2@DTLNDX(%1), %4|%4, %a2@DTLNDX[%1]}
push{l}\t%4\;call\t%a2@TLSPLT\;pop{l}\t%4\;nop"
  [(set_attr "type" "multi")
   (set_attr "length" "14")])

(define_expand "tls_global_dynamic_32"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (unspec:SI
    [(match_dup 2)
     (match_operand:SI 1 "tls_symbolic_operand" "")
     (match_dup 3)]
    UNSPEC_TLS_GD))
      (clobber (match_scratch:SI 4 ""))
      (clobber (match_scratch:SI 5 ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
{
  if (flag_pic)
    operands[2] = pic_offset_table_rtx;
  else
    {
      operands[2] = gen_reg_rtx (Pmode);
      emit_insn (gen_set_got (operands[2]));
    }
  operands[3] = ix86_tls_get_addr ();
})

(define_insn "*tls_global_dynamic_64"
  [(set (match_operand:DI 0 "register_operand" "=a")
(call (mem:QI (match_operand:DI 2 "call_insn_operand" ""))
      (match_operand:DI 3 "" "")))
   (unspec:DI [(match_operand:DI 1 "tls_symbolic_operand" "")]
      UNSPEC_TLS_GD)]
  "TARGET_64BIT"
  ".byte\t0x66\;lea{q}\t{%a1@TLSGD(%%rip), %%rdi|%%rdi, %a1@TLSGD[%%rip]}\;.word\t0x6666\;rex64\;call\
  [(set_attr "type" "multi")
   (set_attr "length" "16")])

(define_expand "tls_global_dynamic_64"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (call (mem:QI (match_dup 2)) (const_int 0)))
      (unspec:DI [(match_operand:DI 1 "tls_symbolic_operand" "")]
```

```
 UNSPEC_TLS_GD)])]
  ""
{
  operands[2] = ix86_tls_get_addr ();
})

(define_insn "*tls_local_dynamic_base_32_gnu"
  [(set (match_operand:SI 0 "register_operand" "=a")
(unspec:SI [(match_operand:SI 1 "register_operand" "b")
                    (match_operand:SI 2 "call_insn_operand" "")]
   UNSPEC_TLS_LD_BASE))
   (clobber (match_scratch:SI 3 "=d"))
   (clobber (match_scratch:SI 4 "=c"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && TARGET_GNU_TLS"
  "lea{l}\t{%&@TLSLDM(%1), %0|%0, %&@TLSLDM[%1]}\;call\t%P2"
  [(set_attr "type" "multi")
   (set_attr "length" "11")])

(define_insn "*tls_local_dynamic_base_32_sun"
  [(set (match_operand:SI 0 "register_operand" "=a")
(unspec:SI [(match_operand:SI 1 "register_operand" "b")
                    (match_operand:SI 2 "call_insn_operand" "")]
   UNSPEC_TLS_LD_BASE))
   (clobber (match_scratch:SI 3 "=d"))
   (clobber (match_scratch:SI 4 "=c"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && TARGET_SUN_TLS"
  "lea{l}\t{%&@TMDNX(%1), %3|%3, %&@TMDNX[%1]}
push{l}\t%3\;call\t%&@TLSPLT\;pop{l}\t%3"
  [(set_attr "type" "multi")
   (set_attr "length" "13")])

(define_expand "tls_local_dynamic_base_32"
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (unspec:SI [(match_dup 1) (match_dup 2)]
      UNSPEC_TLS_LD_BASE))
      (clobber (match_scratch:SI 3 ""))
      (clobber (match_scratch:SI 4 ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
{
  if (flag_pic)
    operands[1] = pic_offset_table_rtx;
  else
    {
      operands[1] = gen_reg_rtx (Pmode);
      emit_insn (gen_set_got (operands[1]));
    }
  operands[2] = ix86_tls_get_addr ();
```

```
})

(define_insn "*tls_local_dynamic_base_64"
  [(set (match_operand:DI 0 "register_operand" "=a")
(call (mem:QI (match_operand:DI 1 "call_insn_operand" ""))
      (match_operand:DI 2 "" "")))
   (unspec:DI [(const_int 0)] UNSPEC_TLS_LD_BASE)]
  "TARGET_64BIT"
  "lea{q}\t{%&@TLSLD(%%rip), %%rdi|%%rdi, %&@TLSLD[%%rip]}\;call\t%P1"
  [(set_attr "type" "multi")
   (set_attr "length" "12")])

(define_expand "tls_local_dynamic_base_64"
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (call (mem:QI (match_dup 1)) (const_int 0)))
      (unspec:DI [(const_int 0)] UNSPEC_TLS_LD_BASE)])]
  ""
{
  operands[1] = ix86_tls_get_addr ();
})

;; Local dynamic of a single variable is a lose.  Show combine how
;; to convert that back to global dynamic.

(define_insn_and_split "*tls_local_dynamic_32_once"
  [(set (match_operand:SI 0 "register_operand" "=a")
(plus:SI (unspec:SI [(match_operand:SI 1 "register_operand" "b")
     (match_operand:SI 2 "call_insn_operand" "")]
    UNSPEC_TLS_LD_BASE)
 (const:SI (unspec:SI
    [(match_operand:SI 3 "tls_symbolic_operand" "")]
    UNSPEC_DTPOFF))))
   (clobber (match_scratch:SI 4 "=d"))
   (clobber (match_scratch:SI 5 "=c"))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "#"
  ""
  [(parallel [(set (match_dup 0)
   (unspec:SI [(match_dup 1) (match_dup 3) (match_dup 2)]
      UNSPEC_TLS_GD))
      (clobber (match_dup 4))
      (clobber (match_dup 5))
      (clobber (reg:CC FLAGS_REG))])]
  "")

;; Load and add the thread base pointer from %gs:0.

(define_insn "*load_tp_si"
  [(set (match_operand:SI 0 "register_operand" "=r")
```

```
(unspec:SI [(const_int 0)] UNSPEC_TP))]
  "!TARGET_64BIT"
  "mov{l}\t{%%gs:0, %0|%0, DWORD PTR %%gs:0}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0")
   (set_attr "length" "7")
   (set_attr "memory" "load")
   (set_attr "imm_disp" "false")])

(define_insn "*add_tp_si"
  [(set (match_operand:SI 0 "register_operand" "=r")
(plus:SI (unspec:SI [(const_int 0)] UNSPEC_TP)
 (match_operand:SI 1 "register_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "add{l}\t{%%gs:0, %0|%0, DWORD PTR %%gs:0}"
  [(set_attr "type" "alu")
   (set_attr "modrm" "0")
   (set_attr "length" "7")
   (set_attr "memory" "load")
   (set_attr "imm_disp" "false")])

(define_insn "*load_tp_di"
  [(set (match_operand:DI 0 "register_operand" "=r")
(unspec:DI [(const_int 0)] UNSPEC_TP))]
  "TARGET_64BIT"
  "mov{q}\t{%%fs:0, %0|%0, QWORD PTR %%fs:0}"
  [(set_attr "type" "imov")
   (set_attr "modrm" "0")
   (set_attr "length" "7")
   (set_attr "memory" "load")
   (set_attr "imm_disp" "false")])

(define_insn "*add_tp_di"
  [(set (match_operand:DI 0 "register_operand" "=r")
(plus:DI (unspec:DI [(const_int 0)] UNSPEC_TP)
 (match_operand:DI 1 "register_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "add{q}\t{%%fs:0, %0|%0, QWORD PTR %%fs:0}"
  [(set_attr "type" "alu")
   (set_attr "modrm" "0")
   (set_attr "length" "7")
   (set_attr "memory" "load")
   (set_attr "imm_disp" "false")])

;; These patterns match the binary 387 instructions for addM3, subM3,
;; mulM3 and divM3.  There are three patterns for each of DFmode and
;; SFmode.  The first is the normal insn, the second the same insn but
;; with one operand a conversion, and the third the same insn but with
```

```
;; the other operand a conversion.  The conversion may be SFmode or
;; SImode if the target mode DFmode, but only SImode if the target mode
;; is SFmode.

;; Gcc is slightly more smart about handling normal two address instructions
;; so use special patterns for add and mull.

(define_insn "*fop_sf_comm_mixed"
  [(set (match_operand:SF 0 "register_operand" "=f#x,x#f")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "nonimmediate_operand" "%0,0")
 (match_operand:SF 2 "nonimmediate_operand" "fm#x,xm#f")]))]
  "TARGET_MIX_SSE_I387
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
(if_then_else (eq_attr "alternative" "1")
   (if_then_else (match_operand:SF 3 "mult_operator" "")
      (const_string "ssemul")
      (const_string "sseadd"))
   (if_then_else (match_operand:SF 3 "mult_operator" "")
      (const_string "fmul")
      (const_string "fop"))))
   (set_attr "mode" "SF")])

(define_insn "*fop_sf_comm_sse"
  [(set (match_operand:SF 0 "register_operand" "=x")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "nonimmediate_operand" "%0")
 (match_operand:SF 2 "nonimmediate_operand" "xm")]))]
  "TARGET_SSE_MATH
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (if_then_else (match_operand:SF 3 "mult_operator" "")
   (const_string "ssemul")
   (const_string "sseadd")))
   (set_attr "mode" "SF")])

(define_insn "*fop_sf_comm_i387"
  [(set (match_operand:SF 0 "register_operand" "=f")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "nonimmediate_operand" "%0")
 (match_operand:SF 2 "nonimmediate_operand" "fm")]))]
  "TARGET_80387
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
```

```
  [(set (attr "type")
(if_then_else (match_operand:SF 3 "mult_operator" "")
   (const_string "fmul")
   (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_sf_1_mixed"
  [(set (match_operand:SF 0 "register_operand" "=f,f,x")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "nonimmediate_operand" "0,fm,0")
 (match_operand:SF 2 "nonimmediate_operand" "fm,0,xm#f")]))]
  "TARGET_MIX_SSE_I387
   && !COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(and (eq_attr "alternative" "2")
            (match_operand:SF 3 "mult_operator" ""))
                 (const_string "ssemul")
       (and (eq_attr "alternative" "2")
            (match_operand:SF 3 "div_operator" ""))
                 (const_string "ssediv")
       (eq_attr "alternative" "2")
                 (const_string "sseadd")
       (match_operand:SF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:SF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_sf_1_sse"
  [(set (match_operand:SF 0 "register_operand" "=x")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "register_operand" "0")
 (match_operand:SF 2 "nonimmediate_operand" "xm")]))]
  "TARGET_SSE_MATH
   && !COMMUTATIVE_ARITH_P (operands[3])"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:SF 3 "mult_operator" "")
                 (const_string "ssemul")
       (match_operand:SF 3 "div_operator" "")
                 (const_string "ssediv")
              ]
              (const_string "sseadd")))
   (set_attr "mode" "SF")])

;; This pattern is not fully shadowed by the pattern above.
```

```
(define_insn "*fop_sf_1_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(match_operator:SF 3 "binary_fp_operator"
[(match_operand:SF 1 "nonimmediate_operand" "0,fm")
 (match_operand:SF 2 "nonimmediate_operand" "fm,0")]))]
  "TARGET_80387 && !TARGET_SSE_MATH
   && !COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:SF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:SF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

;; ??? Add SSE splitters for these!
(define_insn "*fop_sf_2<mode>_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(match_operator:SF 3 "binary_fp_operator"
  [(float:SF (match_operand:X87MODEI12 1 "nonimmediate_operand" "m,?r"))
   (match_operand:SF 2 "register_operand" "0,0")]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP && !TARGET_SSE_MATH"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:SF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:SF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])

(define_insn "*fop_sf_3<mode>_i387"
  [(set (match_operand:SF 0 "register_operand" "=f,f")
(match_operator:SF 3 "binary_fp_operator"
  [(match_operand:SF 1 "register_operand" "0,0")
   (float:SF (match_operand:X87MODEI12 2 "nonimmediate_operand" "m,?r"))]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP && !TARGET_SSE_MATH"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:SF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:SF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
```

```
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])

(define_insn "*fop_df_comm_mixed"
  [(set (match_operand:DF 0 "register_operand" "=f#Y,Y#f")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "nonimmediate_operand" "%0,0")
 (match_operand:DF 2 "nonimmediate_operand" "fm#Y,Ym#f")]))]
  "TARGET_SSE2 && TARGET_MIX_SSE_I387
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
(if_then_else (eq_attr "alternative" "1")
   (if_then_else (match_operand:SF 3 "mult_operator" "")
      (const_string "ssemul")
      (const_string "sseadd"))
   (if_then_else (match_operand:SF 3 "mult_operator" "")
      (const_string "fmul")
      (const_string "fop"))))
   (set_attr "mode" "DF")])

(define_insn "*fop_df_comm_sse"
  [(set (match_operand:DF 0 "register_operand" "=Y")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "nonimmediate_operand" "%0")
 (match_operand:DF 2 "nonimmediate_operand" "Ym")]))]
  "TARGET_SSE2 && TARGET_SSE_MATH
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (if_then_else (match_operand:SF 3 "mult_operator" "")
   (const_string "ssemul")
   (const_string "sseadd")))
   (set_attr "mode" "DF")])

(define_insn "*fop_df_comm_i387"
  [(set (match_operand:DF 0 "register_operand" "=f")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "nonimmediate_operand" "%0")
 (match_operand:DF 2 "nonimmediate_operand" "fm")]))]
  "TARGET_80387
   && COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
(if_then_else (match_operand:SF 3 "mult_operator" "")
   (const_string "fmul")
   (const_string "fop")))
```

```
   (set_attr "mode" "DF")])

(define_insn "*fop_df_1_mixed"
  [(set (match_operand:DF 0 "register_operand" "=f#Y,f#Y,Y#f")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "nonimmediate_operand" "0,fm,0")
 (match_operand:DF 2 "nonimmediate_operand" "fm,0,Ym#f")]))]
  "TARGET_SSE2 && TARGET_SSE_MATH && TARGET_MIX_SSE_I387
   && !COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(and (eq_attr "alternative" "2")
            (match_operand:SF 3 "mult_operator" ""))
                 (const_string "ssemul")
       (and (eq_attr "alternative" "2")
            (match_operand:SF 3 "div_operator" ""))
                 (const_string "ssediv")
       (eq_attr "alternative" "2")
                 (const_string "sseadd")
       (match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "DF")])

(define_insn "*fop_df_1_sse"
  [(set (match_operand:DF 0 "register_operand" "=Y")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "register_operand" "0")
 (match_operand:DF 2 "nonimmediate_operand" "Ym")]))]
  "TARGET_SSE2 && TARGET_SSE_MATH
   && !COMMUTATIVE_ARITH_P (operands[3])"
  "* return output_387_binary_op (insn, operands);"
  [(set_attr "mode" "DF")
   (set (attr "type")
        (cond [(match_operand:SF 3 "mult_operator" "")
                 (const_string "ssemul")
       (match_operand:SF 3 "div_operator" "")
                 (const_string "ssediv")
              ]
              (const_string "sseadd")))])

;; This pattern is not fully shadowed by the pattern above.
(define_insn "*fop_df_1_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
[(match_operand:DF 1 "nonimmediate_operand" "0,fm")
```

```
 (match_operand:DF 2 "nonimmediate_operand" "fm,0")]))]
  "TARGET_80387 && !(TARGET_SSE2 && TARGET_SSE_MATH)
   && !COMMUTATIVE_ARITH_P (operands[3])
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "DF")])

;; ??? Add SSE splitters for these!
(define_insn "*fop_df_2<mode>_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
   [(float:DF (match_operand:X87MODEI12 1 "nonimmediate_operand" "m,?r"))
    (match_operand:DF 2 "register_operand" "0,0")]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP
   && !(TARGET_SSE2 && TARGET_SSE_MATH)"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])

(define_insn "*fop_df_3<mode>_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
   [(match_operand:DF 1 "register_operand" "0,0")
    (float:DF (match_operand:X87MODEI12 2 "nonimmediate_operand" "m,?r"))]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP
   && !(TARGET_SSE2 && TARGET_SSE_MATH)"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])
```

```
(define_insn "*fop_df_4_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
   [(float_extend:DF (match_operand:SF 1 "nonimmediate_operand" "fm,0"))
    (match_operand:DF 2 "register_operand" "0,f")]))]
  "TARGET_80387 && !(TARGET_SSE2 && TARGET_SSE_MATH)
   && (GET_CODE (operands[1]) != MEM || GET_CODE (operands[2]) != MEM)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_df_5_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
  [(match_operand:DF 1 "register_operand" "0,f")
   (float_extend:DF
    (match_operand:SF 2 "nonimmediate_operand" "fm,0"))]))]
  "TARGET_80387 && !(TARGET_SSE2 && TARGET_SSE_MATH)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_df_6_i387"
  [(set (match_operand:DF 0 "register_operand" "=f,f")
(match_operator:DF 3 "binary_fp_operator"
  [(float_extend:DF
    (match_operand:SF 1 "register_operand" "0,f"))
   (float_extend:DF
    (match_operand:SF 2 "nonimmediate_operand" "fm,0"))]))]
  "TARGET_80387 && !(TARGET_SSE2 && TARGET_SSE_MATH)"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:DF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:DF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
```

```
            (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_xf_comm_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(match_operator:XF 3 "binary_fp_operator"
[(match_operand:XF 1 "register_operand" "%0")
 (match_operand:XF 2 "register_operand" "f")]))]
  "TARGET_80387
   && COMMUTATIVE_ARITH_P (operands[3])"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (if_then_else (match_operand:XF 3 "mult_operator" "")
           (const_string "fmul")
           (const_string "fop")))
   (set_attr "mode" "XF")])

(define_insn "*fop_xf_1_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
[(match_operand:XF 1 "register_operand" "0,f")
 (match_operand:XF 2 "register_operand" "f,0")]))]
  "TARGET_80387
   && !COMMUTATIVE_ARITH_P (operands[3])"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "XF")])

(define_insn "*fop_xf_2<mode>_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
   [(float:XF (match_operand:X87MODEI12 1 "nonimmediate_operand" "m,?r"))
    (match_operand:XF 2 "register_operand" "0,0")]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])
```

```
(define_insn "*fop_xf_3<mode>_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
  [(match_operand:XF 1 "register_operand" "0,0")
   (float:XF (match_operand:X87MODEI12 2 "nonimmediate_operand" "m,?r"))]))]
  "TARGET_80387 && TARGET_USE_<MODE>MODE_FIOP"
  "* return which_alternative ? \"#\" : output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "fp_int_src" "true")
   (set_attr "mode" "<MODE>")])

(define_insn "*fop_xf_4_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
   [(float_extend:XF (match_operand 1 "nonimmediate_operand" "fm,0"))
    (match_operand:XF 2 "register_operand" "0,f")]))]
  "TARGET_80387"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

(define_insn "*fop_xf_5_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
  [(match_operand:XF 1 "register_operand" "0,f")
   (float_extend:XF
    (match_operand 2 "nonimmediate_operand" "fm,0"))]))]
  "TARGET_80387"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])
```

```
(define_insn "*fop_xf_6_i387"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(match_operator:XF 3 "binary_fp_operator"
  [(float_extend:XF
    (match_operand 1 "register_operand" "0,f"))
   (float_extend:XF
    (match_operand 2 "nonimmediate_operand" "fm,0"))]))]
  "TARGET_80387"
  "* return output_387_binary_op (insn, operands);"
  [(set (attr "type")
        (cond [(match_operand:XF 3 "mult_operator" "")
                 (const_string "fmul")
               (match_operand:XF 3 "div_operator" "")
                 (const_string "fdiv")
              ]
              (const_string "fop")))
   (set_attr "mode" "SF")])

(define_split
  [(set (match_operand 0 "register_operand" "")
(match_operator 3 "binary_fp_operator"
   [(float (match_operand:X87MODEI12 1 "register_operand" ""))
    (match_operand 2 "register_operand" "")]))]
  "TARGET_80387 && reload_completed
   && FLOAT_MODE_P (GET_MODE (operands[0]))"
  [(const_int 0)]
{
  operands[4] = ix86_force_to_memory (GET_MODE (operands[1]), operands[1]);
  operands[4] = gen_rtx_FLOAT (GET_MODE (operands[0]), operands[4]);
  emit_insn (gen_rtx_SET (VOIDmode, operands[0],
  gen_rtx_fmt_ee (GET_CODE (operands[3]),
  GET_MODE (operands[3]),
  operands[4],
  operands[2])));
  ix86_free_from_memory (GET_MODE (operands[1]));
  DONE;
})

(define_split
  [(set (match_operand 0 "register_operand" "")
(match_operator 3 "binary_fp_operator"
   [(match_operand 1 "register_operand" "")
    (float (match_operand:X87MODEI12 2 "register_operand" ""))]))]
  "TARGET_80387 && reload_completed
   && FLOAT_MODE_P (GET_MODE (operands[0]))"
  [(const_int 0)]
{
  operands[4] = ix86_force_to_memory (GET_MODE (operands[2]), operands[2]);
  operands[4] = gen_rtx_FLOAT (GET_MODE (operands[0]), operands[4]);
```

```
  emit_insn (gen_rtx_SET (VOIDmode, operands[0],
  gen_rtx_fmt_ee (GET_CODE (operands[3]),
  GET_MODE (operands[3]),
  operands[1],
  operands[4])));
  ix86_free_from_memory (GET_MODE (operands[2]));
  DONE;
})

;; FPU special functions.

(define_expand "sqrtsf2"
  [(set (match_operand:SF 0 "register_operand" "")
(sqrt:SF (match_operand:SF 1 "nonimmediate_operand" "")))]
  "TARGET_USE_FANCY_MATH_387 || TARGET_SSE_MATH"
{
  if (!TARGET_SSE_MATH)
    operands[1] = force_reg (SFmode, operands[1]);
})

(define_insn "*sqrtsf2_mixed"
  [(set (match_operand:SF 0 "register_operand" "=f#x,x#f")
(sqrt:SF (match_operand:SF 1 "nonimmediate_operand" "0#x,xm#f")))]
  "TARGET_USE_FANCY_MATH_387 && TARGET_MIX_SSE_I387"
  "@
   fsqrt
   sqrtss\t{%1, %0|%0, %1}"
  [(set_attr "type" "fpspc,sse")
   (set_attr "mode" "SF,SF")
   (set_attr "athlon_decode" "direct,*")])

(define_insn "*sqrtsf2_sse"
  [(set (match_operand:SF 0 "register_operand" "=x")
(sqrt:SF (match_operand:SF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE_MATH"
  "sqrtss\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")
   (set_attr "athlon_decode" "*")])

(define_insn "*sqrtsf2_i387"
  [(set (match_operand:SF 0 "register_operand" "=f")
(sqrt:SF (match_operand:SF 1 "register_operand" "0")))]
  "TARGET_USE_FANCY_MATH_387"
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")
   (set_attr "athlon_decode" "direct")])

(define_expand "sqrtdf2"
```

```
  [(set (match_operand:DF 0 "register_operand" "")
(sqrt:DF (match_operand:DF 1 "nonimmediate_operand" "")))]
  "TARGET_USE_FANCY_MATH_387 || (TARGET_SSE2 && TARGET_SSE_MATH)"
{
  if (!(TARGET_SSE2 && TARGET_SSE_MATH))
    operands[1] = force_reg (DFmode, operands[1]);
})

(define_insn "*sqrtdf2_mixed"
  [(set (match_operand:DF 0 "register_operand" "=f#Y,Y#f")
(sqrt:DF (match_operand:DF 1 "nonimmediate_operand" "0#Y,Ym#f")))]
  "TARGET_USE_FANCY_MATH_387 && TARGET_SSE2 && TARGET_MIX_SSE_I387"
  "@
   fsqrt
   sqrtsd\t{%1, %0|%0, %1}"
  [(set_attr "type" "fpspc,sse")
   (set_attr "mode" "DF,DF")
   (set_attr "athlon_decode" "direct,*")])

(define_insn "*sqrtdf2_sse"
  [(set (match_operand:DF 0 "register_operand" "=Y")
(sqrt:DF (match_operand:DF 1 "nonimmediate_operand" "Ym")))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "sqrtsd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "*")])

(define_insn "*sqrtdf2_i387"
  [(set (match_operand:DF 0 "register_operand" "=f")
(sqrt:DF (match_operand:DF 1 "register_operand" "0")))]
  "TARGET_USE_FANCY_MATH_387"
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "direct")])

(define_insn "*sqrtextendsfdf2_i387"
  [(set (match_operand:DF 0 "register_operand" "=f")
(sqrt:DF (float_extend:DF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)"
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "direct")])

(define_insn "sqrtxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
```

```
(sqrt:XF (match_operand:XF 1 "register_operand" "0")))]
  "TARGET_USE_FANCY_MATH_387
   && (TARGET_IEEE_FP || flag_unsafe_math_optimizations) "
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")
   (set_attr "athlon_decode" "direct")])

(define_insn "*sqrtextendsfxf2_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(sqrt:XF (float_extend:XF
  (match_operand:SF 1 "register_operand" "0"))))]
  "TARGET_USE_FANCY_MATH_387"
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")
   (set_attr "athlon_decode" "direct")])

(define_insn "*sqrtextenddfxf2_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(sqrt:XF (float_extend:XF
  (match_operand:DF 1 "register_operand" "0"))))]
  "TARGET_USE_FANCY_MATH_387"
  "fsqrt"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")
   (set_attr "athlon_decode" "direct")])

(define_insn "fpremxf4"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 2 "register_operand" "0")
    (match_operand:XF 3 "register_operand" "1")]
   UNSPEC_FPREM_F))
   (set (match_operand:XF 1 "register_operand" "=u")
(unspec:XF [(match_dup 2) (match_dup 3)]
   UNSPEC_FPREM_U))
   (set (reg:CCFP FPSR_REG)
(unspec:CCFP [(const_int 0)] UNSPEC_NOP))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fprem"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "fmodsf3"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))
   (use (match_operand:SF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
```

```
   && flag_unsafe_math_optimizations"
{
  rtx label = gen_label_rtx ();

  rtx op1 = gen_reg_rtx (XFmode);
  rtx op2 = gen_reg_rtx (XFmode);

  emit_insn(gen_extendsfxf2 (op1, operands[1]));
  emit_insn(gen_extendsfxf2 (op2, operands[2]));

  emit_label (label);

  emit_insn (gen_fpremxf4 (op1, op2, op1, op2));
  ix86_emit_fp_unordered_jump (label);

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op1));
  DONE;
})

(define_expand "fmoddf3"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))
   (use (match_operand:DF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx label = gen_label_rtx ();

  rtx op1 = gen_reg_rtx (XFmode);
  rtx op2 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_extenddfxf2 (op2, operands[2]));

  emit_label (label);

  emit_insn (gen_fpremxf4 (op1, op2, op1, op2));
  ix86_emit_fp_unordered_jump (label);

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op1));
  DONE;
})

(define_expand "fmodxf3"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))
   (use (match_operand:XF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
```

```
{
  rtx label = gen_label_rtx ();

  emit_label (label);

  emit_insn (gen_fpremxf4 (operands[1], operands[2],
   operands[1], operands[2]));
  ix86_emit_fp_unordered_jump (label);

  emit_move_insn (operands[0], operands[1]);
  DONE;
})

(define_insn "fprem1xf4"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 2 "register_operand" "0")
    (match_operand:XF 3 "register_operand" "1")]
   UNSPEC_FPREM1_F))
   (set (match_operand:XF 1 "register_operand" "=u")
(unspec:XF [(match_dup 2) (match_dup 3)]
   UNSPEC_FPREM1_U))
   (set (reg:CCFP FPSR_REG)
(unspec:CCFP [(const_int 0)] UNSPEC_NOP))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fprem1"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "dremsf3"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))
   (use (match_operand:SF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx label = gen_label_rtx ();

  rtx op1 = gen_reg_rtx (XFmode);
  rtx op2 = gen_reg_rtx (XFmode);

  emit_insn(gen_extendsfxf2 (op1, operands[1]));
  emit_insn(gen_extendsfxf2 (op2, operands[2]));

  emit_label (label);

  emit_insn (gen_fprem1xf4 (op1, op2, op1, op2));
  ix86_emit_fp_unordered_jump (label);
```

```
  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op1));
  DONE;
})

(define_expand "dremdf3"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))
   (use (match_operand:DF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx label = gen_label_rtx ();

  rtx op1 = gen_reg_rtx (XFmode);
  rtx op2 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_extenddfxf2 (op2, operands[2]));

  emit_label (label);

  emit_insn (gen_fprem1xf4 (op1, op2, op1, op2));
  ix86_emit_fp_unordered_jump (label);

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op1));
  DONE;
})

(define_expand "dremxf3"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))
   (use (match_operand:XF 2 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx label = gen_label_rtx ();

  emit_label (label);

  emit_insn (gen_fprem1xf4 (operands[1], operands[2],
    operands[1], operands[2]));
  ix86_emit_fp_unordered_jump (label);

  emit_move_insn (operands[0], operands[1]);
  DONE;
})

(define_insn "*sindf2"
  [(set (match_operand:DF 0 "register_operand" "=f")
```

```
(unspec:DF [(match_operand:DF 1 "register_operand" "0")] UNSPEC_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsin"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_insn "*sinsf2"
  [(set (match_operand:SF 0 "register_operand" "=f")
(unspec:SF [(match_operand:SF 1 "register_operand" "0")] UNSPEC_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsin"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")])

(define_insn "*sinextendsfdf2"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(float_extend:DF
     (match_operand:SF 1 "register_operand" "0"))]
   UNSPEC_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsin"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_insn "*sinxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")] UNSPEC_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fsin"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_insn "*cosdf2"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(match_operand:DF 1 "register_operand" "0")] UNSPEC_COS))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fcos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_insn "*cossf2"
```

```
  [(set (match_operand:SF 0 "register_operand" "=f")
(unspec:SF [(match_operand:SF 1 "register_operand" "0")] UNSPEC_COS))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fcos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")])

(define_insn "*cosextendsfdf2"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(float_extend:DF
     (match_operand:SF 1 "register_operand" "0"))]
   UNSPEC_COS))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fcos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_insn "*cosxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")] UNSPEC_COS))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fcos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

;; With sincos pattern defined, sin and cos builtin function will be
;; expanded to sincos pattern with one of its outputs left unused.
;; Cse pass  will detected, if two sincos patterns can be combined,
;; otherwise sincos pattern will be split back to sin or cos pattern,
;; depending on the unused output.

(define_insn "sincosdf3"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(match_operand:DF 2 "register_operand" "0")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "=u")
        (unspec:DF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsincos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_split
```

```
  [(set (match_operand:DF 0 "register_operand" "")
(unspec:DF [(match_operand:DF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "")
(unspec:DF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[0]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 1) (unspec:DF [(match_dup 2)] UNSPEC_SIN))]
  "")

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(unspec:DF [(match_operand:DF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "")
(unspec:DF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[1]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 0) (unspec:DF [(match_dup 2)] UNSPEC_COS))]
  "")

(define_insn "sincossf3"
  [(set (match_operand:SF 0 "register_operand" "=f")
(unspec:SF [(match_operand:SF 2 "register_operand" "0")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:SF 1 "register_operand" "=u")
        (unspec:SF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsincos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")])

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(unspec:SF [(match_operand:SF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:SF 1 "register_operand" "")
(unspec:SF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[0]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 1) (unspec:SF [(match_dup 2)] UNSPEC_SIN))]
  "")

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(unspec:SF [(match_operand:SF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:SF 1 "register_operand" "")
```

```
(unspec:SF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[1]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 0) (unspec:SF [(match_dup 2)] UNSPEC_COS))]
  "")

(define_insn "*sincosextendsfdf3"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(float_extend:DF
     (match_operand:SF 2 "register_operand" "0"))]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "=u")
        (unspec:DF [(float_extend:DF
     (match_dup 2))] UNSPEC_SINCOS_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fsincos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(unspec:DF [(float_extend:DF
     (match_operand:SF 2 "register_operand" ""))]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "")
        (unspec:DF [(float_extend:DF
     (match_dup 2))] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[0]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 1) (unspec:DF [(float_extend:DF
   (match_dup 2))] UNSPEC_SIN))]
  "")

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(unspec:DF [(float_extend:DF
     (match_operand:SF 2 "register_operand" ""))]
   UNSPEC_SINCOS_COS))
   (set (match_operand:DF 1 "register_operand" "")
        (unspec:DF [(float_extend:DF
     (match_dup 2))] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[1]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 0) (unspec:DF [(float_extend:DF
   (match_dup 2))] UNSPEC_COS))]
  "")

(define_insn "sincosxf3"
```

```
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 2 "register_operand" "0")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:XF 1 "register_operand" "=u")
        (unspec:XF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fsincos"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_split
  [(set (match_operand:XF 0 "register_operand" "")
(unspec:XF [(match_operand:XF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:XF 1 "register_operand" "")
(unspec:XF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[0]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 1) (unspec:XF [(match_dup 2)] UNSPEC_SIN))]
  "")

(define_split
  [(set (match_operand:XF 0 "register_operand" "")
(unspec:XF [(match_operand:XF 2 "register_operand" "")]
   UNSPEC_SINCOS_COS))
   (set (match_operand:XF 1 "register_operand" "")
(unspec:XF [(match_dup 2)] UNSPEC_SINCOS_SIN))]
  "find_regno_note (insn, REG_UNUSED, REGNO (operands[1]))
   && !reload_completed && !reload_in_progress"
  [(set (match_dup 0) (unspec:XF [(match_dup 2)] UNSPEC_COS))]
  "")

(define_insn "*tandf3_1"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(match_operand:DF 2 "register_operand" "0")]
   UNSPEC_TAN_ONE))
   (set (match_operand:DF 1 "register_operand" "=u")
        (unspec:DF [(match_dup 2)] UNSPEC_TAN_TAN))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fptan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

;; optimize sequence: fptan
;;       fstp    %st(0)
;;       fld1
;; into fptan insn.
```

```
(define_peephole2
  [(parallel[(set (match_operand:DF 0 "register_operand" "")
  (unspec:DF [(match_operand:DF 2 "register_operand" "")]
     UNSPEC_TAN_ONE))
     (set (match_operand:DF 1 "register_operand" "")
  (unspec:DF [(match_dup 2)] UNSPEC_TAN_TAN))])
   (set (match_dup 0)
        (match_operand:DF 3 "immediate_operand" ""))]
  "standard_80387_constant_p (operands[3]) == 2"
  [(parallel[(set (match_dup 0) (unspec:DF [(match_dup 2)] UNSPEC_TAN_ONE))
         (set (match_dup 1) (unspec:DF [(match_dup 2)] UNSPEC_TAN_TAN))])]
  "")

(define_expand "tandf2"
  [(parallel [(set (match_dup 2)
   (unspec:DF [(match_operand:DF 1 "register_operand" "")]
      UNSPEC_TAN_ONE))
      (set (match_operand:DF 0 "register_operand" "")
   (unspec:DF [(match_dup 1)] UNSPEC_TAN_TAN))])]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (DFmode);
})

(define_insn "*tansf3_1"
  [(set (match_operand:SF 0 "register_operand" "=f")
(unspec:SF [(match_operand:SF 2 "register_operand" "0")]
   UNSPEC_TAN_ONE))
   (set (match_operand:SF 1 "register_operand" "=u")
        (unspec:SF [(match_dup 2)] UNSPEC_TAN_TAN))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fptan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")])

;; optimize sequence: fptan
;;       fstp    %st(0)
;;       fld1
;; into fptan insn.

(define_peephole2
  [(parallel[(set (match_operand:SF 0 "register_operand" "")
  (unspec:SF [(match_operand:SF 2 "register_operand" "")]
     UNSPEC_TAN_ONE))
     (set (match_operand:SF 1 "register_operand" "")
```

```
  (unspec:SF [(match_dup 2)] UNSPEC_TAN_TAN))])
   (set (match_dup 0)
        (match_operand:SF 3 "immediate_operand" ""))]
  "standard_80387_constant_p (operands[3]) == 2"
  [(parallel[(set (match_dup 0) (unspec:SF [(match_dup 2)] UNSPEC_TAN_ONE))
         (set (match_dup 1) (unspec:SF [(match_dup 2)] UNSPEC_TAN_TAN))])]
  "")

(define_expand "tansf2"
  [(parallel [(set (match_dup 2)
   (unspec:SF [(match_operand:SF 1 "register_operand" "")]
      UNSPEC_TAN_ONE))
      (set (match_operand:SF 0 "register_operand" "")
   (unspec:SF [(match_dup 1)] UNSPEC_TAN_TAN))])]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (SFmode);
})

(define_insn "*tanxf3_1"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 2 "register_operand" "0")]
   UNSPEC_TAN_ONE))
   (set (match_operand:XF 1 "register_operand" "=u")
        (unspec:XF [(match_dup 2)] UNSPEC_TAN_TAN))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fptan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

;; optimize sequence: fptan
;;       fstp    %st(0)
;;       fld1
;; into fptan insn.

(define_peephole2
  [(parallel[(set (match_operand:XF 0 "register_operand" "")
  (unspec:XF [(match_operand:XF 2 "register_operand" "")]
     UNSPEC_TAN_ONE))
     (set (match_operand:XF 1 "register_operand" "")
  (unspec:XF [(match_dup 2)] UNSPEC_TAN_TAN))])
   (set (match_dup 0)
        (match_operand:XF 3 "immediate_operand" ""))]
  "standard_80387_constant_p (operands[3]) == 2"
  [(parallel[(set (match_dup 0) (unspec:XF [(match_dup 2)] UNSPEC_TAN_ONE))
         (set (match_dup 1) (unspec:XF [(match_dup 2)] UNSPEC_TAN_TAN))])]
  "")
```

```
(define_expand "tanxf2"
  [(parallel [(set (match_dup 2)
   (unspec:XF [(match_operand:XF 1 "register_operand" "")]
      UNSPEC_TAN_ONE))
      (set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 1)] UNSPEC_TAN_TAN))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
})

(define_insn "atan2df3_1"
  [(set (match_operand:DF 0 "register_operand" "=f")
(unspec:DF [(match_operand:DF 2 "register_operand" "0")
    (match_operand:DF 1 "register_operand" "u")]
   UNSPEC_FPATAN))
   (clobber (match_scratch:DF 3 "=1"))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fpatan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DF")])

(define_expand "atan2df3"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 2 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx copy = gen_reg_rtx (DFmode);
  emit_move_insn (copy, operands[1]);
  emit_insn (gen_atan2df3_1 (operands[0], copy, operands[2]));
  DONE;
})

(define_expand "atandf2"
  [(parallel [(set (match_operand:DF 0 "register_operand" "")
   (unspec:DF [(match_dup 2)
       (match_operand:DF 1 "register_operand" "")]
    UNSPEC_FPATAN))
      (clobber (match_scratch:DF 3 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
```

```
  operands[2] = gen_reg_rtx (DFmode);
  emit_move_insn (operands[2], CONST1_RTX (DFmode));  /* fld1 */
})

(define_insn "atan2sf3_1"
  [(set (match_operand:SF 0 "register_operand" "=f")
        (unspec:SF [(match_operand:SF 2 "register_operand" "0")
    (match_operand:SF 1 "register_operand" "u")]
   UNSPEC_FPATAN))
   (clobber (match_scratch:SF 3 "=1"))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "fpatan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "SF")])

(define_expand "atan2sf3"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 2 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx copy = gen_reg_rtx (SFmode);
  emit_move_insn (copy, operands[1]);
  emit_insn (gen_atan2sf3_1 (operands[0], copy, operands[2]));
  DONE;
})

(define_expand "atansf2"
  [(parallel [(set (match_operand:SF 0 "register_operand" "")
   (unspec:SF [(match_dup 2)
       (match_operand:SF 1 "register_operand" "")]
    UNSPEC_FPATAN))
      (clobber (match_scratch:SF 3 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (SFmode);
  emit_move_insn (operands[2], CONST1_RTX (SFmode));  /* fld1 */
})

(define_insn "atan2xf3_1"
  [(set (match_operand:XF 0 "register_operand" "=f")
        (unspec:XF [(match_operand:XF 2 "register_operand" "0")
            (match_operand:XF 1 "register_operand" "u")]
          UNSPEC_FPATAN))
```

```
   (clobber (match_scratch:XF 3 "=1"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fpatan"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "atan2xf3"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 2 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx copy = gen_reg_rtx (XFmode);
  emit_move_insn (copy, operands[1]);
  emit_insn (gen_atan2xf3_1 (operands[0], copy, operands[2]));
  DONE;
})

(define_expand "atanxf2"
  [(parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 2)
       (match_operand:XF 1 "register_operand" "")]
    UNSPEC_FPATAN))
      (clobber (match_scratch:XF 3 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  emit_move_insn (operands[2], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "asindf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 3) (mult:XF (match_dup 2) (match_dup 2)))
   (set (match_dup 5) (minus:XF (match_dup 4) (match_dup 3)))
   (set (match_dup 6) (sqrt:XF (match_dup 5)))
   (parallel [(set (match_dup 7)
            (unspec:XF [(match_dup 6) (match_dup 2)]
      UNSPEC_FPATAN))
          (clobber (match_scratch:XF 8 ""))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 7)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;
```

```
  for (i=2; i<8; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[4], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "asinsf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 3) (mult:XF (match_dup 2) (match_dup 2)))
   (set (match_dup 5) (minus:XF (match_dup 4) (match_dup 3)))
   (set (match_dup 6) (sqrt:XF (match_dup 5)))
   (parallel [(set (match_dup 7)
                   (unspec:XF [(match_dup 6) (match_dup 2)]
      UNSPEC_FPATAN))
              (clobber (match_scratch:XF 8 ""))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 7)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<8; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[4], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "asinxf2"
  [(set (match_dup 2)
(mult:XF (match_operand:XF 1 "register_operand" "")
 (match_dup 1)))
   (set (match_dup 4) (minus:XF (match_dup 3) (match_dup 2)))
   (set (match_dup 5) (sqrt:XF (match_dup 4)))
   (parallel [(set (match_operand:XF 0 "register_operand" "")
                   (unspec:XF [(match_dup 5) (match_dup 1)]
      UNSPEC_FPATAN))
              (clobber (match_scratch:XF 6 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<6; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[3], CONST1_RTX (XFmode));  /* fld1 */
```

```
})

(define_expand "acosdf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 3) (mult:XF (match_dup 2) (match_dup 2)))
   (set (match_dup 5) (minus:XF (match_dup 4) (match_dup 3)))
   (set (match_dup 6) (sqrt:XF (match_dup 5)))
   (parallel [(set (match_dup 7)
             (unspec:XF [(match_dup 2) (match_dup 6)]
      UNSPEC_FPATAN))
          (clobber (match_scratch:XF 8 ""))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 7)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<8; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[4], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "acossf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 3) (mult:XF (match_dup 2) (match_dup 2)))
   (set (match_dup 5) (minus:XF (match_dup 4) (match_dup 3)))
   (set (match_dup 6) (sqrt:XF (match_dup 5)))
   (parallel [(set (match_dup 7)
             (unspec:XF [(match_dup 2) (match_dup 6)]
      UNSPEC_FPATAN))
          (clobber (match_scratch:XF 8 ""))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 7)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<8; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[4], CONST1_RTX (XFmode));  /* fld1 */
})
```

```
(define_expand "acosxf2"
  [(set (match_dup 2)
(mult:XF (match_operand:XF 1 "register_operand" "")
 (match_dup 1)))
   (set (match_dup 4) (minus:XF (match_dup 3) (match_dup 2)))
   (set (match_dup 5) (sqrt:XF (match_dup 4)))
   (parallel [(set (match_operand:XF 0 "register_operand" "")
             (unspec:XF [(match_dup 1) (match_dup 5)]
      UNSPEC_FPATAN))
          (clobber (match_scratch:XF 6 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<6; i++)
    operands[i] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[3], CONST1_RTX (XFmode));  /* fld1 */
})

(define_insn "fyl2x_xf3"
  [(set (match_operand:XF 0 "register_operand" "=f")
        (unspec:XF [(match_operand:XF 2 "register_operand" "0")
    (match_operand:XF 1 "register_operand" "u")]
           UNSPEC_FYL2X))
   (clobber (match_scratch:XF 3 "=1"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fyl2x"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "logsf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;

  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
```

```
  operands[4] = gen_reg_rtx (XFmode);

  temp = standard_80387_constant_rtx (4); /* fldln2 */
  emit_move_insn (operands[3], temp);
})

(define_expand "logdf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;

  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);

  temp = standard_80387_constant_rtx (4); /* fldln2 */
  emit_move_insn (operands[3], temp);
})

(define_expand "logxf2"
  [(parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_operand:XF 1 "register_operand" "")
       (match_dup 2)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 3 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx temp;

  operands[2] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (4); /* fldln2 */
  emit_move_insn (operands[2], temp);
})

(define_expand "log10sf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
```

```
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;

  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);

  temp = standard_80387_constant_rtx (3); /* fldlg2 */
  emit_move_insn (operands[3], temp);
})

(define_expand "log10df2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;

  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);

  temp = standard_80387_constant_rtx (3); /* fldlg2 */
  emit_move_insn (operands[3], temp);
})

(define_expand "log10xf2"
  [(parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_operand:XF 1 "register_operand" "")
       (match_dup 2)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 3 ""))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx temp;
```

```
  operands[2] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (3); /* fldlg2 */
  emit_move_insn (operands[2], temp);
})

(define_expand "log2sf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[3], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "log2df2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (parallel [(set (match_dup 4)
   (unspec:XF [(match_dup 2)
       (match_dup 3)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 5 ""))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);

  emit_move_insn (operands[3], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "log2xf2"
  [(parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_operand:XF 1 "register_operand" "")
       (match_dup 2)] UNSPEC_FYL2X))
      (clobber (match_scratch:XF 3 ""))])]
```

```
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  emit_move_insn (operands[2], CONST1_RTX (XFmode));  /* fld1 */
})

(define_insn "fyl2xp1_xf3"
  [(set (match_operand:XF 0 "register_operand" "=f")
        (unspec:XF [(match_operand:XF 2 "register_operand" "0")
    (match_operand:XF 1 "register_operand" "u")]
           UNSPEC_FYL2XP1))
   (clobber (match_scratch:XF 3 "=1"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fyl2xp1"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "log1psf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
  ix86_emit_i387_log1p (op0, op1);
  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "log1pdf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  ix86_emit_i387_log1p (op0, op1);
  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})
```

```
(define_expand "log1pxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  ix86_emit_i387_log1p (operands[0], operands[1]);
  DONE;
})

(define_insn "*fxtractxf3"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 2 "register_operand" "0")]
   UNSPEC_XTRACT_FRACT))
   (set (match_operand:XF 1 "register_operand" "=u")
        (unspec:XF [(match_dup 2)] UNSPEC_XTRACT_EXP))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fxtract"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "logbsf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (parallel [(set (match_dup 3)
   (unspec:XF [(match_dup 2)] UNSPEC_XTRACT_FRACT))
      (set (match_dup 4)
   (unspec:XF [(match_dup 2)] UNSPEC_XTRACT_EXP))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 4)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);
})

(define_expand "logbdf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (parallel [(set (match_dup 3)
   (unspec:XF [(match_dup 2)] UNSPEC_XTRACT_FRACT))
      (set (match_dup 4)
   (unspec:XF [(match_dup 2)] UNSPEC_XTRACT_EXP))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 4)))]
```

```
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
  operands[4] = gen_reg_rtx (XFmode);
})

(define_expand "logbxf2"
  [(parallel [(set (match_dup 2)
   (unspec:XF [(match_operand:XF 1 "register_operand" "")]
      UNSPEC_XTRACT_FRACT))
      (set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 1)] UNSPEC_XTRACT_EXP))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
})

(define_expand "ilogbsi2"
  [(parallel [(set (match_dup 2)
   (unspec:XF [(match_operand:XF 1 "register_operand" "")]
      UNSPEC_XTRACT_FRACT))
      (set (match_operand:XF 3 "register_operand" "")
   (unspec:XF [(match_dup 1)] UNSPEC_XTRACT_EXP))])
   (parallel [(set (match_operand:SI 0 "register_operand" "")
           (fix:SI (match_dup 3)))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  operands[2] = gen_reg_rtx (XFmode);
  operands[3] = gen_reg_rtx (XFmode);
})

(define_insn "*f2xm1xf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_F2XM1))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "f2xm1"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_insn "*fscalexf4"
  [(set (match_operand:XF 0 "register_operand" "=f")
```

```
(unspec:XF [(match_operand:XF 2 "register_operand" "0")
    (match_operand:XF 3 "register_operand" "1")]
   UNSPEC_FSCALE_FRACT))
   (set (match_operand:XF 1 "register_operand" "=u")
(unspec:XF [(match_dup 2) (match_dup 3)]
   UNSPEC_FSCALE_EXP))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fscale"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "expsf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (set (match_dup 9) (plus:XF (match_dup 7) (match_dup 8)))
   (parallel [(set (match_dup 10)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 11)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 10)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<12; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[8], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "expdf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (set (match_dup 9) (plus:XF (match_dup 7) (match_dup 8)))
```

```
   (parallel [(set (match_dup 10)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 11)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 10)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<12; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[8], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "expxf2"
  [(set (match_dup 3) (mult:XF (match_operand:XF 1 "register_operand" "")
       (match_dup 2)))
   (set (match_dup 4) (unspec:XF [(match_dup 3)] UNSPEC_FRNDINT))
   (set (match_dup 5) (minus:XF (match_dup 3) (match_dup 4)))
   (set (match_dup 6) (unspec:XF [(match_dup 5)] UNSPEC_F2XM1))
   (set (match_dup 8) (plus:XF (match_dup 6) (match_dup 7)))
   (parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 8) (match_dup 4)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 9)
   (unspec:XF [(match_dup 8) (match_dup 4)]
      UNSPEC_FSCALE_EXP))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<10; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[2], temp);
  emit_move_insn (operands[7], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp10sf2"
  [(set (match_dup 2)
```

```
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (set (match_dup 9) (plus:XF (match_dup 7) (match_dup 8)))
   (parallel [(set (match_dup 10)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 11)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 10)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<12; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (6); /* fldl2t */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[8], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp10df2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (set (match_dup 9) (plus:XF (match_dup 7) (match_dup 8)))
   (parallel [(set (match_dup 10)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 11)
   (unspec:XF [(match_dup 9) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 10)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;
```

```
  for (i=2; i<12; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (6); /* fldl2t */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[8], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp10xf2"
  [(set (match_dup 3) (mult:XF (match_operand:XF 1 "register_operand" "")
       (match_dup 2)))
   (set (match_dup 4) (unspec:XF [(match_dup 3)] UNSPEC_FRNDINT))
   (set (match_dup 5) (minus:XF (match_dup 3) (match_dup 4)))
   (set (match_dup 6) (unspec:XF [(match_dup 5)] UNSPEC_F2XM1))
   (set (match_dup 8) (plus:XF (match_dup 6) (match_dup 7)))
   (parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 8) (match_dup 4)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 9)
   (unspec:XF [(match_dup 8) (match_dup 4)]
      UNSPEC_FSCALE_EXP))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<10; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (6); /* fldl2t */
  emit_move_insn (operands[2], temp);
  emit_move_insn (operands[7], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp2sf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 3) (unspec:XF [(match_dup 2)] UNSPEC_FRNDINT))
   (set (match_dup 4) (minus:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_F2XM1))
   (set (match_dup 7) (plus:XF (match_dup 5) (match_dup 6)))
   (parallel [(set (match_dup 8)
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 9)
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 8)))]
  "TARGET_USE_FANCY_MATH_387
```

```
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<10; i++)
    operands[i] = gen_reg_rtx (XFmode);
  emit_move_insn (operands[6], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp2df2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 3) (unspec:XF [(match_dup 2)] UNSPEC_FRNDINT))
   (set (match_dup 4) (minus:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_F2XM1))
   (set (match_dup 7) (plus:XF (match_dup 5) (match_dup 6)))
   (parallel [(set (match_dup 8)
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 9)
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 8)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=2; i<10; i++)
    operands[i] = gen_reg_rtx (XFmode);
  emit_move_insn (operands[6], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "exp2xf2"
  [(set (match_dup 2) (match_operand:XF 1 "register_operand" ""))
   (set (match_dup 3) (unspec:XF [(match_dup 2)] UNSPEC_FRNDINT))
   (set (match_dup 4) (minus:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_F2XM1))
   (set (match_dup 7) (plus:XF (match_dup 5) (match_dup 6)))
   (parallel [(set (match_operand:XF 0 "register_operand" "")
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 8)
   (unspec:XF [(match_dup 7) (match_dup 3)]
      UNSPEC_FSCALE_EXP))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
```

```
{
  int i;

  for (i=2; i<9; i++)
    operands[i] = gen_reg_rtx (XFmode);
  emit_move_insn (operands[6], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "expm1df2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (parallel [(set (match_dup 8)
   (unspec:XF [(match_dup 7) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
   (set (match_dup 9)
   (unspec:XF [(match_dup 7) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (parallel [(set (match_dup 11)
   (unspec:XF [(match_dup 10) (match_dup 9)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 12)
   (unspec:XF [(match_dup 10) (match_dup 9)]
      UNSPEC_FSCALE_EXP))])
   (set (match_dup 13) (minus:XF (match_dup 11) (match_dup 10)))
   (set (match_dup 14) (plus:XF (match_dup 13) (match_dup 8)))
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 14)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<15; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[10], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "expm1sf2"
  [(set (match_dup 2)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 4) (mult:XF (match_dup 2) (match_dup 3)))
   (set (match_dup 5) (unspec:XF [(match_dup 4)] UNSPEC_FRNDINT))
```

```
   (set (match_dup 6) (minus:XF (match_dup 4) (match_dup 5)))
   (set (match_dup 7) (unspec:XF [(match_dup 6)] UNSPEC_F2XM1))
   (parallel [(set (match_dup 8)
   (unspec:XF [(match_dup 7) (match_dup 5)]
      UNSPEC_FSCALE_FRACT))
   (set (match_dup 9)
   (unspec:XF [(match_dup 7) (match_dup 5)]
      UNSPEC_FSCALE_EXP))])
   (parallel [(set (match_dup 11)
   (unspec:XF [(match_dup 10) (match_dup 9)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 12)
   (unspec:XF [(match_dup 10) (match_dup 9)]
      UNSPEC_FSCALE_EXP))])
   (set (match_dup 13) (minus:XF (match_dup 11) (match_dup 10)))
   (set (match_dup 14) (plus:XF (match_dup 13) (match_dup 8)))
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 14)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<15; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[3], temp);
  emit_move_insn (operands[10], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "expm1xf2"
  [(set (match_dup 3) (mult:XF (match_operand:XF 1 "register_operand" "")
       (match_dup 2)))
   (set (match_dup 4) (unspec:XF [(match_dup 3)] UNSPEC_FRNDINT))
   (set (match_dup 5) (minus:XF (match_dup 3) (match_dup 4)))
   (set (match_dup 6) (unspec:XF [(match_dup 5)] UNSPEC_F2XM1))
   (parallel [(set (match_dup 7)
   (unspec:XF [(match_dup 6) (match_dup 4)]
      UNSPEC_FSCALE_FRACT))
   (set (match_dup 8)
   (unspec:XF [(match_dup 6) (match_dup 4)]
      UNSPEC_FSCALE_EXP))])
   (parallel [(set (match_dup 10)
   (unspec:XF [(match_dup 9) (match_dup 8)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 11)
   (unspec:XF [(match_dup 9) (match_dup 8)]
      UNSPEC_FSCALE_EXP))])
```

```
   (set (match_dup 12) (minus:XF (match_dup 10) (match_dup 9)))
   (set (match_operand:XF 0 "register_operand" "")
(plus:XF (match_dup 12) (match_dup 7)))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  rtx temp;
  int i;

  for (i=2; i<13; i++)
    operands[i] = gen_reg_rtx (XFmode);
  temp = standard_80387_constant_rtx (5); /* fldl2e */
  emit_move_insn (operands[2], temp);
  emit_move_insn (operands[9], CONST1_RTX (XFmode));  /* fld1 */
})

(define_expand "ldexpdf3"
  [(set (match_dup 3)
(float_extend:XF (match_operand:DF 1 "register_operand" "")))
   (set (match_dup 4)
(float:XF (match_operand:SI 2 "register_operand" "")))
   (parallel [(set (match_dup 5)
   (unspec:XF [(match_dup 3) (match_dup 4)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 6)
   (unspec:XF [(match_dup 3) (match_dup 4)]
      UNSPEC_FSCALE_EXP))])
   (set (match_operand:DF 0 "register_operand" "")
(float_truncate:DF (match_dup 5)))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=3; i<7; i++)
    operands[i] = gen_reg_rtx (XFmode);
})

(define_expand "ldexpsf3"
  [(set (match_dup 3)
(float_extend:XF (match_operand:SF 1 "register_operand" "")))
   (set (match_dup 4)
(float:XF (match_operand:SI 2 "register_operand" "")))
   (parallel [(set (match_dup 5)
   (unspec:XF [(match_dup 3) (match_dup 4)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 6)
   (unspec:XF [(match_dup 3) (match_dup 4)]
      UNSPEC_FSCALE_EXP))])
```

```
   (set (match_operand:SF 0 "register_operand" "")
(float_truncate:SF (match_dup 5)))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=3; i<7; i++)
    operands[i] = gen_reg_rtx (XFmode);
})

(define_expand "ldexpxf3"
  [(set (match_dup 3)
(float:XF (match_operand:SI 2 "register_operand" "")))
   (parallel [(set (match_operand:XF 0 " register_operand" "")
   (unspec:XF [(match_operand:XF 1 "register_operand" "")
       (match_dup 3)]
      UNSPEC_FSCALE_FRACT))
      (set (match_dup 4)
   (unspec:XF [(match_dup 1) (match_dup 3)]
      UNSPEC_FSCALE_EXP))])]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  int i;

  for (i=3; i<5; i++)
    operands[i] = gen_reg_rtx (XFmode);
})


(define_insn "frndintxf2"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "frndint"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "XF")])

(define_expand "rintdf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
```

```
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2 (op0, op1));

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "rintsf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2 (op0, op1));

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "rintxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  emit_insn (gen_frndintxf2 (operands[0], operands[1]));
  DONE;
})

(define_insn_and_split "*fist<mode>2_1"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  if (memory_operand (operands[0], VOIDmode))
    emit_insn (gen_fist<mode>2 (operands[0], operands[1]));
  else
```

```
    {
      operands[2] = assign_386_stack_local (<MODE>mode, SLOT_TEMP);
      emit_insn (gen_fist<mode>2_with_temp (operands[0], operands[1],
    operands[2]));
    }
  DONE;
}
  [(set_attr "type" "fpspc")
   (set_attr "mode" "<MODE>")])

(define_insn "fistdi2"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST))
   (clobber (match_scratch:XF 2 "=&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DI")])

(define_insn "fistdi2_with_temp"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=m,?r")
(unspec:DI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST))
   (clobber (match_operand:DI 2 "memory_operand" "=m,m"))
   (clobber (match_scratch:XF 3 "=&1f,&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST))
   (clobber (match_operand:DI 2 "memory_operand" ""))
   (clobber (match_scratch 3 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 2) (unspec:DI [(match_dup 1)] UNSPEC_FIST))
      (clobber (match_dup 3))])
   (set (match_dup 0) (match_dup 2))]
  "")

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST))
   (clobber (match_operand:DI 2 "memory_operand" ""))
```

```
   (clobber (match_scratch 3 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (unspec:DI [(match_dup 1)] UNSPEC_FIST))
      (clobber (match_dup 3))])]
  "")

(define_insn "fist<mode>2"
  [(set (match_operand:X87MODEI12 0 "memory_operand" "=m")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "<MODE>")])

(define_insn "fist<mode>2_with_temp"
  [(set (match_operand:X87MODEI12 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST))
   (clobber (match_operand:X87MODEI12 2 "memory_operand" "=m,m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fpspc")
   (set_attr "mode" "<MODE>")])

(define_split
  [(set (match_operand:X87MODEI12 0 "register_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST))
   (clobber (match_operand:X87MODEI12 2 "memory_operand" ""))]
  "reload_completed"
  [(set (match_dup 2) (unspec:X87MODEI12 [(match_dup 1)]
       UNSPEC_FIST))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_split
  [(set (match_operand:X87MODEI12 0 "memory_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST))
   (clobber (match_operand:X87MODEI12 2 "memory_operand" ""))]
  "reload_completed"
  [(set (match_dup 0) (unspec:X87MODEI12 [(match_dup 1)]
       UNSPEC_FIST))]
  "")

(define_expand "lrint<mode>2"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "")
```

```
(unspec:X87MODEI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "")

;; Rounding mode control word calculation could clobber FLAGS_REG.
(define_insn_and_split "frndintxf2_floor"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_FLOOR))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_FLOOR] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_FLOOR);

  emit_insn (gen_frndintxf2_floor_i387 (operands[0], operands[1],
operands[2], operands[3]));
  DONE;
}
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "XF")])

(define_insn "frndintxf2_floor_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_FLOOR))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fldcw\t%3\n\tfrndint\n\tfldcw\t%2"
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "XF")])

(define_expand "floorxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
```

```
   && flag_unsafe_math_optimizations"
{
  emit_insn (gen_frndintxf2_floor (operands[0], operands[1]));
  DONE;
})

(define_expand "floordf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_floor (op0, op1));

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "floorsf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_floor (op0, op1));

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})

(define_insn_and_split "*fist<mode>2_floor_1"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_FLOOR))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
```

```
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_FLOOR] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_FLOOR);
  if (memory_operand (operands[0], VOIDmode))
    emit_insn (gen_fist<mode>2_floor (operands[0], operands[1],
      operands[2], operands[3]));
  else
    {
      operands[4] = assign_386_stack_local (<MODE>mode, SLOT_TEMP);
      emit_insn (gen_fist<mode>2_floor_with_temp (operands[0], operands[1],
  operands[2], operands[3],
  operands[4]));
    }
  DONE;
}
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "<MODE>")])

(define_insn "fistdi2_floor"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))
   (clobber (match_scratch:XF 4 "=&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "DI")])

(define_insn "fistdi2_floor_with_temp"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=m,?r")
(unspec:DI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" "m,m"))
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:DI 4 "memory_operand" "=m,m"))
   (clobber (match_scratch:XF 5 "=&1f,&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "DI")])
```

```
(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 4) (unspec:DI [(match_dup 1)] UNSPEC_FIST_FLOOR))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (unspec:DI [(match_dup 1)] UNSPEC_FIST_FLOOR))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])]
  "")

(define_insn "fist<mode>2_floor"
  [(set (match_operand:X87MODEI12 0 "memory_operand" "=m")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "<MODE>")])

(define_insn "fist<mode>2_floor_with_temp"
  [(set (match_operand:X87MODEI12 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" "m,m"))
```

```
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" "=m,m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "floor")
   (set_attr "mode" "<MODE>")])

(define_split
  [(set (match_operand:X87MODEI12 0 "register_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 4) (unspec:X87MODEI12 [(match_dup 1)]
  UNSPEC_FIST_FLOOR))
      (use (match_dup 2))
      (use (match_dup 3))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:X87MODEI12 0 "memory_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_FLOOR))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (unspec:X87MODEI12 [(match_dup 1)]
  UNSPEC_FIST_FLOOR))
      (use (match_dup 2))
      (use (match_dup 3))])]
  "")

(define_expand "lfloor<mode>2"
  [(parallel [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "")
   (unspec:X87MODEI [(match_operand:XF 1 "register_operand" "")]
    UNSPEC_FIST_FLOOR))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "")

;; Rounding mode control word calculation could clobber FLAGS_REG.
(define_insn_and_split "frndintxf2_ceil"
```

```
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_CEIL))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_CEIL] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_CEIL);

  emit_insn (gen_frndintxf2_ceil_i387 (operands[0], operands[1],
       operands[2], operands[3]));
  DONE;
}
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "XF")])

(define_insn "frndintxf2_ceil_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_CEIL))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fldcw\t%3\n\tfrndint\n\tfldcw\t%2"
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "XF")])

(define_expand "ceilxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  emit_insn (gen_frndintxf2_ceil (operands[0], operands[1]));
  DONE;
})

(define_expand "ceildf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
```

```
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_ceil (op0, op1));

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "ceilsf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_ceil (op0, op1));

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})

(define_insn_and_split "*fist<mode>2_ceil_1"
  [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_CEIL))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_CEIL] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_CEIL);
  if (memory_operand (operands[0], VOIDmode))
    emit_insn (gen_fist<mode>2_ceil (operands[0], operands[1],
     operands[2], operands[3]));
```

```
  else
    {
      operands[4] = assign_386_stack_local (<MODE>mode, SLOT_TEMP);
      emit_insn (gen_fist<mode>2_ceil_with_temp (operands[0], operands[1],
 operands[2], operands[3],
 operands[4]));
    }
  DONE;
}
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "<MODE>")])

(define_insn "fistdi2_ceil"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))
   (clobber (match_scratch:XF 4 "=&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "DI")])

(define_insn "fistdi2_ceil_with_temp"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=m,?r")
(unspec:DI [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" "m,m"))
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:DI 4 "memory_operand" "=m,m"))
   (clobber (match_scratch:XF 5 "=&1f,&1f"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "DI")])

(define_split
  [(set (match_operand:DI 0 "register_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
```

```
  "reload_completed"
  [(parallel [(set (match_dup 4) (unspec:DI [(match_dup 1)] UNSPEC_FIST_CEIL))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:DI 0 "memory_operand" "")
(unspec:DI [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:DI 4 "memory_operand" ""))
   (clobber (match_scratch 5 ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (unspec:DI [(match_dup 1)] UNSPEC_FIST_CEIL))
      (use (match_dup 2))
      (use (match_dup 3))
      (clobber (match_dup 5))])]
  "")

(define_insn "fist<mode>2_ceil"
  [(set (match_operand:X87MODEI12 0 "memory_operand" "=m")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "* return output_fix_trunc (insn, operands, 0);"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "<MODE>")])

(define_insn "fist<mode>2_ceil_with_temp"
  [(set (match_operand:X87MODEI12 0 "nonimmediate_operand" "=m,?r")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "f,f")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" "m,m"))
   (use (match_operand:HI 3 "memory_operand" "m,m"))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" "=m,m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "#"
  [(set_attr "type" "fistp")
   (set_attr "i387_cw" "ceil")
   (set_attr "mode" "<MODE>")])
```

```
(define_split
  [(set (match_operand:X87MODEI12 0 "register_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 4) (unspec:X87MODEI12 [(match_dup 1)]
  UNSPEC_FIST_CEIL))
      (use (match_dup 2))
      (use (match_dup 3))])
   (set (match_dup 0) (match_dup 4))]
  "")

(define_split
  [(set (match_operand:X87MODEI12 0 "memory_operand" "")
(unspec:X87MODEI12 [(match_operand:XF 1 "register_operand" "")]
 UNSPEC_FIST_CEIL))
   (use (match_operand:HI 2 "memory_operand" ""))
   (use (match_operand:HI 3 "memory_operand" ""))
   (clobber (match_operand:X87MODEI12 4 "memory_operand" ""))]
  "reload_completed"
  [(parallel [(set (match_dup 0) (unspec:X87MODEI12 [(match_dup 1)]
  UNSPEC_FIST_CEIL))
      (use (match_dup 2))
      (use (match_dup 3))])]
  "")

(define_expand "lceil<mode>2"
  [(parallel [(set (match_operand:X87MODEI 0 "nonimmediate_operand" "")
   (unspec:X87MODEI [(match_operand:XF 1 "register_operand" "")]
    UNSPEC_FIST_CEIL))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
  "")

;; Rounding mode control word calculation could clobber FLAGS_REG.
(define_insn_and_split "frndintxf2_trunc"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_TRUNC))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
```

```
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_TRUNC] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_TRUNC);

  emit_insn (gen_frndintxf2_trunc_i387 (operands[0], operands[1],
operands[2], operands[3]));
  DONE;
}
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "XF")])

(define_insn "frndintxf2_trunc_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_TRUNC))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fldcw\t%3\n\tfrndint\n\tfldcw\t%2"
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "trunc")
   (set_attr "mode" "XF")])

(define_expand "btruncxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  emit_insn (gen_frndintxf2_trunc (operands[0], operands[1]));
  DONE;
})

(define_expand "btruncdf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_trunc (op0, op1));
```

```
  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "btruncsf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_trunc (op0, op1));

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})

;; Rounding mode control word calculation could clobber FLAGS_REG.
(define_insn_and_split "frndintxf2_mask_pm"
  [(set (match_operand:XF 0 "register_operand" "=f")
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_MASK_PM))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations
   && !(reload_completed || reload_in_progress)"
  "#"
  "&& 1"
  [(const_int 0)]
{
  ix86_optimize_mode_switching[I387_MASK_PM] = 1;

  operands[2] = assign_386_stack_local (HImode, SLOT_CW_STORED);
  operands[3] = assign_386_stack_local (HImode, SLOT_CW_MASK_PM);

  emit_insn (gen_frndintxf2_mask_pm_i387 (operands[0], operands[1],
  operands[2], operands[3]));
  DONE;
}
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "mask_pm")
   (set_attr "mode" "XF")])

(define_insn "frndintxf2_mask_pm_i387"
  [(set (match_operand:XF 0 "register_operand" "=f")
```

```
(unspec:XF [(match_operand:XF 1 "register_operand" "0")]
 UNSPEC_FRNDINT_MASK_PM))
   (use (match_operand:HI 2 "memory_operand" "m"))
   (use (match_operand:HI 3 "memory_operand" "m"))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
  "fldcw\t%3\n\tfrndint\n\tfclex\n\tfldcw\t%2"
  [(set_attr "type" "frndint")
   (set_attr "i387_cw" "mask_pm")
   (set_attr "mode" "XF")])

(define_expand "nearbyintxf2"
  [(use (match_operand:XF 0 "register_operand" ""))
   (use (match_operand:XF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && flag_unsafe_math_optimizations"
{
  emit_insn (gen_frndintxf2_mask_pm (operands[0], operands[1]));

  DONE;
})

(define_expand "nearbyintdf2"
  [(use (match_operand:DF 0 "register_operand" ""))
   (use (match_operand:DF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!(TARGET_SSE2 && TARGET_SSE_MATH) || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extenddfxf2 (op1, operands[1]));
  emit_insn (gen_frndintxf2_mask_pm (op0, op1));

  emit_insn (gen_truncxfdf2_i387_noop (operands[0], op0));
  DONE;
})

(define_expand "nearbyintsf2"
  [(use (match_operand:SF 0 "register_operand" ""))
   (use (match_operand:SF 1 "register_operand" ""))]
  "TARGET_USE_FANCY_MATH_387
   && (!TARGET_SSE_MATH || TARGET_MIX_SSE_I387)
   && flag_unsafe_math_optimizations"
{
  rtx op0 = gen_reg_rtx (XFmode);
  rtx op1 = gen_reg_rtx (XFmode);

  emit_insn (gen_extendsfxf2 (op1, operands[1]));
```

```
  emit_insn (gen_frndintxf2_mask_pm (op0, op1));

  emit_insn (gen_truncxfsf2_i387_noop (operands[0], op0));
  DONE;
})


;; Block operation instructions

(define_insn "cld"
 [(set (reg:SI DIRFLAG_REG) (const_int 0))]
 ""
 "cld"
  [(set_attr "type" "cld")])

(define_expand "movmemsi"
  [(use (match_operand:BLK 0 "memory_operand" ""))
   (use (match_operand:BLK 1 "memory_operand" ""))
   (use (match_operand:SI 2 "nonmemory_operand" ""))
   (use (match_operand:SI 3 "const_int_operand" ""))]
  "! optimize_size || TARGET_INLINE_ALL_STRINGOPS"
{
 if (ix86_expand_movmem (operands[0], operands[1], operands[2], operands[3]))
   DONE;
 else
   FAIL;
})

(define_expand "movmemdi"
  [(use (match_operand:BLK 0 "memory_operand" ""))
   (use (match_operand:BLK 1 "memory_operand" ""))
   (use (match_operand:DI 2 "nonmemory_operand" ""))
   (use (match_operand:DI 3 "const_int_operand" ""))]
  "TARGET_64BIT"
{
 if (ix86_expand_movmem (operands[0], operands[1], operands[2], operands[3]))
   DONE;
 else
   FAIL;
})

;; Most CPUs don't like single string operations
;; Handle this case here to simplify previous expander.

(define_expand "strmov"
  [(set (match_dup 4) (match_operand 3 "memory_operand" ""))
   (set (match_operand 1 "memory_operand" "") (match_dup 4))
   (parallel [(set (match_operand 0 "register_operand" "") (match_dup 5))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (match_operand 2 "register_operand" "") (match_dup 6))
```

```
      (clobber (reg:CC FLAGS_REG))])]
  ""
{
  rtx adjust = GEN_INT (GET_MODE_SIZE (GET_MODE (operands[1])));

  /* If .md ever supports :P for Pmode, these can be directly
     in the pattern above.  */
  operands[5] = gen_rtx_PLUS (Pmode, operands[0], adjust);
  operands[6] = gen_rtx_PLUS (Pmode, operands[2], adjust);

  if (TARGET_SINGLE_STRINGOP || optimize_size)
    {
      emit_insn (gen_strmov_singleop (operands[0], operands[1],
      operands[2], operands[3],
      operands[5], operands[6]));
      DONE;
    }

  operands[4] = gen_reg_rtx (GET_MODE (operands[1]));
})

(define_expand "strmov_singleop"
  [(parallel [(set (match_operand 1 "memory_operand" "")
   (match_operand 3 "memory_operand" ""))
      (set (match_operand 0 "register_operand" "")
   (match_operand 4 "" ""))
      (set (match_operand 2 "register_operand" "")
   (match_operand 5 "" ""))
      (use (reg:SI DIRFLAG_REG))])]
  "TARGET_SINGLE_STRINGOP || optimize_size"
  "")

(define_insn "*strmovdi_rex_1"
  [(set (mem:DI (match_operand:DI 2 "register_operand" "0"))
(mem:DI (match_operand:DI 3 "register_operand" "1")))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 2)
 (const_int 8)))
   (set (match_operand:DI 1 "register_operand" "=S")
(plus:DI (match_dup 3)
 (const_int 8)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "movsq"
  [(set_attr "type" "str")
   (set_attr "mode" "DI")
   (set_attr "memory" "both")])

(define_insn "*strmovsi_1"
  [(set (mem:SI (match_operand:SI 2 "register_operand" "0"))
```

```
(mem:SI (match_operand:SI 3 "register_operand" "1")))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 2)
 (const_int 4)))
   (set (match_operand:SI 1 "register_operand" "=S")
(plus:SI (match_dup 3)
 (const_int 4)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "{movsl|movsd}"
  [(set_attr "type" "str")
   (set_attr "mode" "SI")
   (set_attr "memory" "both")])

(define_insn "*strmovsi_rex_1"
  [(set (mem:SI (match_operand:DI 2 "register_operand" "0"))
(mem:SI (match_operand:DI 3 "register_operand" "1")))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 2)
 (const_int 4)))
   (set (match_operand:DI 1 "register_operand" "=S")
(plus:DI (match_dup 3)
 (const_int 4)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "{movsl|movsd}"
  [(set_attr "type" "str")
   (set_attr "mode" "SI")
   (set_attr "memory" "both")])

(define_insn "*strmovhi_1"
  [(set (mem:HI (match_operand:SI 2 "register_operand" "0"))
(mem:HI (match_operand:SI 3 "register_operand" "1")))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 2)
 (const_int 2)))
   (set (match_operand:SI 1 "register_operand" "=S")
(plus:SI (match_dup 3)
 (const_int 2)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "movsw"
  [(set_attr "type" "str")
   (set_attr "memory" "both")
   (set_attr "mode" "HI")])

(define_insn "*strmovhi_rex_1"
  [(set (mem:HI (match_operand:DI 2 "register_operand" "0"))
(mem:HI (match_operand:DI 3 "register_operand" "1")))
   (set (match_operand:DI 0 "register_operand" "=D")
```

```
(plus:DI (match_dup 2)
 (const_int 2)))
   (set (match_operand:DI 1 "register_operand" "=S")
(plus:DI (match_dup 3)
 (const_int 2)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "movsw"
  [(set_attr "type" "str")
   (set_attr "memory" "both")
   (set_attr "mode" "HI")])

(define_insn "*strmovqi_1"
  [(set (mem:QI (match_operand:SI 2 "register_operand" "0"))
(mem:QI (match_operand:SI 3 "register_operand" "1")))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 2)
 (const_int 1)))
   (set (match_operand:SI 1 "register_operand" "=S")
(plus:SI (match_dup 3)
 (const_int 1)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "movsb"
  [(set_attr "type" "str")
   (set_attr "memory" "both")
   (set_attr "mode" "QI")])

(define_insn "*strmovqi_rex_1"
  [(set (mem:QI (match_operand:DI 2 "register_operand" "0"))
(mem:QI (match_operand:DI 3 "register_operand" "1")))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 2)
 (const_int 1)))
   (set (match_operand:DI 1 "register_operand" "=S")
(plus:DI (match_dup 3)
 (const_int 1)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "movsb"
  [(set_attr "type" "str")
   (set_attr "memory" "both")
   (set_attr "mode" "QI")])

(define_expand "rep_mov"
  [(parallel [(set (match_operand 4 "register_operand" "") (const_int 0))
      (set (match_operand 0 "register_operand" "")
   (match_operand 5 "" ""))
      (set (match_operand 2 "register_operand" "")
   (match_operand 6 "" ""))
```

```
      (set (match_operand 1 "memory_operand" "")
   (match_operand 3 "memory_operand" ""))
      (use (match_dup 4))
      (use (reg:SI DIRFLAG_REG))])]
  ""
  "")

(define_insn "*rep_movdi_rex64"
  [(set (match_operand:DI 2 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (ashift:DI (match_operand:DI 5 "register_operand" "2")
    (const_int 3))
 (match_operand:DI 3 "register_operand" "0")))
   (set (match_operand:DI 1 "register_operand" "=S")
        (plus:DI (ashift:DI (match_dup 5) (const_int 3))
 (match_operand:DI 4 "register_operand" "1")))
   (set (mem:BLK (match_dup 3))
(mem:BLK (match_dup 4)))
   (use (match_dup 5))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;movsq|rep movsq}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "both")
   (set_attr "mode" "DI")])

(define_insn "*rep_movsi"
  [(set (match_operand:SI 2 "register_operand" "=c") (const_int 0))
   (set (match_operand:SI 0 "register_operand" "=D")
        (plus:SI (ashift:SI (match_operand:SI 5 "register_operand" "2")
    (const_int 2))
 (match_operand:SI 3 "register_operand" "0")))
   (set (match_operand:SI 1 "register_operand" "=S")
        (plus:SI (ashift:SI (match_dup 5) (const_int 2))
 (match_operand:SI 4 "register_operand" "1")))
   (set (mem:BLK (match_dup 3))
(mem:BLK (match_dup 4)))
   (use (match_dup 5))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT"
  "{rep\;movsl|rep movsd}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "both")
   (set_attr "mode" "SI")])

(define_insn "*rep_movsi_rex64"
  [(set (match_operand:DI 2 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
```

```
        (plus:DI (ashift:DI (match_operand:DI 5 "register_operand" "2")
    (const_int 2))
 (match_operand:DI 3 "register_operand" "0")))
   (set (match_operand:DI 1 "register_operand" "=S")
        (plus:DI (ashift:DI (match_dup 5) (const_int 2))
 (match_operand:DI 4 "register_operand" "1")))
   (set (mem:BLK (match_dup 3))
(mem:BLK (match_dup 4)))
   (use (match_dup 5))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;movsl|rep movsd}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "both")
   (set_attr "mode" "SI")])

(define_insn "*rep_movqi"
  [(set (match_operand:SI 2 "register_operand" "=c") (const_int 0))
   (set (match_operand:SI 0 "register_operand" "=D")
        (plus:SI (match_operand:SI 3 "register_operand" "0")
 (match_operand:SI 5 "register_operand" "2")))
   (set (match_operand:SI 1 "register_operand" "=S")
        (plus:SI (match_operand:SI 4 "register_operand" "1") (match_dup 5)))
   (set (mem:BLK (match_dup 3))
(mem:BLK (match_dup 4)))
   (use (match_dup 5))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT"
  "{rep\;movsb|rep movsb}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "both")
   (set_attr "mode" "SI")])

(define_insn "*rep_movqi_rex64"
  [(set (match_operand:DI 2 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (match_operand:DI 3 "register_operand" "0")
 (match_operand:DI 5 "register_operand" "2")))
   (set (match_operand:DI 1 "register_operand" "=S")
        (plus:DI (match_operand:DI 4 "register_operand" "1") (match_dup 5)))
   (set (mem:BLK (match_dup 3))
(mem:BLK (match_dup 4)))
   (use (match_dup 5))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;movsb|rep movsb}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
```

```
   (set_attr "memory" "both")
   (set_attr "mode" "SI")])

(define_expand "setmemsi"
   [(use (match_operand:BLK 0 "memory_operand" ""))
    (use (match_operand:SI 1 "nonmemory_operand" ""))
    (use (match_operand 2 "const_int_operand" ""))
    (use (match_operand 3 "const_int_operand" ""))]
  ""
{
 /* If value to set is not zero, use the library routine.  */
 if (operands[2] != const0_rtx)
   FAIL;

 if (ix86_expand_clrmem (operands[0], operands[1], operands[3]))
   DONE;
 else
   FAIL;
})

(define_expand "setmemdi"
   [(use (match_operand:BLK 0 "memory_operand" ""))
    (use (match_operand:DI 1 "nonmemory_operand" ""))
    (use (match_operand 2 "const_int_operand" ""))
    (use (match_operand 3 "const_int_operand" ""))]
  "TARGET_64BIT"
{
 /* If value to set is not zero, use the library routine.  */
 if (operands[2] != const0_rtx)
   FAIL;

 if (ix86_expand_clrmem (operands[0], operands[1], operands[3]))
   DONE;
 else
   FAIL;
})

;; Most CPUs don't like single string operations
;; Handle this case here to simplify previous expander.

(define_expand "strset"
  [(set (match_operand 1 "memory_operand" "")
(match_operand 2 "register_operand" ""))
   (parallel [(set (match_operand 0 "register_operand" "")
   (match_dup 3))
      (clobber (reg:CC FLAGS_REG))])]
  ""
{
  if (GET_MODE (operands[1]) != GET_MODE (operands[2]))
    operands[1] = adjust_address_nv (operands[1], GET_MODE (operands[2]), 0);
```

```
  /* If .md ever supports :P for Pmode, this can be directly
     in the pattern above.  */
  operands[3] = gen_rtx_PLUS (Pmode, operands[0],
      GEN_INT (GET_MODE_SIZE (GET_MODE
      (operands[2]))));
  if (TARGET_SINGLE_STRINGOP || optimize_size)
    {
      emit_insn (gen_strset_singleop (operands[0], operands[1], operands[2],
      operands[3]));
      DONE;
    }
})

(define_expand "strset_singleop"
  [(parallel [(set (match_operand 1 "memory_operand" "")
   (match_operand 2 "register_operand" ""))
      (set (match_operand 0 "register_operand" "")
   (match_operand 3 "" ""))
      (use (reg:SI DIRFLAG_REG))])]
  "TARGET_SINGLE_STRINGOP || optimize_size"
  "")

(define_insn "*strsetdi_rex_1"
  [(set (mem:DI (match_operand:DI 1 "register_operand" "0"))
(match_operand:DI 2 "register_operand" "a"))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 1)
 (const_int 8)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "stosq"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "DI")])

(define_insn "*strsetsi_1"
  [(set (mem:SI (match_operand:SI 1 "register_operand" "0"))
(match_operand:SI 2 "register_operand" "a"))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 1)
 (const_int 4)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "{stosl|stosd}"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "SI")])

(define_insn "*strsetsi_rex_1"
```

```
  [(set (mem:SI (match_operand:DI 1 "register_operand" "0"))
(match_operand:SI 2 "register_operand" "a"))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 1)
 (const_int 4)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "{stosl|stosd}"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "SI")])

(define_insn "*strsethi_1"
  [(set (mem:HI (match_operand:SI 1 "register_operand" "0"))
(match_operand:HI 2 "register_operand" "a"))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 1)
 (const_int 2)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "stosw"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "HI")])

(define_insn "*strsethi_rex_1"
  [(set (mem:HI (match_operand:DI 1 "register_operand" "0"))
(match_operand:HI 2 "register_operand" "a"))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 1)
 (const_int 2)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "stosw"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "HI")])

(define_insn "*strsetqi_1"
  [(set (mem:QI (match_operand:SI 1 "register_operand" "0"))
(match_operand:QI 2 "register_operand" "a"))
   (set (match_operand:SI 0 "register_operand" "=D")
(plus:SI (match_dup 1)
 (const_int 1)))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "stosb"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "QI")])
```

```
(define_insn "*strsetqi_rex_1"
  [(set (mem:QI (match_operand:DI 1 "register_operand" "0"))
(match_operand:QI 2 "register_operand" "a"))
   (set (match_operand:DI 0 "register_operand" "=D")
(plus:DI (match_dup 1)
 (const_int 1)))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT && (TARGET_SINGLE_STRINGOP || optimize_size)"
  "stosb"
  [(set_attr "type" "str")
   (set_attr "memory" "store")
   (set_attr "mode" "QI")])

(define_expand "rep_stos"
  [(parallel [(set (match_operand 1 "register_operand" "") (const_int 0))
      (set (match_operand 0 "register_operand" "")
   (match_operand 4 "" ""))
      (set (match_operand 2 "memory_operand" "") (const_int 0))
      (use (match_operand 3 "register_operand" ""))
      (use (match_dup 1))
      (use (reg:SI DIRFLAG_REG))])]
  ""
  "")

(define_insn "*rep_stosdi_rex64"
  [(set (match_operand:DI 1 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (ashift:DI (match_operand:DI 4 "register_operand" "1")
    (const_int 3))
 (match_operand:DI 3 "register_operand" "0")))
   (set (mem:BLK (match_dup 3))
(const_int 0))
   (use (match_operand:DI 2 "register_operand" "a"))
   (use (match_dup 4))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;stosq|rep stosq}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "store")
   (set_attr "mode" "DI")])

(define_insn "*rep_stossi"
  [(set (match_operand:SI 1 "register_operand" "=c") (const_int 0))
   (set (match_operand:SI 0 "register_operand" "=D")
        (plus:SI (ashift:SI (match_operand:SI 4 "register_operand" "1")
    (const_int 2))
 (match_operand:SI 3 "register_operand" "0")))
   (set (mem:BLK (match_dup 3))
```

```
(const_int 0))
   (use (match_operand:SI 2 "register_operand" "a"))
   (use (match_dup 4))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT"
  "{rep\;stosl|rep stosd}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "store")
   (set_attr "mode" "SI")])

(define_insn "*rep_stossi_rex64"
  [(set (match_operand:DI 1 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (ashift:DI (match_operand:DI 4 "register_operand" "1")
    (const_int 2))
 (match_operand:DI 3 "register_operand" "0")))
   (set (mem:BLK (match_dup 3))
(const_int 0))
   (use (match_operand:SI 2 "register_operand" "a"))
   (use (match_dup 4))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;stosl|rep stosd}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "store")
   (set_attr "mode" "SI")])

(define_insn "*rep_stosqi"
  [(set (match_operand:SI 1 "register_operand" "=c") (const_int 0))
   (set (match_operand:SI 0 "register_operand" "=D")
        (plus:SI (match_operand:SI 3 "register_operand" "0")
 (match_operand:SI 4 "register_operand" "1")))
   (set (mem:BLK (match_dup 3))
(const_int 0))
   (use (match_operand:QI 2 "register_operand" "a"))
   (use (match_dup 4))
   (use (reg:SI DIRFLAG_REG))]
  "!TARGET_64BIT"
  "{rep\;stosb|rep stosb}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "store")
   (set_attr "mode" "QI")])

(define_insn "*rep_stosqi_rex64"
  [(set (match_operand:DI 1 "register_operand" "=c") (const_int 0))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (match_operand:DI 3 "register_operand" "0")
```

```
 (match_operand:DI 4 "register_operand" "1")))
   (set (mem:BLK (match_dup 3))
(const_int 0))
   (use (match_operand:QI 2 "register_operand" "a"))
   (use (match_dup 4))
   (use (reg:SI DIRFLAG_REG))]
  "TARGET_64BIT"
  "{rep\;stosb|rep stosb}"
  [(set_attr "type" "str")
   (set_attr "prefix_rep" "1")
   (set_attr "memory" "store")
   (set_attr "mode" "QI")])

(define_expand "cmpstrnsi"
  [(set (match_operand:SI 0 "register_operand" "")
(compare:SI (match_operand:BLK 1 "general_operand" "")
    (match_operand:BLK 2 "general_operand" "")))
   (use (match_operand 3 "general_operand" ""))
   (use (match_operand 4 "immediate_operand" ""))]
  "! optimize_size || TARGET_INLINE_ALL_STRINGOPS"
{
  rtx addr1, addr2, out, outlow, count, countreg, align;

  /* Can't use this if the user has appropriated esi or edi.  */
  if (global_regs[4] || global_regs[5])
    FAIL;

  out = operands[0];
  if (GET_CODE (out) != REG)
    out = gen_reg_rtx (SImode);

  addr1 = copy_to_mode_reg (Pmode, XEXP (operands[1], 0));
  addr2 = copy_to_mode_reg (Pmode, XEXP (operands[2], 0));
  if (addr1 != XEXP (operands[1], 0))
    operands[1] = replace_equiv_address_nv (operands[1], addr1);
  if (addr2 != XEXP (operands[2], 0))
    operands[2] = replace_equiv_address_nv (operands[2], addr2);

  count = operands[3];
  countreg = ix86_zero_extend_to_Pmode (count);

  /* %%% Iff we are testing strict equality, we can use known alignment
     to good advantage.  This may be possible with combine, particularly
     once cc0 is dead.  */
  align = operands[4];

  emit_insn (gen_cld ());
  if (GET_CODE (count) == CONST_INT)
    {
      if (INTVAL (count) == 0)
```

```
{
  emit_move_insn (operands[0], const0_rtx);
  DONE;
}
      emit_insn (gen_cmpstrnqi_nz_1 (addr1, addr2, countreg, align,
     operands[1], operands[2]));
    }
  else
    {
      if (TARGET_64BIT)
emit_insn (gen_cmpdi_1_rex64 (countreg, countreg));
      else
emit_insn (gen_cmpsi_1 (countreg, countreg));
      emit_insn (gen_cmpstrnqi_1 (addr1, addr2, countreg, align,
  operands[1], operands[2]));
    }

  outlow = gen_lowpart (QImode, out);
  emit_insn (gen_cmpintqi (outlow));
  emit_move_insn (out, gen_rtx_SIGN_EXTEND (SImode, outlow));

  if (operands[0] != out)
    emit_move_insn (operands[0], out);

  DONE;
})

;; Produce a tri-state integer (-1, 0, 1) from condition codes.

(define_expand "cmpintqi"
  [(set (match_dup 1)
(gtu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (set (match_dup 2)
(ltu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (parallel [(set (match_operand:QI 0 "register_operand" "")
   (minus:QI (match_dup 1)
     (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "operands[1] = gen_reg_rtx (QImode);
   operands[2] = gen_reg_rtx (QImode);")

;; memcmp recognizers.  The `cmpsb' opcode does nothing if the count is
;; zero.  Emit extra code to make sure that a zero-length compare is EQ.

(define_expand "cmpstrnqi_nz_1"
  [(parallel [(set (reg:CC FLAGS_REG)
   (compare:CC (match_operand 4 "memory_operand" "")
       (match_operand 5 "memory_operand" "")))
      (use (match_operand 2 "register_operand" ""))
```

```
      (use (match_operand:SI 3 "immediate_operand" ""))
      (use (reg:SI DIRFLAG_REG))
      (clobber (match_operand 0 "register_operand" ""))
      (clobber (match_operand 1 "register_operand" ""))
      (clobber (match_dup 2))])]
  ""
  "")

(define_insn "*cmpstrnqi_nz_1"
  [(set (reg:CC FLAGS_REG)
(compare:CC (mem:BLK (match_operand:SI 4 "register_operand" "0"))
    (mem:BLK (match_operand:SI 5 "register_operand" "1"))))
   (use (match_operand:SI 6 "register_operand" "2"))
   (use (match_operand:SI 3 "immediate_operand" "i"))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:SI 0 "register_operand" "=S"))
   (clobber (match_operand:SI 1 "register_operand" "=D"))
   (clobber (match_operand:SI 2 "register_operand" "=c"))]
  "!TARGET_64BIT"
  "repz{\;| }cmpsb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

(define_insn "*cmpstrnqi_nz_rex_1"
  [(set (reg:CC FLAGS_REG)
(compare:CC (mem:BLK (match_operand:DI 4 "register_operand" "0"))
    (mem:BLK (match_operand:DI 5 "register_operand" "1"))))
   (use (match_operand:DI 6 "register_operand" "2"))
   (use (match_operand:SI 3 "immediate_operand" "i"))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:DI 0 "register_operand" "=S"))
   (clobber (match_operand:DI 1 "register_operand" "=D"))
   (clobber (match_operand:DI 2 "register_operand" "=c"))]
  "TARGET_64BIT"
  "repz{\;| }cmpsb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

;; The same, but the count is not known to not be zero.

(define_expand "cmpstrnqi_1"
  [(parallel [(set (reg:CC FLAGS_REG)
(if_then_else:CC (ne (match_operand 2 "register_operand" "")
     (const_int 0))
  (compare:CC (match_operand 4 "memory_operand" "")
      (match_operand 5 "memory_operand" ""))
  (const_int 0)))
      (use (match_operand:SI 3 "immediate_operand" ""))
```

```
      (use (reg:CC FLAGS_REG))
      (use (reg:SI DIRFLAG_REG))
      (clobber (match_operand 0 "register_operand" ""))
      (clobber (match_operand 1 "register_operand" ""))
      (clobber (match_dup 2))])]
  ""
  "")

(define_insn "*cmpstrnqi_1"
  [(set (reg:CC FLAGS_REG)
(if_then_else:CC (ne (match_operand:SI 6 "register_operand" "2")
     (const_int 0))
  (compare:CC (mem:BLK (match_operand:SI 4 "register_operand" "0"))
      (mem:BLK (match_operand:SI 5 "register_operand" "1")))
  (const_int 0)))
   (use (match_operand:SI 3 "immediate_operand" "i"))
   (use (reg:CC FLAGS_REG))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:SI 0 "register_operand" "=S"))
   (clobber (match_operand:SI 1 "register_operand" "=D"))
   (clobber (match_operand:SI 2 "register_operand" "=c"))]
  "!TARGET_64BIT"
  "repz{\;| }cmpsb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

(define_insn "*cmpstrnqi_rex_1"
  [(set (reg:CC FLAGS_REG)
(if_then_else:CC (ne (match_operand:DI 6 "register_operand" "2")
     (const_int 0))
  (compare:CC (mem:BLK (match_operand:DI 4 "register_operand" "0"))
      (mem:BLK (match_operand:DI 5 "register_operand" "1")))
  (const_int 0)))
   (use (match_operand:SI 3 "immediate_operand" "i"))
   (use (reg:CC FLAGS_REG))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:DI 0 "register_operand" "=S"))
   (clobber (match_operand:DI 1 "register_operand" "=D"))
   (clobber (match_operand:DI 2 "register_operand" "=c"))]
  "TARGET_64BIT"
  "repz{\;| }cmpsb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

(define_expand "strlensi"
  [(set (match_operand:SI 0 "register_operand" "")
(unspec:SI [(match_operand:BLK 1 "general_operand" "")
    (match_operand:QI 2 "immediate_operand" "")
```

```
    (match_operand 3 "immediate_operand" "")] UNSPEC_SCAS))]
  ""
{
 if (ix86_expand_strlen (operands[0], operands[1], operands[2], operands[3]))
   DONE;
 else
   FAIL;
})

(define_expand "strlendi"
  [(set (match_operand:DI 0 "register_operand" "")
(unspec:DI [(match_operand:BLK 1 "general_operand" "")
    (match_operand:QI 2 "immediate_operand" "")
    (match_operand 3 "immediate_operand" "")] UNSPEC_SCAS))]
  ""
{
 if (ix86_expand_strlen (operands[0], operands[1], operands[2], operands[3]))
   DONE;
 else
   FAIL;
})

(define_expand "strlenqi_1"
  [(parallel [(set (match_operand 0 "register_operand" "") (match_operand 2 "" ""))
      (use (reg:SI DIRFLAG_REG))
      (clobber (match_operand 1 "register_operand" ""))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "*strlenqi_1"
  [(set (match_operand:SI 0 "register_operand" "=&c")
(unspec:SI [(mem:BLK (match_operand:SI 5 "register_operand" "1"))
    (match_operand:QI 2 "register_operand" "a")
    (match_operand:SI 3 "immediate_operand" "i")
    (match_operand:SI 4 "register_operand" "0")] UNSPEC_SCAS))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:SI 1 "register_operand" "=D"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT"
  "repnz{\;| }scasb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

(define_insn "*strlenqi_rex_1"
  [(set (match_operand:DI 0 "register_operand" "=&c")
(unspec:DI [(mem:BLK (match_operand:DI 5 "register_operand" "1"))
    (match_operand:QI 2 "register_operand" "a")
    (match_operand:DI 3 "immediate_operand" "i")
```

```
    (match_operand:DI 4 "register_operand" "0")] UNSPEC_SCAS))
   (use (reg:SI DIRFLAG_REG))
   (clobber (match_operand:DI 1 "register_operand" "=D"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "repnz{\;| }scasb"
  [(set_attr "type" "str")
   (set_attr "mode" "QI")
   (set_attr "prefix_rep" "1")])

;; Peephole optimizations to clean up after cmpstrn*.  This should be
;; handled in combine, but it is not currently up to the task.
;; When used for their truth value, the cmpstrn* expanders generate
;; code like this:
;;
;;   repz cmpsb
;;   seta  %al
;;   setb  %dl
;;   cmpb  %al, %dl
;;   jcc label
;;
;; The intermediate three instructions are unnecessary.

;; This one handles cmpstrn*_nz_1...
(define_peephole2
  [(parallel[
     (set (reg:CC FLAGS_REG)
  (compare:CC (mem:BLK (match_operand 4 "register_operand" ""))
      (mem:BLK (match_operand 5 "register_operand" ""))))
     (use (match_operand 6 "register_operand" ""))
     (use (match_operand:SI 3 "immediate_operand" ""))
     (use (reg:SI DIRFLAG_REG))
     (clobber (match_operand 0 "register_operand" ""))
     (clobber (match_operand 1 "register_operand" ""))
     (clobber (match_operand 2 "register_operand" ""))])
   (set (match_operand:QI 7 "register_operand" "")
(gtu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (set (match_operand:QI 8 "register_operand" "")
(ltu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (set (reg FLAGS_REG)
(compare (match_dup 7) (match_dup 8)))
  ]
  "peep2_reg_dead_p (4, operands[7]) && peep2_reg_dead_p (4, operands[8])"
  [(parallel[
     (set (reg:CC FLAGS_REG)
  (compare:CC (mem:BLK (match_dup 4))
      (mem:BLK (match_dup 5))))
     (use (match_dup 6))
     (use (match_dup 3))
     (use (reg:SI DIRFLAG_REG))
```

```
     (clobber (match_dup 0))
     (clobber (match_dup 1))
     (clobber (match_dup 2))])]
  "")

;; ...and this one handles cmpstrn*_1.
(define_peephole2
  [(parallel[
     (set (reg:CC FLAGS_REG)
  (if_then_else:CC (ne (match_operand 6 "register_operand" "")
       (const_int 0))
    (compare:CC (mem:BLK (match_operand 4 "register_operand" ""))
        (mem:BLK (match_operand 5 "register_operand" "")))
    (const_int 0)))
     (use (match_operand:SI 3 "immediate_operand" ""))
     (use (reg:CC FLAGS_REG))
     (use (reg:SI DIRFLAG_REG))
     (clobber (match_operand 0 "register_operand" ""))
     (clobber (match_operand 1 "register_operand" ""))
     (clobber (match_operand 2 "register_operand" ""))])
   (set (match_operand:QI 7 "register_operand" "")
(gtu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (set (match_operand:QI 8 "register_operand" "")
(ltu:QI (reg:CC FLAGS_REG) (const_int 0)))
   (set (reg FLAGS_REG)
(compare (match_dup 7) (match_dup 8)))
  ]
  "peep2_reg_dead_p (4, operands[7]) && peep2_reg_dead_p (4, operands[8])"
  [(parallel[
     (set (reg:CC FLAGS_REG)
  (if_then_else:CC (ne (match_dup 6)
       (const_int 0))
    (compare:CC (mem:BLK (match_dup 4))
(mem:BLK (match_dup 5)))
    (const_int 0)))
     (use (match_dup 3))
     (use (reg:CC FLAGS_REG))
     (use (reg:SI DIRFLAG_REG))
     (clobber (match_dup 0))
     (clobber (match_dup 1))
     (clobber (match_dup 2))])]
  "")



;; Conditional move instructions.

(define_expand "movdicc"
  [(set (match_operand:DI 0 "register_operand" "")
(if_then_else:DI (match_operand 1 "comparison_operator" "")
```

```
 (match_operand:DI 2 "general_operand" "")
 (match_operand:DI 3 "general_operand" "")))]
  "TARGET_64BIT"
  "if (!ix86_expand_int_movcc (operands)) FAIL; DONE;")

(define_insn "x86_movdicc_0_m1_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r")
(if_then_else:DI (match_operand 1 "ix86_carry_flag_operator" "")
  (const_int -1)
  (const_int 0)))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "sbb{q}\t%0, %0"
  ; Since we don't have the proper number of operands for an alu insn,
  ; fill in all the blanks.
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "memory" "none")
   (set_attr "imm_disp" "false")
   (set_attr "mode" "DI")
   (set_attr "length_immediate" "0")])

(define_insn "*movdicc_c_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(if_then_else:DI (match_operator 1 "ix86_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:DI 2 "nonimmediate_operand" "rm,0")
      (match_operand:DI 3 "nonimmediate_operand" "0,rm")))]
  "TARGET_64BIT && TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   cmov%O2%C1\t{%2, %0|%0, %2}
   cmov%O2%c1\t{%3, %0|%0, %3}"
  [(set_attr "type" "icmov")
   (set_attr "mode" "DI")])

(define_expand "movsicc"
  [(set (match_operand:SI 0 "register_operand" "")
(if_then_else:SI (match_operand 1 "comparison_operator" "")
 (match_operand:SI 2 "general_operand" "")
 (match_operand:SI 3 "general_operand" "")))]
  ""
  "if (!ix86_expand_int_movcc (operands)) FAIL; DONE;")

;; Data flow gets confused by our desire for `sbbl reg,reg', and clearing
;; the register first winds up with `sbbl $0,reg', which is also weird.
;; So just document what we're doing explicitly.

(define_insn "x86_movsicc_0_m1"
  [(set (match_operand:SI 0 "register_operand" "=r")
```

```
(if_then_else:SI (match_operand 1 "ix86_carry_flag_operator" "")
  (const_int -1)
  (const_int 0)))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "sbb{l}\t%0, %0"
  ; Since we don't have the proper number of operands for an alu insn,
  ; fill in all the blanks.
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "memory" "none")
   (set_attr "imm_disp" "false")
   (set_attr "mode" "SI")
   (set_attr "length_immediate" "0")])

(define_insn "*movsicc_noc"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(if_then_else:SI (match_operator 1 "ix86_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:SI 2 "nonimmediate_operand" "rm,0")
      (match_operand:SI 3 "nonimmediate_operand" "0,rm")))]
  "TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   cmov%O2%C1\t{%2, %0|%0, %2}
   cmov%O2%c1\t{%3, %0|%0, %3}"
  [(set_attr "type" "icmov")
   (set_attr "mode" "SI")])

(define_expand "movhicc"
  [(set (match_operand:HI 0 "register_operand" "")
(if_then_else:HI (match_operand 1 "comparison_operator" "")
 (match_operand:HI 2 "general_operand" "")
 (match_operand:HI 3 "general_operand" "")))]
  "TARGET_HIMODE_MATH"
  "if (!ix86_expand_int_movcc (operands)) FAIL; DONE;")

(define_insn "*movhicc_noc"
  [(set (match_operand:HI 0 "register_operand" "=r,r")
(if_then_else:HI (match_operator 1 "ix86_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:HI 2 "nonimmediate_operand" "rm,0")
      (match_operand:HI 3 "nonimmediate_operand" "0,rm")))]
  "TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   cmov%O2%C1\t{%2, %0|%0, %2}
   cmov%O2%c1\t{%3, %0|%0, %3}"
  [(set_attr "type" "icmov")
   (set_attr "mode" "HI")])
```

```
(define_expand "movqicc"
  [(set (match_operand:QI 0 "register_operand" "")
(if_then_else:QI (match_operand 1 "comparison_operator" "")
 (match_operand:QI 2 "general_operand" "")
 (match_operand:QI 3 "general_operand" "")))]
  "TARGET_QIMODE_MATH"
  "if (!ix86_expand_int_movcc (operands)) FAIL; DONE;")

(define_insn_and_split "*movqicc_noc"
  [(set (match_operand:QI 0 "register_operand" "=r,r")
(if_then_else:QI (match_operator 1 "ix86_comparison_operator"
[(match_operand 4 "flags_reg_operand" "")
 (const_int 0)])
      (match_operand:QI 2 "register_operand" "r,0")
      (match_operand:QI 3 "register_operand" "0,r")))]
  "TARGET_CMOVE && !TARGET_PARTIAL_REG_STALL"
  "#"
  "&& reload_completed"
  [(set (match_dup 0)
(if_then_else:SI (match_op_dup 1 [(match_dup 4) (const_int 0)])
      (match_dup 2)
      (match_dup 3)))]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[2] = gen_lowpart (SImode, operands[2]);
   operands[3] = gen_lowpart (SImode, operands[3]);"
  [(set_attr "type" "icmov")
   (set_attr "mode" "SI")])

(define_expand "movsfcc"
  [(set (match_operand:SF 0 "register_operand" "")
(if_then_else:SF (match_operand 1 "comparison_operator" "")
 (match_operand:SF 2 "register_operand" "")
 (match_operand:SF 3 "register_operand" "")))]
  "(TARGET_80387 && TARGET_CMOVE) || TARGET_SSE_MATH"
  "if (! ix86_expand_fp_movcc (operands)) FAIL; DONE;")

(define_insn "*movsfcc_1_387"
  [(set (match_operand:SF 0 "register_operand" "=f#r,f#r,r#f,r#f")
(if_then_else:SF (match_operator 1 "fcmov_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:SF 2 "nonimmediate_operand" "f#r,0,rm#f,0")
      (match_operand:SF 3 "nonimmediate_operand" "0,f#r,0,rm#f")))]
  "TARGET_80387 && TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   fcmov%F1\t{%2, %0|%0, %2}
   fcmov%f1\t{%3, %0|%0, %3}
   cmov%O2%C1\t{%2, %0|%0, %2}
   cmov%O2%c1\t{%3, %0|%0, %3}"
```

```
  [(set_attr "type" "fcmov,fcmov,icmov,icmov")
   (set_attr "mode" "SF,SF,SI,SI")])

(define_expand "movdfcc"
  [(set (match_operand:DF 0 "register_operand" "")
(if_then_else:DF (match_operand 1 "comparison_operator" "")
 (match_operand:DF 2 "register_operand" "")
 (match_operand:DF 3 "register_operand" "")))]
  "(TARGET_80387 && TARGET_CMOVE) || (TARGET_SSE2 && TARGET_SSE_MATH)"
  "if (! ix86_expand_fp_movcc (operands)) FAIL; DONE;")

(define_insn "*movdfcc_1"
  [(set (match_operand:DF 0 "register_operand" "=f#r,f#r,&r#f,&r#f")
(if_then_else:DF (match_operator 1 "fcmov_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:DF 2 "nonimmediate_operand" "f#r,0,rm#f,0")
      (match_operand:DF 3 "nonimmediate_operand" "0,f#r,0,rm#f")))]
  "!TARGET_64BIT && TARGET_80387 && TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   fcmov%F1\t{%2, %0|%0, %2}
   fcmov%f1\t{%3, %0|%0, %3}
   #
   #"
  [(set_attr "type" "fcmov,fcmov,multi,multi")
   (set_attr "mode" "DF")])

(define_insn "*movdfcc_1_rex64"
  [(set (match_operand:DF 0 "register_operand" "=f#r,f#r,r#f,r#f")
(if_then_else:DF (match_operator 1 "fcmov_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:DF 2 "nonimmediate_operand" "f#r,0#r,rm#f,0#f")
      (match_operand:DF 3 "nonimmediate_operand" "0#r,f#r,0#f,rm#f")))]
  "TARGET_64BIT && TARGET_80387 && TARGET_CMOVE
   && (GET_CODE (operands[2]) != MEM || GET_CODE (operands[3]) != MEM)"
  "@
   fcmov%F1\t{%2, %0|%0, %2}
   fcmov%f1\t{%3, %0|%0, %3}
   cmov%O2%C1\t{%2, %0|%0, %2}
   cmov%O2%c1\t{%3, %0|%0, %3}"
  [(set_attr "type" "fcmov,fcmov,icmov,icmov")
   (set_attr "mode" "DF")])

(define_split
  [(set (match_operand:DF 0 "register_and_not_any_fp_reg_operand" "")
(if_then_else:DF (match_operator 1 "fcmov_comparison_operator"
[(match_operand 4 "flags_reg_operand" "")
 (const_int 0)])
      (match_operand:DF 2 "nonimmediate_operand" "")
      (match_operand:DF 3 "nonimmediate_operand" "")))]
```

```
  "!TARGET_64BIT && reload_completed"
  [(set (match_dup 2)
(if_then_else:SI (match_op_dup 1 [(match_dup 4) (const_int 0)])
      (match_dup 5)
      (match_dup 7)))
   (set (match_dup 3)
(if_then_else:SI (match_op_dup 1 [(match_dup 4) (const_int 0)])
      (match_dup 6)
      (match_dup 8)))]
  "split_di (operands+2, 1, operands+5, operands+6);
   split_di (operands+3, 1, operands+7, operands+8);
   split_di (operands, 1, operands+2, operands+3);")

(define_expand "movxfcc"
  [(set (match_operand:XF 0 "register_operand" "")
(if_then_else:XF (match_operand 1 "comparison_operator" "")
 (match_operand:XF 2 "register_operand" "")
 (match_operand:XF 3 "register_operand" "")))]
  "TARGET_80387 && TARGET_CMOVE"
  "if (! ix86_expand_fp_movcc (operands)) FAIL; DONE;")

(define_insn "*movxfcc_1"
  [(set (match_operand:XF 0 "register_operand" "=f,f")
(if_then_else:XF (match_operator 1 "fcmov_comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand:XF 2 "register_operand" "f,0")
      (match_operand:XF 3 "register_operand" "0,f")))]
  "TARGET_80387 && TARGET_CMOVE"
  "@
   fcmov%F1\t{%2, %0|%0, %2}
   fcmov%f1\t{%3, %0|%0, %3}"
  [(set_attr "type" "fcmov")
   (set_attr "mode" "XF")])

;; These versions of the min/max patterns are intentionally ignorant of
;; their behavior wrt -0.0 and NaN (via the commutative operand mark).
;; Since both the tree-level MAX_EXPR and the rtl-level SMAX operator
;; are undefined in this condition, we're certain this is correct.

(define_insn "sminsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(smin:SF (match_operand:SF 1 "nonimmediate_operand" "%0")
 (match_operand:SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE_MATH"
  "minss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])

(define_insn "smaxsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
```

```
(smax:SF (match_operand:SF 1 "nonimmediate_operand" "%0")
 (match_operand:SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE_MATH"
  "maxss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])

(define_insn "smindf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(smin:DF (match_operand:DF 1 "nonimmediate_operand" "%0")
 (match_operand:DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "minsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_insn "smaxdf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(smax:DF (match_operand:DF 1 "nonimmediate_operand" "%0")
 (match_operand:DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "maxsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

;; These versions of the min/max patterns implement exactly the operations
;;   min = (op1 < op2 ? op1 : op2)
;;   max = (!(op1 < op2) ? op1 : op2)
;; Their operands are not commutative, and thus they may be used in the
;; presence of -0.0 and NaN.

(define_insn "*ieee_sminsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(unspec:SF [(match_operand:SF 1 "register_operand" "0")
    (match_operand:SF 2 "nonimmediate_operand" "xm")]
   UNSPEC_IEEE_MIN))]
  "TARGET_SSE_MATH"
  "minss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])

(define_insn "*ieee_smaxsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(unspec:SF [(match_operand:SF 1 "register_operand" "0")
    (match_operand:SF 2 "nonimmediate_operand" "xm")]
   UNSPEC_IEEE_MAX))]
  "TARGET_SSE_MATH"
  "maxss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])
```

```
(define_insn "*ieee_smindf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(unspec:DF [(match_operand:DF 1 "register_operand" "0")
    (match_operand:DF 2 "nonimmediate_operand" "xm")]
   UNSPEC_IEEE_MIN))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "minsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_insn "*ieee_smaxdf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(unspec:DF [(match_operand:DF 1 "register_operand" "0")
    (match_operand:DF 2 "nonimmediate_operand" "xm")]
   UNSPEC_IEEE_MAX))]
  "TARGET_SSE2 && TARGET_SSE_MATH"
  "maxsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

;; Conditional addition patterns
(define_expand "addqicc"
  [(match_operand:QI 0 "register_operand" "")
   (match_operand 1 "comparison_operator" "")
   (match_operand:QI 2 "register_operand" "")
   (match_operand:QI 3 "const_int_operand" "")]
  ""
  "if (!ix86_expand_int_addcc (operands)) FAIL; DONE;")

(define_expand "addhicc"
  [(match_operand:HI 0 "register_operand" "")
   (match_operand 1 "comparison_operator" "")
   (match_operand:HI 2 "register_operand" "")
   (match_operand:HI 3 "const_int_operand" "")]
  ""
  "if (!ix86_expand_int_addcc (operands)) FAIL; DONE;")

(define_expand "addsicc"
  [(match_operand:SI 0 "register_operand" "")
   (match_operand 1 "comparison_operator" "")
   (match_operand:SI 2 "register_operand" "")
   (match_operand:SI 3 "const_int_operand" "")]
  ""
  "if (!ix86_expand_int_addcc (operands)) FAIL; DONE;")

(define_expand "adddicc"
  [(match_operand:DI 0 "register_operand" "")
   (match_operand 1 "comparison_operator" "")
   (match_operand:DI 2 "register_operand" "")
```

```
   (match_operand:DI 3 "const_int_operand" "")]
  "TARGET_64BIT"
  "if (!ix86_expand_int_addcc (operands)) FAIL; DONE;")


;; Misc patterns (?)

;; This pattern exists to put a dependency on all ebp-based memory accesses.
;; Otherwise there will be nothing to keep
;;
;; [(set (reg ebp) (reg esp))]
;; [(set (reg esp) (plus (reg esp) (const_int -160000)))
;;  (clobber (eflags)]
;; [(set (mem (plus (reg ebp) (const_int -160000))) (const_int 0))]
;;
;; in proper program order.
(define_insn "pro_epilogue_adjust_stack_1"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(plus:SI (match_operand:SI 1 "register_operand" "0,r")
         (match_operand:SI 2 "immediate_operand" "i,i")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (mem:BLK (scratch)))]
  "!TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOV:
      return "mov{l}\t{%1, %0|%0, %1}";

    case TYPE_ALU:
      if (GET_CODE (operands[2]) == CONST_INT
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
          && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{l}\t{%2, %0|%0, %2}";
}
      return "add{l}\t{%2, %0|%0, %2}";

    case TYPE_LEA:
      operands[2] = SET_SRC (XVECEXP (PATTERN (insn), 0, 0));
      return "lea{l}\t{%a2, %0|%0, %a2}";

    default:
      gcc_unreachable ();
    }
}
  [(set (attr "type")
(cond [(eq_attr "alternative" "0")
```

```
 (const_string "alu")
       (match_operand:SI 2 "const0_operand" "")
 (const_string "imov")
      ]
      (const_string "lea")))
   (set_attr "mode" "SI")])

(define_insn "pro_epilogue_adjust_stack_rex64"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(plus:DI (match_operand:DI 1 "register_operand" "0,r")
 (match_operand:DI 2 "x86_64_immediate_operand" "e,e")))
   (clobber (reg:CC FLAGS_REG))
   (clobber (mem:BLK (scratch)))]
  "TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_IMOV:
      return "mov{q}\t{%1, %0|%0, %1}";

    case TYPE_ALU:
      if (GET_CODE (operands[2]) == CONST_INT
  /* Avoid overflows.  */
  && ((INTVAL (operands[2]) & ((((unsigned int) 1) << 31) - 1)))
          && (INTVAL (operands[2]) == 128
      || (INTVAL (operands[2]) < 0
          && INTVAL (operands[2]) != -128)))
{
  operands[2] = GEN_INT (-INTVAL (operands[2]));
  return "sub{q}\t{%2, %0|%0, %2}";
}
      return "add{q}\t{%2, %0|%0, %2}";

    case TYPE_LEA:
      operands[2] = SET_SRC (XVECEXP (PATTERN (insn), 0, 0));
      return "lea{q}\t{%a2, %0|%0, %a2}";

    default:
      gcc_unreachable ();
    }
}
  [(set (attr "type")
(cond [(eq_attr "alternative" "0")
 (const_string "alu")
       (match_operand:DI 2 "const0_operand" "")
 (const_string "imov")
      ]
      (const_string "lea")))
   (set_attr "mode" "DI")])
```

```
(define_insn "pro_epilogue_adjust_stack_rex64_2"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(plus:DI (match_operand:DI 1 "register_operand" "0,r")
 (match_operand:DI 3 "immediate_operand" "i,i")))
   (use (match_operand:DI 2 "register_operand" "r,r"))
   (clobber (reg:CC FLAGS_REG))
   (clobber (mem:BLK (scratch)))]
  "TARGET_64BIT"
{
  switch (get_attr_type (insn))
    {
    case TYPE_ALU:
      return "add{q}\t{%2, %0|%0, %2}";

    case TYPE_LEA:
      operands[2] = gen_rtx_PLUS (DImode, operands[1], operands[2]);
      return "lea{q}\t{%a2, %0|%0, %a2}";

    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "alu,lea")
   (set_attr "mode" "DI")])

(define_expand "allocate_stack_worker"
  [(match_operand:SI 0 "register_operand" "")]
  "TARGET_STACK_PROBE"
{
  if (reload_completed)
    {
      if (TARGET_64BIT)
emit_insn (gen_allocate_stack_worker_rex64_postreload (operands[0]));
      else
emit_insn (gen_allocate_stack_worker_postreload (operands[0]));
    }
  else
    {
      if (TARGET_64BIT)
emit_insn (gen_allocate_stack_worker_rex64 (operands[0]));
      else
emit_insn (gen_allocate_stack_worker_1 (operands[0]));
    }
  DONE;
})

(define_insn "allocate_stack_worker_1"
  [(unspec_volatile:SI [(match_operand:SI 0 "register_operand" "a")]
    UNSPECV_STACK_PROBE)
   (set (reg:SI SP_REG) (minus:SI (reg:SI SP_REG) (match_dup 0)))
```

```
   (clobber (match_scratch:SI 1 "=0"))
   (clobber (reg:CC FLAGS_REG))]
  "!TARGET_64BIT && TARGET_STACK_PROBE"
  "call\t__alloca"
  [(set_attr "type" "multi")
   (set_attr "length" "5")])

(define_expand "allocate_stack_worker_postreload"
  [(parallel [(unspec_volatile:SI [(match_operand:SI 0 "register_operand" "a")]
    UNSPECV_STACK_PROBE)
      (set (reg:SI SP_REG) (minus:SI (reg:SI SP_REG) (match_dup 0)))
      (clobber (match_dup 0))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_insn "allocate_stack_worker_rex64"
  [(unspec_volatile:DI [(match_operand:DI 0 "register_operand" "a")]
    UNSPECV_STACK_PROBE)
   (set (reg:DI SP_REG) (minus:DI (reg:DI SP_REG) (match_dup 0)))
   (clobber (match_scratch:DI 1 "=0"))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && TARGET_STACK_PROBE"
  "call\t__alloca"
  [(set_attr "type" "multi")
   (set_attr "length" "5")])

(define_expand "allocate_stack_worker_rex64_postreload"
  [(parallel [(unspec_volatile:DI [(match_operand:DI 0 "register_operand" "a")]
    UNSPECV_STACK_PROBE)
      (set (reg:DI SP_REG) (minus:DI (reg:DI SP_REG) (match_dup 0)))
      (clobber (match_dup 0))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  "")

(define_expand "allocate_stack"
  [(parallel [(set (match_operand:SI 0 "register_operand" "=r")
   (minus:SI (reg:SI SP_REG)
     (match_operand:SI 1 "general_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (parallel [(set (reg:SI SP_REG)
   (minus:SI (reg:SI SP_REG) (match_dup 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_STACK_PROBE"
{
#ifdef CHECK_STACK_LIMIT
  if (GET_CODE (operands[1]) == CONST_INT
      && INTVAL (operands[1]) < CHECK_STACK_LIMIT)
    emit_insn (gen_subsi3 (stack_pointer_rtx, stack_pointer_rtx,
```

```
   operands[1]));
  else
#endif
    emit_insn (gen_allocate_stack_worker (copy_to_mode_reg (SImode,
    operands[1])));

  emit_move_insn (operands[0], virtual_stack_dynamic_rtx);
  DONE;
})

(define_expand "builtin_setjmp_receiver"
  [(label_ref (match_operand 0 "" ""))]
  "!TARGET_64BIT && flag_pic"
{
  emit_insn (gen_set_got (pic_offset_table_rtx));
  DONE;
})

;; Avoid redundant prefixes by splitting HImode arithmetic to SImode.

(define_split
  [(set (match_operand 0 "register_operand" "")
(match_operator 3 "promotable_binary_operator"
   [(match_operand 1 "register_operand" "")
    (match_operand 2 "aligned_operand" "")]))
   (clobber (reg:CC FLAGS_REG))]
  "! TARGET_PARTIAL_REG_STALL && reload_completed
   && ((GET_MODE (operands[0]) == HImode
&& ((!optimize_size && !TARGET_FAST_PREFIX)
    || GET_CODE (operands[2]) != CONST_INT
    || CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K')))
       || (GET_MODE (operands[0]) == QImode
   && (TARGET_PROMOTE_QImode || optimize_size)))"
  [(parallel [(set (match_dup 0)
   (match_op_dup 3 [(match_dup 1) (match_dup 2)]))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);
   if (GET_CODE (operands[3]) != ASHIFT)
     operands[2] = gen_lowpart (SImode, operands[2]);
   PUT_MODE (operands[3], SImode);")

; Promote the QImode tests, as i386 has encoding of the AND
; instruction with 32-bit sign-extended immediate and thus the
; instruction size is unchanged, except in the %eax case for
; which it is increased by one byte, hence the ! optimize_size.
(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 2 "compare_operator"
  [(and (match_operand 3 "aligned_operand" "")
```

```
(match_operand 4 "const_int_operand" ""))
   (const_int 0)]))
   (set (match_operand 1 "register_operand" "")
(and (match_dup 3) (match_dup 4)))]
  "! TARGET_PARTIAL_REG_STALL && reload_completed
   /* Ensure that the operand will remain sign-extended immediate.  */
   && ix86_match_ccmode (insn, INTVAL (operands[4]) >= 0 ? CCNOmode : CCZmode)
   && ! optimize_size
   && ((GET_MODE (operands[1]) == HImode && ! TARGET_FAST_PREFIX)
       || (GET_MODE (operands[1]) == QImode && TARGET_PROMOTE_QImode))"
  [(parallel [(set (match_dup 0)
   (match_op_dup 2 [(and:SI (match_dup 3) (match_dup 4))
            (const_int 0)]))
      (set (match_dup 1)
   (and:SI (match_dup 3) (match_dup 4)))])]
{
  operands[4]
    = gen_int_mode (INTVAL (operands[4])
    & GET_MODE_MASK (GET_MODE (operands[1])), SImode);
  operands[1] = gen_lowpart (SImode, operands[1]);
  operands[3] = gen_lowpart (SImode, operands[3]);
})

; Don't promote the QImode tests, as i386 doesn't have encoding of
; the TEST instruction with 32-bit sign-extended immediate and thus
; the instruction size would at least double, which is not what we
; want even with ! optimize_size.
(define_split
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and (match_operand:HI 2 "aligned_operand" "")
(match_operand:HI 3 "const_int_operand" ""))
   (const_int 0)]))]
  "! TARGET_PARTIAL_REG_STALL && reload_completed
   /* Ensure that the operand will remain sign-extended immediate.  */
   && ix86_match_ccmode (insn, INTVAL (operands[3]) >= 0 ? CCNOmode : CCZmode)
   && ! TARGET_FAST_PREFIX
   && ! optimize_size"
  [(set (match_dup 0)
(match_op_dup 1 [(and:SI (match_dup 2) (match_dup 3))
         (const_int 0)]))]
{
  operands[3]
    = gen_int_mode (INTVAL (operands[3])
    & GET_MODE_MASK (GET_MODE (operands[2])), SImode);
  operands[2] = gen_lowpart (SImode, operands[2]);
})

(define_split
  [(set (match_operand 0 "register_operand" "")
```

```
(neg (match_operand 1 "register_operand" "")))
   (clobber (reg:CC FLAGS_REG))]
  "! TARGET_PARTIAL_REG_STALL && reload_completed
   && (GET_MODE (operands[0]) == HImode
       || (GET_MODE (operands[0]) == QImode
   && (TARGET_PROMOTE_QImode || optimize_size)))"
  [(parallel [(set (match_dup 0)
   (neg:SI (match_dup 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);")

(define_split
  [(set (match_operand 0 "register_operand" "")
(not (match_operand 1 "register_operand" "")))]
  "! TARGET_PARTIAL_REG_STALL && reload_completed
   && (GET_MODE (operands[0]) == HImode
       || (GET_MODE (operands[0]) == QImode
   && (TARGET_PROMOTE_QImode || optimize_size)))"
  [(set (match_dup 0)
(not:SI (match_dup 1)))]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[1] = gen_lowpart (SImode, operands[1]);")

(define_split
  [(set (match_operand 0 "register_operand" "")
(if_then_else (match_operator 1 "comparison_operator"
[(reg FLAGS_REG) (const_int 0)])
      (match_operand 2 "register_operand" "")
      (match_operand 3 "register_operand" "")))]
  "! TARGET_PARTIAL_REG_STALL && TARGET_CMOVE
   && (GET_MODE (operands[0]) == HImode
       || (GET_MODE (operands[0]) == QImode
   && (TARGET_PROMOTE_QImode || optimize_size)))"
  [(set (match_dup 0)
(if_then_else:SI (match_dup 1) (match_dup 2) (match_dup 3)))]
  "operands[0] = gen_lowpart (SImode, operands[0]);
   operands[2] = gen_lowpart (SImode, operands[2]);
   operands[3] = gen_lowpart (SImode, operands[3]);")


;; RTL Peephole optimizations, run before sched2.  These primarily look to
;; transform a complex memory operation into two memory to register operations.

;; Don't push memory operands
(define_peephole2
  [(set (match_operand:SI 0 "push_operand" "")
(match_operand:SI 1 "memory_operand" ""))
   (match_scratch:SI 2 "r")]
  "! optimize_size && ! TARGET_PUSH_MEMORY"
```

```
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(set (match_operand:DI 0 "push_operand" "")
(match_operand:DI 1 "memory_operand" ""))
   (match_scratch:DI 2 "r")]
  "! optimize_size && ! TARGET_PUSH_MEMORY"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

;; We need to handle SFmode only, because DFmode and XFmode is split to
;; SImode pushes.
(define_peephole2
  [(set (match_operand:SF 0 "push_operand" "")
(match_operand:SF 1 "memory_operand" ""))
   (match_scratch:SF 2 "r")]
  "! optimize_size && ! TARGET_PUSH_MEMORY"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(set (match_operand:HI 0 "push_operand" "")
(match_operand:HI 1 "memory_operand" ""))
   (match_scratch:HI 2 "r")]
  "! optimize_size && ! TARGET_PUSH_MEMORY"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(set (match_operand:QI 0 "push_operand" "")
(match_operand:QI 1 "memory_operand" ""))
   (match_scratch:QI 2 "q")]
  "! optimize_size && ! TARGET_PUSH_MEMORY"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

;; Don't move an immediate directly to memory when the instruction
;; gets too big.
(define_peephole2
  [(match_scratch:SI 1 "r")
   (set (match_operand:SI 0 "memory_operand" "")
        (const_int 0))]
  "! optimize_size
   && ! TARGET_USE_MOV0
```

```
   && TARGET_SPLIT_LONG_MOVES
   && get_attr_length (insn) >= ix86_cost->large_insn
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 1) (const_int 0))
      (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 0) (match_dup 1))]
  "")

(define_peephole2
  [(match_scratch:HI 1 "r")
   (set (match_operand:HI 0 "memory_operand" "")
        (const_int 0))]
  "! optimize_size
   && ! TARGET_USE_MOV0
   && TARGET_SPLIT_LONG_MOVES
   && get_attr_length (insn) >= ix86_cost->large_insn
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 2) (const_int 0))
      (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 0) (match_dup 1))]
  "operands[2] = gen_lowpart (SImode, operands[1]);")

(define_peephole2
  [(match_scratch:QI 1 "q")
   (set (match_operand:QI 0 "memory_operand" "")
        (const_int 0))]
  "! optimize_size
   && ! TARGET_USE_MOV0
   && TARGET_SPLIT_LONG_MOVES
   && get_attr_length (insn) >= ix86_cost->large_insn
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 2) (const_int 0))
      (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 0) (match_dup 1))]
  "operands[2] = gen_lowpart (SImode, operands[1]);")

(define_peephole2
  [(match_scratch:SI 2 "r")
   (set (match_operand:SI 0 "memory_operand" "")
        (match_operand:SI 1 "immediate_operand" ""))]
  "! optimize_size
   && get_attr_length (insn) >= ix86_cost->large_insn
   && TARGET_SPLIT_LONG_MOVES"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(match_scratch:HI 2 "r")
   (set (match_operand:HI 0 "memory_operand" "")
```

```
       (match_operand:HI 1 "immediate_operand" ""))]
  "! optimize_size && get_attr_length (insn) >= ix86_cost->large_insn
  && TARGET_SPLIT_LONG_MOVES"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(match_scratch:QI 2 "q")
   (set (match_operand:QI 0 "memory_operand" "")
        (match_operand:QI 1 "immediate_operand" ""))]
  "! optimize_size && get_attr_length (insn) >= ix86_cost->large_insn
  && TARGET_SPLIT_LONG_MOVES"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")

;; Don't compare memory with zero, load and use a test instead.
(define_peephole2
  [(set (match_operand 0 "flags_reg_operand" "")
  (match_operator 1 "compare_operator"
  [(match_operand:SI 2 "memory_operand" "")
   (const_int 0)]))
   (match_scratch:SI 3 "r")]
  "ix86_match_ccmode (insn, CCNOmode) && ! optimize_size"
  [(set (match_dup 3) (match_dup 2))
   (set (match_dup 0) (match_op_dup 1 [(match_dup 3) (const_int 0)]))]
  "")

;; NOT is not pairable on Pentium, while XOR is, but one byte longer.
;; Don't split NOTs with a displacement operand, because resulting XOR
;; will not be pairable anyway.
;;
;; On AMD K6, NOT is vector decoded with memory operand that cannot be
;; represented using a modRM byte.  The XOR replacement is long decoded,
;; so this split helps here as well.
;;
;; Note: Can't do this as a regular split because we can't get proper
;; lifetime information then.

(define_peephole2
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
(not:SI (match_operand:SI 1 "nonimmediate_operand" "")))]
  "!optimize_size
   && peep2_regno_dead_p (0, FLAGS_REG)
   && ((TARGET_PENTIUM
        && (GET_CODE (operands[0]) != MEM
            || !memory_displacement_operand (operands[0], SImode)))
       || (TARGET_K6 && long_memory_operand (operands[0], SImode)))"
  [(parallel [(set (match_dup 0)
```

```
   (xor:SI (match_dup 1) (const_int -1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_peephole2
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
(not:HI (match_operand:HI 1 "nonimmediate_operand" "")))]
  "!optimize_size
   && peep2_regno_dead_p (0, FLAGS_REG)
   && ((TARGET_PENTIUM
        && (GET_CODE (operands[0]) != MEM
            || !memory_displacement_operand (operands[0], HImode)))
       || (TARGET_K6 && long_memory_operand (operands[0], HImode)))"
  [(parallel [(set (match_dup 0)
   (xor:HI (match_dup 1) (const_int -1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_peephole2
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
(not:QI (match_operand:QI 1 "nonimmediate_operand" "")))]
  "!optimize_size
   && peep2_regno_dead_p (0, FLAGS_REG)
   && ((TARGET_PENTIUM
        && (GET_CODE (operands[0]) != MEM
            || !memory_displacement_operand (operands[0], QImode)))
       || (TARGET_K6 && long_memory_operand (operands[0], QImode)))"
  [(parallel [(set (match_dup 0)
   (xor:QI (match_dup 1) (const_int -1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

;; Non pairable "test imm, reg" instructions can be translated to
;; "and imm, reg" if reg dies.  The "and" form is also shorter (one
;; byte opcode instead of two, have a short form for byte operands),
;; so do it for other CPUs as well.  Given that the value was dead,
;; this should not create any new dependencies.  Pass on the sub-word
;; versions if we're concerned about partial register stalls.

(define_peephole2
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and:SI (match_operand:SI 2 "register_operand" "")
   (match_operand:SI 3 "immediate_operand" ""))
   (const_int 0)]))]
  "ix86_match_ccmode (insn, CCNOmode)
   && (true_regnum (operands[2]) != 0
       || (GET_CODE (operands[3]) == CONST_INT
   && CONST_OK_FOR_LETTER_P (INTVAL (operands[3]), 'K')))
   && peep2_reg_dead_p (1, operands[2])"
```

```
  [(parallel
     [(set (match_dup 0)
   (match_op_dup 1 [(and:SI (match_dup 2) (match_dup 3))
            (const_int 0)]))
      (set (match_dup 2)
   (and:SI (match_dup 2) (match_dup 3)))])]
  "")

;; We don't need to handle HImode case, because it will be promoted to SImode
;; on ! TARGET_PARTIAL_REG_STALL

(define_peephole2
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and:QI (match_operand:QI 2 "register_operand" "")
   (match_operand:QI 3 "immediate_operand" ""))
   (const_int 0)]))]
  "! TARGET_PARTIAL_REG_STALL
   && ix86_match_ccmode (insn, CCNOmode)
   && true_regnum (operands[2]) != 0
   && peep2_reg_dead_p (1, operands[2])"
  [(parallel
     [(set (match_dup 0)
   (match_op_dup 1 [(and:QI (match_dup 2) (match_dup 3))
            (const_int 0)]))
      (set (match_dup 2)
   (and:QI (match_dup 2) (match_dup 3)))])]
  "")

(define_peephole2
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(and:SI
     (zero_extract:SI
       (match_operand 2 "ext_register_operand" "")
       (const_int 8)
       (const_int 8))
     (match_operand 3 "const_int_operand" ""))
   (const_int 0)]))]
  "! TARGET_PARTIAL_REG_STALL
   && ix86_match_ccmode (insn, CCNOmode)
   && true_regnum (operands[2]) != 0
   && peep2_reg_dead_p (1, operands[2])"
  [(parallel [(set (match_dup 0)
   (match_op_dup 1
     [(and:SI
(zero_extract:SI
  (match_dup 2)
  (const_int 8)
  (const_int 8))
```

```
(match_dup 3))
      (const_int 0)]))
      (set (zero_extract:SI (match_dup 2)
    (const_int 8)
    (const_int 8))
   (and:SI
     (zero_extract:SI
       (match_dup 2)
       (const_int 8)
       (const_int 8))
     (match_dup 3)))])]
  "")

;; Don't do logical operations with memory inputs.
(define_peephole2
  [(match_scratch:SI 2 "r")
   (parallel [(set (match_operand:SI 0 "register_operand" "")
                   (match_operator:SI 3 "arith_or_logical_operator"
                     [(match_dup 0)
                      (match_operand:SI 1 "memory_operand" "")]))
              (clobber (reg:CC FLAGS_REG))])]
  "! optimize_size && ! TARGET_READ_MODIFY"
  [(set (match_dup 2) (match_dup 1))
   (parallel [(set (match_dup 0)
                   (match_op_dup 3 [(match_dup 0) (match_dup 2)]))
              (clobber (reg:CC FLAGS_REG))])]
  "")

(define_peephole2
  [(match_scratch:SI 2 "r")
   (parallel [(set (match_operand:SI 0 "register_operand" "")
                   (match_operator:SI 3 "arith_or_logical_operator"
                     [(match_operand:SI 1 "memory_operand" "")
                      (match_dup 0)]))
              (clobber (reg:CC FLAGS_REG))])]
  "! optimize_size && ! TARGET_READ_MODIFY"
  [(set (match_dup 2) (match_dup 1))
   (parallel [(set (match_dup 0)
                   (match_op_dup 3 [(match_dup 2) (match_dup 0)]))
              (clobber (reg:CC FLAGS_REG))])]
  "")

; Don't do logical operations with memory outputs
;
; These two don't make sense for PPro/PII -- we're expanding a 4-uop
; instruction into two 1-uop insns plus a 2-uop insn.  That last has
; the same decoder scheduling characteristics as the original.

(define_peephole2
  [(match_scratch:SI 2 "r")
```

```
   (parallel [(set (match_operand:SI 0 "memory_operand" "")
                   (match_operator:SI 3 "arith_or_logical_operator"
                     [(match_dup 0)
                      (match_operand:SI 1 "nonmemory_operand" "")]))
              (clobber (reg:CC FLAGS_REG))])]
  "! optimize_size && ! TARGET_READ_MODIFY_WRITE"
  [(set (match_dup 2) (match_dup 0))
   (parallel [(set (match_dup 2)
                   (match_op_dup 3 [(match_dup 2) (match_dup 1)]))
              (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 0) (match_dup 2))]
  "")

(define_peephole2
  [(match_scratch:SI 2 "r")
   (parallel [(set (match_operand:SI 0 "memory_operand" "")
                   (match_operator:SI 3 "arith_or_logical_operator"
                     [(match_operand:SI 1 "nonmemory_operand" "")
                      (match_dup 0)]))
              (clobber (reg:CC FLAGS_REG))])]
  "! optimize_size && ! TARGET_READ_MODIFY_WRITE"
  [(set (match_dup 2) (match_dup 0))
   (parallel [(set (match_dup 2)
                   (match_op_dup 3 [(match_dup 1) (match_dup 2)]))
              (clobber (reg:CC FLAGS_REG))])
   (set (match_dup 0) (match_dup 2))]
  "")

;; Attempt to always use XOR for zeroing registers.
(define_peephole2
  [(set (match_operand 0 "register_operand" "")
(match_operand 1 "const0_operand" ""))]
  "GET_MODE_SIZE (GET_MODE (operands[0])) <= UNITS_PER_WORD
   && (! TARGET_USE_MOV0 || optimize_size)
   && GENERAL_REG_P (operands[0])
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (const_int 0))
      (clobber (reg:CC FLAGS_REG))])]
{
  operands[0] = gen_lowpart (word_mode, operands[0]);
})

(define_peephole2
  [(set (strict_low_part (match_operand 0 "register_operand" ""))
(const_int 0))]
  "(GET_MODE (operands[0]) == QImode
    || GET_MODE (operands[0]) == HImode)
   && (! TARGET_USE_MOV0 || optimize_size)
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (strict_low_part (match_dup 0)) (const_int 0))
```

```
      (clobber (reg:CC FLAGS_REG))])])

;; For HI and SI modes, or $-1,reg is smaller than mov $-1,reg.
(define_peephole2
  [(set (match_operand 0 "register_operand" "")
(const_int -1))]
  "(GET_MODE (operands[0]) == HImode
    || GET_MODE (operands[0]) == SImode
    || (GET_MODE (operands[0]) == DImode && TARGET_64BIT))
   && (optimize_size || TARGET_PENTIUM)
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (const_int -1))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[0] = gen_lowpart (GET_MODE (operands[0]) == DImode ? DImode : SImode,
      operands[0]);")

;; Attempt to convert simple leas to adds. These can be created by
;; move expanders.
(define_peephole2
  [(set (match_operand:SI 0 "register_operand" "")
   (plus:SI (match_dup 0)
 (match_operand:SI 1 "nonmemory_operand" "")))]
  "peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (plus:SI (match_dup 0) (match_dup 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_peephole2
  [(set (match_operand:SI 0 "register_operand" "")
   (subreg:SI (plus:DI (match_operand:DI 1 "register_operand" "")
    (match_operand:DI 2 "nonmemory_operand" "")) 0))]
  "peep2_regno_dead_p (0, FLAGS_REG) && REGNO (operands[0]) == REGNO (operands[1])"
  [(parallel [(set (match_dup 0) (plus:SI (match_dup 0) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[2] = gen_lowpart (SImode, operands[2]);")

(define_peephole2
  [(set (match_operand:DI 0 "register_operand" "")
   (plus:DI (match_dup 0)
 (match_operand:DI 1 "x86_64_general_operand" "")))]
  "peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (plus:DI (match_dup 0) (match_dup 1)))
      (clobber (reg:CC FLAGS_REG))])]
  "")

(define_peephole2
  [(set (match_operand:SI 0 "register_operand" "")
   (mult:SI (match_dup 0)
 (match_operand:SI 1 "const_int_operand" "")))]
  "exact_log2 (INTVAL (operands[1])) >= 0
```

```
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (ashift:SI (match_dup 0) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[2] = GEN_INT (exact_log2 (INTVAL (operands[1])));")

(define_peephole2
  [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (match_dup 0)
 (match_operand:DI 1 "const_int_operand" "")))]
  "exact_log2 (INTVAL (operands[1])) >= 0
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (ashift:DI (match_dup 0) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[2] = GEN_INT (exact_log2 (INTVAL (operands[1])));")

(define_peephole2
  [(set (match_operand:SI 0 "register_operand" "")
   (subreg:SI (mult:DI (match_operand:DI 1 "register_operand" "")
   (match_operand:DI 2 "const_int_operand" "")) 0))]
  "exact_log2 (INTVAL (operands[2])) >= 0
   && REGNO (operands[0]) == REGNO (operands[1])
   && peep2_regno_dead_p (0, FLAGS_REG)"
  [(parallel [(set (match_dup 0) (ashift:SI (match_dup 0) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
  "operands[2] = GEN_INT (exact_log2 (INTVAL (operands[2])));")

;; The ESP adjustments can be done by the push and pop instructions.  Resulting
;; code is shorter, since push is only 1 byte, while add imm, %esp 3 bytes.  On
;; many CPUs it is also faster, since special hardware to avoid esp
;; dependencies is present.

;; While some of these conversions may be done using splitters, we use peepholes
;; in order to allow combine_stack_adjustments pass to see nonobfuscated RTL.

;; Convert prologue esp subtractions to push.
;; We need register to push.  In order to keep verify_flow_info happy we have
;; two choices
;; - use scratch and clobber it in order to avoid dependencies
;; - use already live register
;; We can't use the second way right now, since there is no reliable way how to
;; verify that given register is live.  First choice will also most likely in
;; fewer dependencies.  On the place of esp adjustments it is very likely that
;; call clobbered registers are dead.  We may want to use base pointer as an
;; alternative when no register is available later.

(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -4)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
```

```
  "optimize_size || !TARGET_SUB_ESP_4"
  [(clobber (match_dup 0))
   (parallel [(set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))
      (clobber (mem:BLK (scratch)))])])

(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -8)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_SUB_ESP_8"
  [(clobber (match_dup 0))
   (set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))
   (parallel [(set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))
      (clobber (mem:BLK (scratch)))])])

;; Convert esp subtractions to push.
(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -4)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size || !TARGET_SUB_ESP_4"
  [(clobber (match_dup 0))
   (set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))])

(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int -8)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size || !TARGET_SUB_ESP_8"
  [(clobber (match_dup 0))
   (set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))
   (set (mem:SI (pre_dec:SI (reg:SI SP_REG))) (match_dup 0))])

;; Convert epilogue deallocator to pop.
(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_ADD_ESP_4"
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))
      (clobber (mem:BLK (scratch)))])]
  "")

;; Two pops case is tricky, since pop causes dependency on destination register.
;; We use two registers if available.
(define_peephole2
  [(match_scratch:SI 0 "r")
```

```
   (match_scratch:SI 1 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_ADD_ESP_8"
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))
      (clobber (mem:BLK (scratch)))])
   (parallel [(set (match_dup 1) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])]
  "")

(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size"
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))
      (clobber (mem:BLK (scratch)))])
   (parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])]
  "")

;; Convert esp additions to pop.
(define_peephole2
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])]
  "")

;; Two pops case is tricky, since pop causes dependency on destination register.
;; We use two registers if available.
(define_peephole2
  [(match_scratch:SI 0 "r")
   (match_scratch:SI 1 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])
   (parallel [(set (match_dup 1) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])]
  "")

(define_peephole2
```

```
  [(match_scratch:SI 0 "r")
   (parallel [(set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size"
  [(parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])
   (parallel [(set (match_dup 0) (mem:SI (reg:SI SP_REG)))
      (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG) (const_int 4)))])]
  "")

;; Convert compares with 1 to shorter inc/dec operations when CF is not
;; required and register dies.  Similarly for 128 to plus -128.
(define_peephole2
  [(set (match_operand 0 "flags_reg_operand" "")
(match_operator 1 "compare_operator"
  [(match_operand 2 "register_operand" "")
   (match_operand 3 "const_int_operand" "")]))]
  "(INTVAL (operands[3]) == -1
    || INTVAL (operands[3]) == 1
    || INTVAL (operands[3]) == 128)
   && ix86_match_ccmode (insn, CCGCmode)
   && peep2_reg_dead_p (1, operands[2])"
  [(parallel [(set (match_dup 0)
   (match_op_dup 1 [(match_dup 2) (match_dup 3)]))
      (clobber (match_dup 2))])]
  "")

(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -8)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_SUB_ESP_4"
  [(clobber (match_dup 0))
   (parallel [(set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))
      (clobber (mem:BLK (scratch)))])])

(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -16)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_SUB_ESP_8"
  [(clobber (match_dup 0))
   (set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))
   (parallel [(set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))
      (clobber (mem:BLK (scratch)))])])

;; Convert esp subtractions to push.
(define_peephole2
```

```
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -8)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size || !TARGET_SUB_ESP_4"
  [(clobber (match_dup 0))
   (set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))])

(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int -16)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size || !TARGET_SUB_ESP_8"
  [(clobber (match_dup 0))
   (set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))
   (set (mem:DI (pre_dec:DI (reg:DI SP_REG))) (match_dup 0))])

;; Convert epilogue deallocator to pop.
(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_ADD_ESP_4"
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))
      (clobber (mem:BLK (scratch)))])]
  "")

;; Two pops case is tricky, since pop causes dependency on destination register.
;; We use two registers if available.
(define_peephole2
  [(match_scratch:DI 0 "r")
   (match_scratch:DI 1 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 16)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size || !TARGET_ADD_ESP_8"
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))
      (clobber (mem:BLK (scratch)))])
   (parallel [(set (match_dup 1) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])]
  "")

(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 16)))
      (clobber (reg:CC FLAGS_REG))
      (clobber (mem:BLK (scratch)))])]
  "optimize_size"
```

```
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))
      (clobber (mem:BLK (scratch)))])
   (parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])]
  "")

;; Convert esp additions to pop.
(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])]
  "")

;; Two pops case is tricky, since pop causes dependency on destination register.
;; We use two registers if available.
(define_peephole2
  [(match_scratch:DI 0 "r")
   (match_scratch:DI 1 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 16)))
      (clobber (reg:CC FLAGS_REG))])]
  ""
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])
   (parallel [(set (match_dup 1) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])]
  "")

(define_peephole2
  [(match_scratch:DI 0 "r")
   (parallel [(set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 16)))
      (clobber (reg:CC FLAGS_REG))])]
  "optimize_size"
  [(parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])
   (parallel [(set (match_dup 0) (mem:DI (reg:DI SP_REG)))
      (set (reg:DI SP_REG) (plus:DI (reg:DI SP_REG) (const_int 8)))])]
  "")

;; Convert imul by three, five and nine into lea
(define_peephole2
  [(parallel
    [(set (match_operand:SI 0 "register_operand" "")
  (mult:SI (match_operand:SI 1 "register_operand" "")
   (match_operand:SI 2 "const_int_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  "INTVAL (operands[2]) == 3
```

```
   || INTVAL (operands[2]) == 5
   || INTVAL (operands[2]) == 9"
  [(set (match_dup 0)
        (plus:SI (mult:SI (match_dup 1) (match_dup 2))
                 (match_dup 1)))]
  { operands[2] = GEN_INT (INTVAL (operands[2]) - 1); })

(define_peephole2
  [(parallel
    [(set (match_operand:SI 0 "register_operand" "")
          (mult:SI (match_operand:SI 1 "nonimmediate_operand" "")
                   (match_operand:SI 2 "const_int_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  "!optimize_size
   && (INTVAL (operands[2]) == 3
       || INTVAL (operands[2]) == 5
       || INTVAL (operands[2]) == 9)"
  [(set (match_dup 0) (match_dup 1))
   (set (match_dup 0)
        (plus:SI (mult:SI (match_dup 0) (match_dup 2))
                 (match_dup 0)))]
  { operands[2] = GEN_INT (INTVAL (operands[2]) - 1); })

(define_peephole2
  [(parallel
    [(set (match_operand:DI 0 "register_operand" "")
  (mult:DI (match_operand:DI 1 "register_operand" "")
   (match_operand:DI 2 "const_int_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT
   && (INTVAL (operands[2]) == 3
       || INTVAL (operands[2]) == 5
       || INTVAL (operands[2]) == 9)"
  [(set (match_dup 0)
        (plus:DI (mult:DI (match_dup 1) (match_dup 2))
                 (match_dup 1)))]
  { operands[2] = GEN_INT (INTVAL (operands[2]) - 1); })

(define_peephole2
  [(parallel
    [(set (match_operand:DI 0 "register_operand" "")
          (mult:DI (match_operand:DI 1 "nonimmediate_operand" "")
                   (match_operand:DI 2 "const_int_operand" "")))
     (clobber (reg:CC FLAGS_REG))])]
  "TARGET_64BIT
   && !optimize_size
   && (INTVAL (operands[2]) == 3
       || INTVAL (operands[2]) == 5
       || INTVAL (operands[2]) == 9)"
  [(set (match_dup 0) (match_dup 1))
```

```
   (set (match_dup 0)
        (plus:DI (mult:DI (match_dup 0) (match_dup 2))
                 (match_dup 0)))]
  { operands[2] = GEN_INT (INTVAL (operands[2]) - 1); })

;; Imul $32bit_imm, mem, reg is vector decoded, while
;; imul $32bit_imm, reg, reg is direct decoded.
(define_peephole2
  [(match_scratch:DI 3 "r")
   (parallel [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (match_operand:DI 1 "memory_operand" "")
    (match_operand:DI 2 "immediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_K8 && !optimize_size
   && (GET_CODE (operands[2]) != CONST_INT
       || !CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K'))"
  [(set (match_dup 3) (match_dup 1))
   (parallel [(set (match_dup 0) (mult:DI (match_dup 3) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
"")

(define_peephole2
  [(match_scratch:SI 3 "r")
   (parallel [(set (match_operand:SI 0 "register_operand" "")
   (mult:SI (match_operand:SI 1 "memory_operand" "")
    (match_operand:SI 2 "immediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_K8 && !optimize_size
   && (GET_CODE (operands[2]) != CONST_INT
       || !CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K'))"
  [(set (match_dup 3) (match_dup 1))
   (parallel [(set (match_dup 0) (mult:SI (match_dup 3) (match_dup 2)))
      (clobber (reg:CC FLAGS_REG))])]
"")

(define_peephole2
  [(match_scratch:SI 3 "r")
   (parallel [(set (match_operand:DI 0 "register_operand" "")
   (zero_extend:DI
     (mult:SI (match_operand:SI 1 "memory_operand" "")
      (match_operand:SI 2 "immediate_operand" ""))))
      (clobber (reg:CC FLAGS_REG))])]
  "TARGET_K8 && !optimize_size
   && (GET_CODE (operands[2]) != CONST_INT
       || !CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K'))"
  [(set (match_dup 3) (match_dup 1))
   (parallel [(set (match_dup 0) (zero_extend:DI (mult:SI (match_dup 3) (match_dup 2))))
      (clobber (reg:CC FLAGS_REG))])]
"")
```

```
;; imul $8/16bit_imm, regmem, reg is vector decoded.
;; Convert it into imul reg, reg
;; It would be better to force assembler to encode instruction using long
;; immediate, but there is apparently no way to do so.
(define_peephole2
  [(parallel [(set (match_operand:DI 0 "register_operand" "")
   (mult:DI (match_operand:DI 1 "nonimmediate_operand" "")
    (match_operand:DI 2 "const_int_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_scratch:DI 3 "r")]
  "TARGET_K8 && !optimize_size
   && CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K')"
  [(set (match_dup 3) (match_dup 2))
   (parallel [(set (match_dup 0) (mult:DI (match_dup 0) (match_dup 3)))
      (clobber (reg:CC FLAGS_REG))])]
{
  if (!rtx_equal_p (operands[0], operands[1]))
    emit_move_insn (operands[0], operands[1]);
})

(define_peephole2
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
   (mult:SI (match_operand:SI 1 "nonimmediate_operand" "")
    (match_operand:SI 2 "const_int_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_scratch:SI 3 "r")]
  "TARGET_K8 && !optimize_size
   && CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K')"
  [(set (match_dup 3) (match_dup 2))
   (parallel [(set (match_dup 0) (mult:SI (match_dup 0) (match_dup 3)))
      (clobber (reg:CC FLAGS_REG))])]
{
  if (!rtx_equal_p (operands[0], operands[1]))
    emit_move_insn (operands[0], operands[1]);
})

(define_peephole2
  [(parallel [(set (match_operand:HI 0 "register_operand" "")
   (mult:HI (match_operand:HI 1 "nonimmediate_operand" "")
    (match_operand:HI 2 "immediate_operand" "")))
      (clobber (reg:CC FLAGS_REG))])
   (match_scratch:HI 3 "r")]
  "TARGET_K8 && !optimize_size"
  [(set (match_dup 3) (match_dup 2))
   (parallel [(set (match_dup 0) (mult:HI (match_dup 0) (match_dup 3)))
      (clobber (reg:CC FLAGS_REG))])]
{
  if (!rtx_equal_p (operands[0], operands[1]))
    emit_move_insn (operands[0], operands[1]);
})
```

```
;; Call-value patterns last so that the wildcard operand does not
;; disrupt insn-recog's switch tables.

(define_insn "*call_value_pop_0"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:SI 1 "constant_call_address_operand" ""))
      (match_operand:SI 2 "" "")))
   (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG)
    (match_operand:SI 3 "immediate_operand" "")))]
  "!TARGET_64BIT"
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P1";
  else
    return "call\t%P1";
}
  [(set_attr "type" "callv")])

(define_insn "*call_value_pop_1"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:SI 1 "call_insn_operand" "rsm"))
      (match_operand:SI 2 "" "")))
   (set (reg:SI SP_REG) (plus:SI (reg:SI SP_REG)
    (match_operand:SI 3 "immediate_operand" "i")))]
  "!TARGET_64BIT"
{
  if (constant_call_address_operand (operands[1], Pmode))
    {
      if (SIBLING_CALL_P (insn))
return "jmp\t%P1";
      else
return "call\t%P1";
    }
  if (SIBLING_CALL_P (insn))
    return "jmp\t%A1";
  else
    return "call\t%A1";
}
  [(set_attr "type" "callv")])

(define_insn "*call_value_0"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:SI 1 "constant_call_address_operand" ""))
      (match_operand:SI 2 "" "")))]
  "!TARGET_64BIT"
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P1";
  else
```

```
    return "call\t%P1";
}
  [(set_attr "type" "callv")])

(define_insn "*call_value_0_rex64"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:DI 1 "constant_call_address_operand" ""))
      (match_operand:DI 2 "const_int_operand" "")))]
  "TARGET_64BIT"
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P1";
  else
    return "call\t%P1";
}
  [(set_attr "type" "callv")])

(define_insn "*call_value_1"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:SI 1 "call_insn_operand" "rsm"))
      (match_operand:SI 2 "" "")))]
  "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[1], Pmode))
    return "call\t%P1";
  return "call\t%A1";
}
  [(set_attr "type" "callv")])

(define_insn "*sibcall_value_1"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:SI 1 "sibcall_insn_operand" "s,c,d,a"))
      (match_operand:SI 2 "" "")))]
  "SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[1], Pmode))
    return "jmp\t%P1";
  return "jmp\t%A1";
}
  [(set_attr "type" "callv")])

(define_insn "*call_value_1_rex64"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:DI 1 "call_insn_operand" "rsm"))
      (match_operand:DI 2 "" "")))]
  "!SIBLING_CALL_P (insn) && TARGET_64BIT"
{
  if (constant_call_address_operand (operands[1], Pmode))
    return "call\t%P1";
  return "call\t%A1";
```

```
}
  [(set_attr "type" "callv")])

(define_insn "*sibcall_value_1_rex64"
  [(set (match_operand 0 "" "")
(call (mem:QI (match_operand:DI 1 "constant_call_address_operand" ""))
      (match_operand:DI 2 "" "")))]
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t%P1"
  [(set_attr "type" "callv")])

(define_insn "*sibcall_value_1_rex64_v"
  [(set (match_operand 0 "" "")
(call (mem:QI (reg:DI 40))
      (match_operand:DI 1 "" "")))]
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t*%%r11"
  [(set_attr "type" "callv")])

;; We used to use "int $5", in honor of #BR which maps to interrupt vector 5.
;; That, however, is usually mapped by the OS to SIGSEGV, which is often
;; caught for use by garbage collectors and the like.  Using an insn that
;; maps to SIGILL makes it more likely the program will rightfully die.
;; Keeping with tradition, "6" is in honor of #UD.
(define_insn "trap"
  [(trap_if (const_int 1) (const_int 6))]
  ""
  ".word\t0x0b0f"
  [(set_attr "length" "2")])

(define_expand "sse_prologue_save"
  [(parallel [(set (match_operand:BLK 0 "" "")
   (unspec:BLK [(reg:DI 21)
(reg:DI 22)
(reg:DI 23)
(reg:DI 24)
(reg:DI 25)
(reg:DI 26)
(reg:DI 27)
(reg:DI 28)] UNSPEC_SSE_PROLOGUE_SAVE))
      (use (match_operand:DI 1 "register_operand" ""))
      (use (match_operand:DI 2 "immediate_operand" ""))
      (use (label_ref:DI (match_operand 3 "" "")))])]
  "TARGET_64BIT"
  "")

(define_insn "*sse_prologue_save_insn"
  [(set (mem:BLK (plus:DI (match_operand:DI 0 "register_operand" "R")
  (match_operand:DI 4 "const_int_operand" "n")))
(unspec:BLK [(reg:DI 21)
```

```
     (reg:DI 22)
     (reg:DI 23)
     (reg:DI 24)
     (reg:DI 25)
     (reg:DI 26)
     (reg:DI 27)
     (reg:DI 28)] UNSPEC_SSE_PROLOGUE_SAVE))
   (use (match_operand:DI 1 "register_operand" "r"))
   (use (match_operand:DI 2 "const_int_operand" "i"))
   (use (label_ref:DI (match_operand 3 "" "X")))]
  "TARGET_64BIT
   && INTVAL (operands[4]) + SSE_REGPARM_MAX * 16 - 16 < 128
   && INTVAL (operands[4]) + INTVAL (operands[2]) * 16 >= -128"
  "*
{
  int i;
  operands[0] = gen_rtx_MEM (Pmode,
     gen_rtx_PLUS (Pmode, operands[0], operands[4]));
  output_asm_insn (\"jmp\\t%A1\", operands);
  for (i = SSE_REGPARM_MAX - 1; i >= INTVAL (operands[2]); i--)
    {
      operands[4] = adjust_address (operands[0], DImode, i*16);
      operands[5] = gen_rtx_REG (TImode, SSE_REGNO (i));
      PUT_MODE (operands[4], TImode);
      if (GET_CODE (XEXP (operands[0], 0)) != PLUS)
        output_asm_insn (\"rex\", operands);
      output_asm_insn (\"movaps\\t{%5, %4|%4, %5}\", operands);
    }
  (*targetm.asm_out.internal_label) (asm_out_file, \"L\",
     CODE_LABEL_NUMBER (operands[3]));
  RET;
}
  "
  [(set_attr "type" "other")
   (set_attr "length_immediate" "0")
   (set_attr "length_address" "0")
   (set_attr "length" "135")
   (set_attr "memory" "store")
   (set_attr "modrm" "0")
   (set_attr "mode" "DI")])

(define_expand "prefetch"
  [(prefetch (match_operand 0 "address_operand" "")
     (match_operand:SI 1 "const_int_operand" "")
     (match_operand:SI 2 "const_int_operand" ""))]
  "TARGET_PREFETCH_SSE || TARGET_3DNOW"
{
  int rw = INTVAL (operands[1]);
  int locality = INTVAL (operands[2]);
```

```
  gcc_assert (rw == 0 || rw == 1);
  gcc_assert (locality >= 0 && locality <= 3);
  gcc_assert (GET_MODE (operands[0]) == Pmode
      || GET_MODE (operands[0]) == VOIDmode);

  /* Use 3dNOW prefetch in case we are asking for write prefetch not
     supported by SSE counterpart or the SSE prefetch is not available
     (K6 machines).  Otherwise use SSE prefetch as it allows specifying
     of locality.  */
  if (TARGET_3DNOW && (!TARGET_PREFETCH_SSE || rw))
    operands[2] = GEN_INT (3);
  else
    operands[1] = const0_rtx;
})

(define_insn "*prefetch_sse"
  [(prefetch (match_operand:SI 0 "address_operand" "p")
     (const_int 0)
     (match_operand:SI 1 "const_int_operand" ""))]
  "TARGET_PREFETCH_SSE && !TARGET_64BIT"
{
  static const char * const patterns[4] = {
   "prefetchnta\t%a0", "prefetcht2\t%a0", "prefetcht1\t%a0", "prefetcht0\t%a0"
  };

  int locality = INTVAL (operands[1]);
  gcc_assert (locality >= 0 && locality <= 3);

  return patterns[locality];
}
  [(set_attr "type" "sse")
   (set_attr "memory" "none")])

(define_insn "*prefetch_sse_rex"
  [(prefetch (match_operand:DI 0 "address_operand" "p")
     (const_int 0)
     (match_operand:SI 1 "const_int_operand" ""))]
  "TARGET_PREFETCH_SSE && TARGET_64BIT"
{
  static const char * const patterns[4] = {
   "prefetchnta\t%a0", "prefetcht2\t%a0", "prefetcht1\t%a0", "prefetcht0\t%a0"
  };

  int locality = INTVAL (operands[1]);
  gcc_assert (locality >= 0 && locality <= 3);

  return patterns[locality];
}
  [(set_attr "type" "sse")
   (set_attr "memory" "none")])
```

```
(define_insn "*prefetch_3dnow"
  [(prefetch (match_operand:SI 0 "address_operand" "p")
     (match_operand:SI 1 "const_int_operand" "n")
     (const_int 3))]
  "TARGET_3DNOW && !TARGET_64BIT"
{
  if (INTVAL (operands[1]) == 0)
    return "prefetch\t%a0";
  else
    return "prefetchw\t%a0";
}
  [(set_attr "type" "mmx")
   (set_attr "memory" "none")])

(define_insn "*prefetch_3dnow_rex"
  [(prefetch (match_operand:DI 0 "address_operand" "p")
     (match_operand:SI 1 "const_int_operand" "n")
     (const_int 3))]
  "TARGET_3DNOW && TARGET_64BIT"
{
  if (INTVAL (operands[1]) == 0)
    return "prefetch\t%a0";
  else
    return "prefetchw\t%a0";
}
  [(set_attr "type" "mmx")
   (set_attr "memory" "none")])

(define_expand "stack_protect_set"
  [(match_operand 0 "memory_operand" "")
   (match_operand 1 "memory_operand" "")]
  ""
{
#ifdef TARGET_THREAD_SSP_OFFSET
  if (TARGET_64BIT)
    emit_insn (gen_stack_tls_protect_set_di (operands[0],
GEN_INT (TARGET_THREAD_SSP_OFFSET)));
  else
    emit_insn (gen_stack_tls_protect_set_si (operands[0],
GEN_INT (TARGET_THREAD_SSP_OFFSET)));
#else
  if (TARGET_64BIT)
    emit_insn (gen_stack_protect_set_di (operands[0], operands[1]));
  else
    emit_insn (gen_stack_protect_set_si (operands[0], operands[1]));
#endif
  DONE;
})
```

```
(define_insn "stack_protect_set_si"
  [(set (match_operand:SI 0 "memory_operand" "=m")
(unspec:SI [(match_operand:SI 1 "memory_operand" "m")] UNSPEC_SP_SET))
   (set (match_scratch:SI 2 "=&r") (const_int 0))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "mov{l}\t{%1, %2|%2, %1}\;mov{l}\t{%2, %0|%0, %2}\;xor{l}\t%2, %2"
  [(set_attr "type" "multi")])

(define_insn "stack_protect_set_di"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:DI 1 "memory_operand" "m")] UNSPEC_SP_SET))
   (set (match_scratch:DI 2 "=&r") (const_int 0))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "mov{q}\t{%1, %2|%2, %1}\;mov{q}\t{%2, %0|%0, %2}\;xor{l}\t%k2, %k2"
  [(set_attr "type" "multi")])

(define_insn "stack_tls_protect_set_si"
  [(set (match_operand:SI 0 "memory_operand" "=m")
(unspec:SI [(match_operand:SI 1 "const_int_operand" "i")] UNSPEC_SP_TLS_SET))
   (set (match_scratch:SI 2 "=&r") (const_int 0))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "mov{l}\t{%%gs:%P1, %2|%2, DWORD PTR %%gs:%P1}\;mov{l}\t{%2, %0|%0, %2}\;xor{l}\t%2, %2"
  [(set_attr "type" "multi")])

(define_insn "stack_tls_protect_set_di"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:DI 1 "const_int_operand" "i")] UNSPEC_SP_TLS_SET))
   (set (match_scratch:DI 2 "=&r") (const_int 0))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT"
  "mov{q}\t{%%fs:%P1, %2|%2, QWORD PTR %%fs:%P1}\;mov{q}\t{%2, %0|%0, %2}\;xor{l}\t%k2, %k2"
  [(set_attr "type" "multi")])

(define_expand "stack_protect_test"
  [(match_operand 0 "memory_operand" "")
   (match_operand 1 "memory_operand" "")
   (match_operand 2 "" "")]
  ""
{
  rtx flags = gen_rtx_REG (CCZmode, FLAGS_REG);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  ix86_compare_emitted = flags;

#ifdef TARGET_THREAD_SSP_OFFSET
  if (TARGET_64BIT)
    emit_insn (gen_stack_tls_protect_test_di (flags, operands[0],
```

```
GEN_INT (TARGET_THREAD_SSP_OFFSET)));
  else
    emit_insn (gen_stack_tls_protect_test_si (flags, operands[0],
GEN_INT (TARGET_THREAD_SSP_OFFSET)));
#else
  if (TARGET_64BIT)
    emit_insn (gen_stack_protect_test_di (flags, operands[0], operands[1]));
  else
    emit_insn (gen_stack_protect_test_si (flags, operands[0], operands[1]));
#endif
  emit_jump_insn (gen_beq (operands[2]));
  DONE;
})

(define_insn "stack_protect_test_si"
  [(set (match_operand:CCZ 0 "flags_reg_operand" "")
(unspec:CCZ [(match_operand:SI 1 "memory_operand" "m")
     (match_operand:SI 2 "memory_operand" "m")]
    UNSPEC_SP_TEST))
   (clobber (match_scratch:SI 3 "=&r"))]
  ""
  "mov{l}\t{%1, %3|%3, %1}\;xor{l}\t{%2, %3|%3, %2}"
  [(set_attr "type" "multi")])

(define_insn "stack_protect_test_di"
  [(set (match_operand:CCZ 0 "flags_reg_operand" "")
(unspec:CCZ [(match_operand:DI 1 "memory_operand" "m")
     (match_operand:DI 2 "memory_operand" "m")]
    UNSPEC_SP_TEST))
   (clobber (match_scratch:DI 3 "=&r"))]
  "TARGET_64BIT"
  "mov{q}\t{%1, %3|%3, %1}\;xor{q}\t{%2, %3|%3, %2}"
  [(set_attr "type" "multi")])

(define_insn "stack_tls_protect_test_si"
  [(set (match_operand:CCZ 0 "flags_reg_operand" "")
(unspec:CCZ [(match_operand:SI 1 "memory_operand" "m")
     (match_operand:SI 2 "const_int_operand" "i")]
    UNSPEC_SP_TLS_TEST))
   (clobber (match_scratch:SI 3 "=r"))]
  ""
  "mov{l}\t{%1, %3|%3, %1}\;xor{l}\t{%%gs:%P2, %3|%3, DWORD PTR %%gs:%P2}"
  [(set_attr "type" "multi")])

(define_insn "stack_tls_protect_test_di"
  [(set (match_operand:CCZ 0 "flags_reg_operand" "")
(unspec:CCZ [(match_operand:DI 1 "memory_operand" "m")
     (match_operand:DI 2 "const_int_operand" "i")]
    UNSPEC_SP_TLS_TEST))
   (clobber (match_scratch:DI 3 "=r"))]
```

```
  "TARGET_64BIT"
  "mov{q}\t{%1, %3|%3, %1}\;xor{q}\t{%%fs:%P2, %3|%3, QWORD PTR %%fs:%P2}"
  [(set_attr "type" "multi")])
```

1.8 GCC machine description for SSE instructions

;;(include "sse.md")

```
;; 16 byte integral modes handled by SSE, minus TImode, which gets
;; special-cased for TARGET_64BIT.
(define_mode_macro SSEMODEI [V16QI V8HI V4SI V2DI])

;; All 16-byte vector modes handled by SSE
(define_mode_macro SSEMODE [V16QI V8HI V4SI V2DI V4SF V2DF])

;; Mix-n-match
(define_mode_macro SSEMODE12 [V16QI V8HI])
(define_mode_macro SSEMODE24 [V8HI V4SI])
(define_mode_macro SSEMODE14 [V16QI V4SI])
(define_mode_macro SSEMODE124 [V16QI V8HI V4SI])
(define_mode_macro SSEMODE248 [V8HI V4SI V2DI])

;; Mapping from integer vector mode to mnemonic suffix
(define_mode_attr ssevecsize [(V16QI "b") (V8HI "w") (V4SI "d") (V2DI "q")])

;; Patterns whose name begins with "sse{,2,3}_" are invoked by intrinsics.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Move patterns
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; All of these patterns are enabled for SSE1 as well as SSE2.
;; This is essential for maintaining stable calling conventions.

(define_expand "mov<mode>"
  [(set (match_operand:SSEMODEI 0 "nonimmediate_operand" "")
(match_operand:SSEMODEI 1 "nonimmediate_operand" ""))]
  "TARGET_SSE"
{
  ix86_expand_vector_move (<MODE>mode, operands);
  DONE;
})

(define_insn "*mov<mode>_internal"
  [(set (match_operand:SSEMODEI 0 "nonimmediate_operand" "=x,x ,m")
(match_operand:SSEMODEI 1 "vector_move_operand"  "C ,xm,x"))]
```

```
  "TARGET_SSE && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
{
  switch (which_alternative)
    {
    case 0:
      if (get_attr_mode (insn) == MODE_V4SF)
return "xorps\t%0, %0";
      else
return "pxor\t%0, %0";
    case 1:
    case 2:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
return "movdqa\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "sselog1,ssemov,ssemov")
   (set (attr "mode")
(cond [(eq (symbol_ref "TARGET_SSE2") (const_int 0))
 (const_string "V4SF")

       (eq_attr "alternative" "0,1")
 (if_then_else
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
   (const_string "TI"))
       (eq_attr "alternative" "2")
 (if_then_else
   (ior (ne (symbol_ref "TARGET_SSE_TYPELESS_STORES")
    (const_int 0))
(ne (symbol_ref "optimize_size")
    (const_int 0)))
   (const_string "V4SF")
   (const_string "TI"))]
       (const_string "TI")))])

(define_expand "movv4sf"
  [(set (match_operand:V4SF 0 "nonimmediate_operand" "")
(match_operand:V4SF 1 "nonimmediate_operand" ""))]
  "TARGET_SSE"
{
  ix86_expand_vector_move (V4SFmode, operands);
  DONE;
})

(define_insn "*movv4sf_internal"
```

```
  [(set (match_operand:V4SF 0 "nonimmediate_operand" "=x,x,m")
(match_operand:V4SF 1 "vector_move_operand" "C,xm,x"))]
  "TARGET_SSE"
  "@
   xorps\t%0, %0
   movaps\t{%1, %0|%0, %1}
   movaps\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog1,ssemov,ssemov")
   (set_attr "mode" "V4SF")])

(define_split
  [(set (match_operand:V4SF 0 "register_operand" "")
(match_operand:V4SF 1 "zero_extended_scalar_load_operand" ""))]
  "TARGET_SSE && reload_completed"
  [(set (match_dup 0)
(vec_merge:V4SF
  (vec_duplicate:V4SF (match_dup 1))
  (match_dup 2)
  (const_int 1)))]
{
  operands[1] = simplify_gen_subreg (SFmode, operands[1], V4SFmode, 0);
  operands[2] = CONST0_RTX (V4SFmode);
})

(define_expand "movv2df"
  [(set (match_operand:V2DF 0 "nonimmediate_operand" "")
(match_operand:V2DF 1 "nonimmediate_operand" ""))]
  "TARGET_SSE"
{
  ix86_expand_vector_move (V2DFmode, operands);
  DONE;
})

(define_insn "*movv2df_internal"
  [(set (match_operand:V2DF 0 "nonimmediate_operand" "=x,x,m")
(match_operand:V2DF 1 "vector_move_operand" "C,xm,x"))]
  "TARGET_SSE && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
{
  switch (which_alternative)
    {
    case 0:
      if (get_attr_mode (insn) == MODE_V4SF)
return "xorps\t%0, %0";
      else
return "xorpd\t%0, %0";
    case 1:
    case 2:
      if (get_attr_mode (insn) == MODE_V4SF)
return "movaps\t{%1, %0|%0, %1}";
      else
```

```
return "movapd\t{%1, %0|%0, %1}";
    default:
      gcc_unreachable ();
    }
}
  [(set_attr "type" "sselog1,ssemov,ssemov")
   (set (attr "mode")
(cond [(eq (symbol_ref "TARGET_SSE2") (const_int 0))
 (const_string "V4SF")
       (eq_attr "alternative" "0,1")
 (if_then_else
   (ne (symbol_ref "optimize_size")
       (const_int 0))
   (const_string "V4SF")
   (const_string "V2DF"))
       (eq_attr "alternative" "2")
 (if_then_else
   (ior (ne (symbol_ref "TARGET_SSE_TYPELESS_STORES")
    (const_int 0))
(ne (symbol_ref "optimize_size")
    (const_int 0)))
   (const_string "V4SF")
   (const_string "V2DF"))]
       (const_string "V2DF")))])

(define_split
  [(set (match_operand:V2DF 0 "register_operand" "")
(match_operand:V2DF 1 "zero_extended_scalar_load_operand" ""))]
  "TARGET_SSE2 && reload_completed"
  [(set (match_dup 0) (vec_concat:V2DF (match_dup 1) (match_dup 2)))]
{
  operands[1] = simplify_gen_subreg (DFmode, operands[1], V2DFmode, 0);
  operands[2] = CONST0_RTX (DFmode);
})

(define_expand "push<mode>1"
  [(match_operand:SSEMODE 0 "register_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_push (<MODE>mode, operands[0]);
  DONE;
})

(define_expand "movmisalign<mode>"
  [(set (match_operand:SSEMODE 0 "nonimmediate_operand" "")
(match_operand:SSEMODE 1 "nonimmediate_operand" ""))]
  "TARGET_SSE"
{
  ix86_expand_vector_move_misalign (<MODE>mode, operands);
  DONE;
```

```
})

(define_insn "sse_movups"
  [(set (match_operand:V4SF 0 "nonimmediate_operand" "=x,m")
(unspec:V4SF [(match_operand:V4SF 1 "nonimmediate_operand" "xm,x")]
     UNSPEC_MOVU))]
  "TARGET_SSE && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "movups\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_movupd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand" "=x,m")
(unspec:V2DF [(match_operand:V2DF 1 "nonimmediate_operand" "xm,x")]
     UNSPEC_MOVU))]
  "TARGET_SSE2 && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "movupd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_movdqu"
  [(set (match_operand:V16QI 0 "nonimmediate_operand" "=x,m")
(unspec:V16QI [(match_operand:V16QI 1 "nonimmediate_operand" "xm,x")]
      UNSPEC_MOVU))]
  "TARGET_SSE2 && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "movdqu\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "TI")])

(define_insn "sse_movntv4sf"
  [(set (match_operand:V4SF 0 "memory_operand" "=m")
(unspec:V4SF [(match_operand:V4SF 1 "register_operand" "x")]
     UNSPEC_MOVNT))]
  "TARGET_SSE"
  "movntps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V4SF")])

(define_insn "sse2_movntv2df"
  [(set (match_operand:V2DF 0 "memory_operand" "=m")
(unspec:V2DF [(match_operand:V2DF 1 "register_operand" "x")]
     UNSPEC_MOVNT))]
  "TARGET_SSE2"
  "movntpd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_movntv2di"
  [(set (match_operand:V2DI 0 "memory_operand" "=m")
(unspec:V2DI [(match_operand:V2DI 1 "register_operand" "x")]
```

```
     UNSPEC_MOVNT))]
  "TARGET_SSE2"
  "movntdq\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

(define_insn "sse2_movntsi"
  [(set (match_operand:SI 0 "memory_operand" "=m")
(unspec:SI [(match_operand:SI 1 "register_operand" "r")]
   UNSPEC_MOVNT))]
  "TARGET_SSE2"
  "movnti\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_insn "sse3_lddqu"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(unspec:V16QI [(match_operand:V16QI 1 "memory_operand" "m")]
      UNSPEC_LDQQU))]
  "TARGET_SSE3"
  "lddqu\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point arithmetic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "negv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "")
(neg:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_expand_fp_absneg_operator (NEG, V4SFmode, operands); DONE;")

(define_expand "absv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "")
(abs:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_expand_fp_absneg_operator (ABS, V4SFmode, operands); DONE;")

(define_expand "addv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(plus:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
   (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (PLUS, V4SFmode, operands);")

(define_insn "*addv4sf3"
```

```
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(plus:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && ix86_binary_operator_ok (PLUS, V4SFmode, operands)"
  "addps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmaddv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (plus:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
     (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE && ix86_binary_operator_ok (PLUS, V4SFmode, operands)"
  "addss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])

(define_expand "subv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(minus:V4SF (match_operand:V4SF 1 "register_operand" "")
    (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (MINUS, V4SFmode, operands);")

(define_insn "*subv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(minus:V4SF (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
  "subps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmsubv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (minus:V4SF (match_operand:V4SF 1 "register_operand" "0")
      (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE"
  "subss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "SF")])

(define_expand "mulv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
```

```
(mult:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
   (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (MULT, V4SFmode, operands);")

(define_insn "*mulv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(mult:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && ix86_binary_operator_ok (MULT, V4SFmode, operands)"
  "mulps\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemul")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmmulv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (mult:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
     (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE && ix86_binary_operator_ok (MULT, V4SFmode, operands)"
  "mulss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemul")
   (set_attr "mode" "SF")])

(define_expand "divv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(div:V4SF (match_operand:V4SF 1 "register_operand" "")
  (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (DIV, V4SFmode, operands);")

(define_insn "*divv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(div:V4SF (match_operand:V4SF 1 "register_operand" "0")
  (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
  "divps\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssediv")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmdivv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (div:V4SF (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE"
```

```
  "divss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssediv")
   (set_attr "mode" "SF")])

(define_insn "sse_rcpv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(unspec:V4SF
 [(match_operand:V4SF 1 "nonimmediate_operand" "xm")] UNSPEC_RCP))]
  "TARGET_SSE"
  "rcpps\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmrcpv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (unspec:V4SF [(match_operand:V4SF 1 "nonimmediate_operand" "xm")]
       UNSPEC_RCP)
  (match_operand:V4SF 2 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE"
  "rcpss\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_insn "sse_rsqrtv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(unspec:V4SF
  [(match_operand:V4SF 1 "nonimmediate_operand" "xm")] UNSPEC_RSQRT))]
  "TARGET_SSE"
  "rsqrtps\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmrsqrtv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (unspec:V4SF [(match_operand:V4SF 1 "nonimmediate_operand" "xm")]
       UNSPEC_RSQRT)
  (match_operand:V4SF 2 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE"
  "rsqrtss\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_insn "sqrtv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(sqrt:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
```

```
  "sqrtps\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmsqrtv4sf2"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (sqrt:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "xm"))
  (match_operand:V4SF 2 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE"
  "sqrtss\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

;; ??? For !flag_finite_math_only, the representation with SMIN/SMAX
;; isn't really correct, as those rtl operators aren't defined when
;; applied to NaNs.  Hopefully the optimizers won't get too smart on us.

(define_expand "smaxv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(smax:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
   (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
{
  if (!flag_finite_math_only)
    operands[1] = force_reg (V4SFmode, operands[1]);
  ix86_fixup_binary_operands_no_copy (SMAX, V4SFmode, operands);
})

(define_insn "*smaxv4sf3_finite"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(smax:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && flag_finite_math_only
   && ix86_binary_operator_ok (SMAX, V4SFmode, operands)"
  "maxps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "*smaxv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(smax:V4SF (match_operand:V4SF 1 "register_operand" "0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
  "maxps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "*sse_vmsmaxv4sf3_finite"
```

```
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
 (smax:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
 (match_dup 1)
 (const_int 1)))]
  "TARGET_SSE && flag_finite_math_only
   && ix86_binary_operator_ok (SMAX, V4SFmode, operands)"
  "maxss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_insn "sse_vmsmaxv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
 (smax:V4SF (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
 (match_dup 1)
 (const_int 1)))]
  "TARGET_SSE"
  "maxss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_expand "sminv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(smin:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
   (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
{
  if (!flag_finite_math_only)
    operands[1] = force_reg (V4SFmode, operands[1]);
  ix86_fixup_binary_operands_no_copy (SMIN, V4SFmode, operands);
})

(define_insn "*sminv4sf3_finite"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(smin:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && flag_finite_math_only
   && ix86_binary_operator_ok (SMIN, V4SFmode, operands)"
  "minps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "*sminv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(smin:V4SF (match_operand:V4SF 1 "register_operand" "0")
   (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
```

```
  "minps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "*sse_vmsminv4sf3_finite"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
 (smin:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
 (match_dup 1)
 (const_int 1)))]
  "TARGET_SSE && flag_finite_math_only
   && ix86_binary_operator_ok (SMIN, V4SFmode, operands)"
  "minss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_insn "sse_vmsminv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
 (smin:V4SF (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
 (match_dup 1)
 (const_int 1)))]
  "TARGET_SSE"
  "minss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

(define_insn "sse3_addsubv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (plus:V4SF
    (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (minus:V4SF (match_dup 1) (match_dup 2))
  (const_int 5)))]
  "TARGET_SSE3"
  "addsubps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V4SF")])

(define_insn "sse3_haddv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_concat:V4SF
  (vec_concat:V2SF
    (plus:SF
      (vec_select:SF
(match_operand:V4SF 1 "register_operand" "0")
(parallel [(const_int 0)]))
```

```
      (vec_select:SF (match_dup 1) (parallel [(const_int 1)])))
    (plus:SF
      (vec_select:SF (match_dup 1) (parallel [(const_int 2)]))
      (vec_select:SF (match_dup 1) (parallel [(const_int 3)])))))
  (vec_concat:V2SF
    (plus:SF
      (vec_select:SF
(match_operand:V4SF 2 "nonimmediate_operand" "xm")
(parallel [(const_int 0)]))
      (vec_select:SF (match_dup 2) (parallel [(const_int 1)])))
    (plus:SF
      (vec_select:SF (match_dup 2) (parallel [(const_int 2)]))
      (vec_select:SF (match_dup 2) (parallel [(const_int 3)]))))))]
  "TARGET_SSE3"
  "haddps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V4SF")])

(define_insn "sse3_hsubv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_concat:V4SF
  (vec_concat:V2SF
    (minus:SF
      (vec_select:SF
(match_operand:V4SF 1 "register_operand" "0")
(parallel [(const_int 0)]))
      (vec_select:SF (match_dup 1) (parallel [(const_int 1)])))
    (minus:SF
      (vec_select:SF (match_dup 1) (parallel [(const_int 2)]))
      (vec_select:SF (match_dup 1) (parallel [(const_int 3)])))))
  (vec_concat:V2SF
    (minus:SF
      (vec_select:SF
(match_operand:V4SF 2 "nonimmediate_operand" "xm")
(parallel [(const_int 0)]))
      (vec_select:SF (match_dup 2) (parallel [(const_int 1)])))
    (minus:SF
      (vec_select:SF (match_dup 2) (parallel [(const_int 2)]))
      (vec_select:SF (match_dup 2) (parallel [(const_int 3)]))))))]
  "TARGET_SSE3"
  "hsubps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V4SF")])

(define_expand "reduc_plus_v4sf"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand:V4SF 1 "register_operand" "")]
  "TARGET_SSE"
{
  if (TARGET_SSE3)
```

```
    {
      rtx tmp = gen_reg_rtx (V4SFmode);
      emit_insn (gen_sse3_haddv4sf3 (tmp, operands[1], operands[1]));
      emit_insn (gen_sse3_haddv4sf3 (operands[0], tmp, tmp));
    }
  else
    ix86_expand_reduc_v4sf (gen_addv4sf3, operands[0], operands[1]);
  DONE;
})

(define_expand "reduc_smax_v4sf"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand:V4SF 1 "register_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_reduc_v4sf (gen_smaxv4sf3, operands[0], operands[1]);
  DONE;
})

(define_expand "reduc_smin_v4sf"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand:V4SF 1 "register_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_reduc_v4sf (gen_sminv4sf3, operands[0], operands[1]);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point comparisons
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse_maskcmpv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(match_operator:V4SF 3 "sse_comparison_operator"
[(match_operand:V4SF 1 "register_operand" "0")
 (match_operand:V4SF 2 "nonimmediate_operand" "xm")]))]
  "TARGET_SSE"
  "cmp%D3ps\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "V4SF")])

(define_insn "sse_vmmaskcmpv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
 (match_operator:V4SF 3 "sse_comparison_operator"
[(match_operand:V4SF 1 "register_operand" "0")
 (match_operand:V4SF 2 "register_operand" "x")])
```

```
 (match_dup 1)
 (const_int 1)))]
  "TARGET_SSE"
  "cmp%D3ss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "SF")])

(define_insn "sse_comi"
  [(set (reg:CCFP FLAGS_REG)
(compare:CCFP
  (vec_select:SF
    (match_operand:V4SF 0 "register_operand" "x")
    (parallel [(const_int 0)]))
  (vec_select:SF
    (match_operand:V4SF 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0)])))))]
  "TARGET_SSE"
  "comiss\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecomi")
   (set_attr "mode" "SF")])

(define_insn "sse_ucomi"
  [(set (reg:CCFPU FLAGS_REG)
(compare:CCFPU
  (vec_select:SF
    (match_operand:V4SF 0 "register_operand" "x")
    (parallel [(const_int 0)]))
  (vec_select:SF
    (match_operand:V4SF 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0)])))))]
  "TARGET_SSE"
  "ucomiss\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecomi")
   (set_attr "mode" "SF")])

(define_expand "vcondv4sf"
  [(set (match_operand:V4SF 0 "register_operand" "")
        (if_then_else:V4SF
          (match_operator 3 ""
            [(match_operand:V4SF 4 "nonimmediate_operand" "")
             (match_operand:V4SF 5 "nonimmediate_operand" "")])
          (match_operand:V4SF 1 "general_operand" "")
          (match_operand:V4SF 2 "general_operand" "")))]
  "TARGET_SSE"
{
  if (ix86_expand_fp_vcond (operands))
    DONE;
  else
    FAIL;
})
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point logical operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "andv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(and:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
  (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (AND, V4SFmode, operands);")

(define_insn "*andv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(and:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
  (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && ix86_binary_operator_ok (AND, V4SFmode, operands)"
  "andps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_insn "sse_nandv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(and:V4SF (not:V4SF (match_operand:V4SF 1 "register_operand" "0"))
  (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE"
  "andnps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_expand "iorv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
(ior:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
  (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (IOR, V4SFmode, operands);")

(define_insn "*iorv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(ior:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
  (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && ix86_binary_operator_ok (IOR, V4SFmode, operands)"
  "orps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_expand "xorv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "")
```

```
(xor:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "")
  (match_operand:V4SF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE"
  "ix86_fixup_binary_operands_no_copy (XOR, V4SFmode, operands);")

(define_insn "*xorv4sf3"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(xor:V4SF (match_operand:V4SF 1 "nonimmediate_operand" "%0")
  (match_operand:V4SF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE && ix86_binary_operator_ok (XOR, V4SFmode, operands)"
  "xorps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

;; Also define scalar versions.  These are used for abs, neg, and
;; conditional move.  Using subregs into vector modes causes register
;; allocation lossage.  These patterns do not allow memory operands
;; because the native instructions read the full 128-bits.

(define_insn "*andsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(and:SF (match_operand:SF 1 "register_operand" "0")
(match_operand:SF 2 "register_operand" "x")))]
  "TARGET_SSE"
  "andps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_insn "*nandsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(and:SF (not:SF (match_operand:SF 1 "register_operand" "0"))
(match_operand:SF 2 "register_operand" "x")))]
  "TARGET_SSE"
  "andnps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_insn "*iorsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(ior:SF (match_operand:SF 1 "register_operand" "0")
(match_operand:SF 2 "register_operand" "x")))]
  "TARGET_SSE"
  "orps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_insn "*xorsf3"
  [(set (match_operand:SF 0 "register_operand" "=x")
(xor:SF (match_operand:SF 1 "register_operand" "0")
(match_operand:SF 2 "register_operand" "x")))]
```

```
  "TARGET_SSE"
  "xorps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point conversion operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse_cvtpi2ps"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (vec_duplicate:V4SF
    (float:V2SF (match_operand:V2SI 2 "nonimmediate_operand" "ym")))
  (match_operand:V4SF 1 "register_operand" "0")
  (const_int 3)))]
  "TARGET_SSE"
  "cvtpi2ps\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V4SF")])

(define_insn "sse_cvtps2pi"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_select:V2SI
  (unspec:V4SI [(match_operand:V4SF 1 "nonimmediate_operand" "xm")]
       UNSPEC_FIX_NOTRUNC)
  (parallel [(const_int 0) (const_int 1)])))]
  "TARGET_SSE"
  "cvtps2pi\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "unit" "mmx")
   (set_attr "mode" "DI")])

(define_insn "sse_cvttps2pi"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_select:V2SI
  (fix:V4SI (match_operand:V4SF 1 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0) (const_int 1)])))]
  "TARGET_SSE"
  "cvttps2pi\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "unit" "mmx")
   (set_attr "mode" "SF")])

(define_insn "sse_cvtsi2ss"
  [(set (match_operand:V4SF 0 "register_operand" "=x,x")
(vec_merge:V4SF
  (vec_duplicate:V4SF
```

```
    (float:SF (match_operand:SI 2 "nonimmediate_operand" "r,m")))
  (match_operand:V4SF 1 "register_operand" "0,0")
  (const_int 1)))]
  "TARGET_SSE"
  "cvtsi2ss\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "vector,double")
   (set_attr "mode" "SF")])

(define_insn "sse_cvtsi2ssq"
  [(set (match_operand:V4SF 0 "register_operand" "=x,x")
(vec_merge:V4SF
  (vec_duplicate:V4SF
    (float:SF (match_operand:DI 2 "nonimmediate_operand" "r,rm")))
  (match_operand:V4SF 1 "register_operand" "0,0")
  (const_int 1)))]
  "TARGET_SSE && TARGET_64BIT"
  "cvtsi2ssq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "vector,double")
   (set_attr "mode" "SF")])

(define_insn "sse_cvtss2si"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(unspec:SI
  [(vec_select:SF
     (match_operand:V4SF 1 "nonimmediate_operand" "x,m")
     (parallel [(const_int 0)]))]
  UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE"
  "cvtss2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "SI")])

(define_insn "sse_cvtss2siq"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(unspec:DI
  [(vec_select:SF
     (match_operand:V4SF 1 "nonimmediate_operand" "x,m")
     (parallel [(const_int 0)]))]
  UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE && TARGET_64BIT"
  "cvtss2siq\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "DI")])

(define_insn "sse_cvttss2si"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
```

```
(fix:SI
  (vec_select:SF
    (match_operand:V4SF 1 "nonimmediate_operand" "x,m")
    (parallel [(const_int 0)]))))]
  "TARGET_SSE"
  "cvttss2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "SI")])

(define_insn "sse_cvttss2siq"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(fix:DI
  (vec_select:SF
    (match_operand:V4SF 1 "nonimmediate_operand" "x,m")
    (parallel [(const_int 0)]))))]
  "TARGET_SSE && TARGET_64BIT"
  "cvttss2siq\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "DI")])

(define_insn "sse2_cvtdq2ps"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(float:V4SF (match_operand:V4SI 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "cvtdq2ps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_cvtps2dq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(unspec:V4SI [(match_operand:V4SF 1 "nonimmediate_operand" "xm")]
     UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE2"
  "cvtps2dq\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

(define_insn "sse2_cvttps2dq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(fix:V4SI (match_operand:V4SF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "cvttps2dq\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point element swizzling
```

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse_movhlps"
  [(set (match_operand:V4SF 0 "nonimmediate_operand"     "=x,x,m")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "nonimmediate_operand" " 0,o,x")
    (match_operand:V4SF 2 "nonimmediate_operand" " x,0,0"))
  (parallel [(const_int 6)
     (const_int 7)
     (const_int 2)
     (const_int 3)])))]
  "TARGET_SSE && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   movhlps\t{%2, %0|%0, %2}
   movlps\t{%H1, %0|%0, %H1}
   movhps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V4SF,V2SF,V2SF")])

(define_insn "sse_movlhps"
  [(set (match_operand:V4SF 0 "nonimmediate_operand"     "=x,x,o")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "nonimmediate_operand" " 0,0,0")
    (match_operand:V4SF 2 "nonimmediate_operand" " x,m,x"))
  (parallel [(const_int 0)
     (const_int 1)
     (const_int 4)
     (const_int 5)])))]
  "TARGET_SSE && ix86_binary_operator_ok (UNKNOWN, V4SFmode, operands)"
  "@
   movlhps\t{%2, %0|%0, %2}
   movhps\t{%2, %0|%0, %2}
   movlps\t{%2, %H0|%H0, %2}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V4SF,V2SF,V2SF")])

(define_insn "sse_unpckhps"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 2) (const_int 6)
     (const_int 3) (const_int 7)])))]
  "TARGET_SSE"
  "unpckhps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
```

```
   (set_attr "mode" "V4SF")])

(define_insn "sse_unpcklps"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0) (const_int 4)
     (const_int 1) (const_int 5)])))]
  "TARGET_SSE"
  "unpcklps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

;; These are modeled with the same vec_concat as the others so that we
;; capture users of shufps that can use the new instructions
(define_insn "sse3_movshdup"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "nonimmediate_operand" "xm")
    (match_dup 1))
  (parallel [(const_int 1)
     (const_int 1)
     (const_int 7)
     (const_int 7)])))]
  "TARGET_SSE3"
  "movshdup\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_insn "sse3_movsldup"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "nonimmediate_operand" "xm")
    (match_dup 1))
  (parallel [(const_int 0)
     (const_int 0)
     (const_int 6)
     (const_int 6)])))]
  "TARGET_SSE3"
  "movsldup\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V4SF")])

(define_expand "sse_shufps"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand:V4SF 1 "register_operand" "")
```

```
   (match_operand:V4SF 2 "nonimmediate_operand" "")
   (match_operand:SI 3 "const_int_operand" "")]
  "TARGET_SSE"
{
  int mask = INTVAL (operands[3]);
  emit_insn (gen_sse_shufps_1 (operands[0], operands[1], operands[2],
       GEN_INT ((mask >> 0) & 3),
       GEN_INT ((mask >> 2) & 3),
       GEN_INT (((mask >> 4) & 3) + 4),
       GEN_INT (((mask >> 6) & 3) + 4)));
  DONE;
})

(define_insn "sse_shufps_1"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_select:V4SF
  (vec_concat:V8SF
    (match_operand:V4SF 1 "register_operand" "0")
    (match_operand:V4SF 2 "nonimmediate_operand" "xm"))
  (parallel [(match_operand 3 "const_0_to_3_operand" "")
     (match_operand 4 "const_0_to_3_operand" "")
     (match_operand 5 "const_4_to_7_operand" "")
     (match_operand 6 "const_4_to_7_operand" "")]))))]
  "TARGET_SSE"
{
  int mask = 0;
  mask |= INTVAL (operands[3]) << 0;
  mask |= INTVAL (operands[4]) << 2;
  mask |= (INTVAL (operands[5]) - 4) << 4;
  mask |= (INTVAL (operands[6]) - 4) << 6;
  operands[3] = GEN_INT (mask);

  return "shufps\t{%3, %2, %0|%0, %2, %3}";
}
  [(set_attr "type" "sselog")
   (set_attr "mode" "V4SF")])

(define_insn "sse_storehps"
  [(set (match_operand:V2SF 0 "nonimmediate_operand" "=m,x,x")
(vec_select:V2SF
  (match_operand:V4SF 1 "nonimmediate_operand" "x,x,o")
  (parallel [(const_int 2) (const_int 3)])))]
  "TARGET_SSE"
  "@
   movhps\t{%1, %0|%0, %1}
   movhlps\t{%1, %0|%0, %1}
   movlps\t{%H1, %0|%0, %H1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V2SF,V4SF,V2SF")])
```

```
(define_insn "sse_loadhps"
  [(set (match_operand:V4SF 0 "nonimmediate_operand" "=x,x,o")
(vec_concat:V4SF
  (vec_select:V2SF
    (match_operand:V4SF 1 "nonimmediate_operand" "0,0,0")
    (parallel [(const_int 0) (const_int 1)]))
  (match_operand:V2SF 2 "nonimmediate_operand" "m,x,x")))]
  "TARGET_SSE"
  "@
   movhps\t{%2, %0|%0, %2}
   movlhps\t{%2, %0|%0, %2}
   movlps\t{%2, %H0|%H0, %2}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V2SF,V4SF,V2SF")])

(define_insn "sse_storelps"
  [(set (match_operand:V2SF 0 "nonimmediate_operand" "=m,x,x")
(vec_select:V2SF
  (match_operand:V4SF 1 "nonimmediate_operand" "x,x,m")
  (parallel [(const_int 0) (const_int 1)])))]
  "TARGET_SSE"
  "@
   movlps\t{%1, %0|%0, %1}
   movaps\t{%1, %0|%0, %1}
   movlps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V2SF,V4SF,V2SF")])

(define_insn "sse_loadlps"
  [(set (match_operand:V4SF 0 "nonimmediate_operand" "=x,x,m")
(vec_concat:V4SF
  (match_operand:V2SF 2 "nonimmediate_operand" "0,m,x")
  (vec_select:V2SF
    (match_operand:V4SF 1 "nonimmediate_operand" "x,0,0")
    (parallel [(const_int 2) (const_int 3)]))))]
  "TARGET_SSE"
  "@
   shufps\t{$0xe4, %1, %0|%0, %1, 0xe4}
   movlps\t{%2, %0|%0, %2}
   movlps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog,ssemov,ssemov")
   (set_attr "mode" "V4SF,V2SF,V2SF")])

(define_insn "sse_movss"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_merge:V4SF
  (match_operand:V4SF 2 "register_operand" "x")
  (match_operand:V4SF 1 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE"
```

```
  "movss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "SF")])

(define_insn "*vec_dupv4sf"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_duplicate:V4SF
  (match_operand:SF 1 "register_operand" "0")))]
  "TARGET_SSE"
  "shufps\t{$0, %0, %0|%0, %0, 0}"
  [(set_attr "type" "sselog1")
   (set_attr "mode" "V4SF")])

;; ??? In theory we can match memory for the MMX alternative, but allowing
;; nonimmediate_operand for operand 2 and *not* allowing memory for the SSE
;; alternatives pretty much forces the MMX alternative to be chosen.
(define_insn "*sse_concatv2sf"
  [(set (match_operand:V2SF 0 "register_operand"     "=x,x,*y,*y")
(vec_concat:V2SF
  (match_operand:SF 1 "nonimmediate_operand" " 0,m, 0, m")
  (match_operand:SF 2 "vector_move_operand"  " x,C,*y, C")))]
  "TARGET_SSE"
  "@
   unpcklps\t{%2, %0|%0, %2}
   movss\t{%1, %0|%0, %1}
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog,ssemov,mmxcvt,mmxmov")
   (set_attr "mode" "V4SF,SF,DI,DI")])

(define_insn "*sse_concatv4sf"
  [(set (match_operand:V4SF 0 "register_operand"   "=x,x")
(vec_concat:V4SF
  (match_operand:V2SF 1 "register_operand" " 0,0")
  (match_operand:V2SF 2 "nonimmediate_operand" " x,m")))]
  "TARGET_SSE"
  "@
   movlhps\t{%2, %0|%0, %2}
   movhps\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V4SF,V2SF")])

(define_expand "vec_initv4sf"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})
```

```
(define_insn "*vec_setv4sf_0"
  [(set (match_operand:V4SF 0 "nonimmediate_operand"  "=x,x,Y ,m")
(vec_merge:V4SF
  (vec_duplicate:V4SF
    (match_operand:SF 2 "general_operand"     " x,m,*r,x*rfF"))
  (match_operand:V4SF 1 "vector_move_operand" " 0,C,C ,0")
  (const_int 1)))]
  "TARGET_SSE"
  "@
   movss\t{%2, %0|%0, %2}
   movss\t{%2, %0|%0, %2}
   movd\t{%2, %0|%0, %2}
   #"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "SF")])

(define_split
  [(set (match_operand:V4SF 0 "memory_operand" "")
(vec_merge:V4SF
  (vec_duplicate:V4SF
    (match_operand:SF 1 "nonmemory_operand" ""))
  (match_dup 0)
  (const_int 1)))]
  "TARGET_SSE && reload_completed"
  [(const_int 0)]
{
  emit_move_insn (adjust_address (operands[0], SFmode, 0), operands[1]);
  DONE;
})

(define_expand "vec_setv4sf"
  [(match_operand:V4SF 0 "register_operand" "")
   (match_operand:SF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_insn_and_split "*vec_extractv4sf_0"
  [(set (match_operand:SF 0 "nonimmediate_operand" "=x,m,fr")
(vec_select:SF
  (match_operand:V4SF 1 "nonimmediate_operand" "xm,x,m")
  (parallel [(const_int 0)])))]
  "TARGET_SSE && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "#"
  "&& reload_completed"
```

```
  [(const_int 0)]
{
  rtx op1 = operands[1];
  if (REG_P (op1))
    op1 = gen_rtx_REG (SFmode, REGNO (op1));
  else
    op1 = gen_lowpart (SFmode, op1);
  emit_move_insn (operands[0], op1);
  DONE;
})

(define_expand "vec_extractv4sf"
  [(match_operand:SF 0 "register_operand" "")
   (match_operand:V4SF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel double-precision floating point arithmetic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "negv2df2"
  [(set (match_operand:V2DF 0 "register_operand" "")
(neg:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_expand_fp_absneg_operator (NEG, V2DFmode, operands); DONE;")

(define_expand "absv2df2"
  [(set (match_operand:V2DF 0 "register_operand" "")
(abs:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_expand_fp_absneg_operator (ABS, V2DFmode, operands); DONE;")

(define_expand "addv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(plus:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
   (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (PLUS, V2DFmode, operands);")

(define_insn "*addv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(plus:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
```

```
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (PLUS, V2DFmode, operands)"
  "addpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_vmaddv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (plus:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2 && ix86_binary_operator_ok (PLUS, V4SFmode, operands)"
  "addsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_expand "subv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(minus:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
    (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (MINUS, V2DFmode, operands);")

(define_insn "*subv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(minus:V2DF (match_operand:V2DF 1 "register_operand" "0")
    (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "subpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_vmsubv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (minus:V2DF (match_operand:V2DF 1 "register_operand" "0")
      (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2"
  "subsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_expand "mulv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(mult:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
   (match_operand:V2DF 2 "nonimmediate_operand" "")))]
```

```
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (MULT, V2DFmode, operands);")

(define_insn "*mulv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(mult:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V2DFmode, operands)"
  "mulpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemul")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_vmmulv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (mult:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V2DFmode, operands)"
  "mulsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemul")
   (set_attr "mode" "DF")])

(define_expand "divv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(div:V2DF (match_operand:V2DF 1 "register_operand" "")
  (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (DIV, V2DFmode, operands);")

(define_insn "*divv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(div:V2DF (match_operand:V2DF 1 "register_operand" "0")
  (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "divpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssediv")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_vmdivv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (div:V2DF (match_operand:V2DF 1 "register_operand" "0")
    (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2"
  "divsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssediv")
```

```
   (set_attr "mode" "DF")])

(define_insn "sqrtv2df2"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(sqrt:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "sqrtpd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_vmsqrtv2df2"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (sqrt:V2DF (match_operand:V2DF 1 "register_operand" "xm"))
  (match_operand:V2DF 2 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE2"
  "sqrtsd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sse")
   (set_attr "mode" "SF")])

;; ??? For !flag_finite_math_only, the representation with SMIN/SMAX
;; isn't really correct, as those rtl operators aren't defined when
;; applied to NaNs.  Hopefully the optimizers won't get too smart on us.

(define_expand "smaxv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(smax:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
   (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
{
  if (!flag_finite_math_only)
    operands[1] = force_reg (V2DFmode, operands[1]);
  ix86_fixup_binary_operands_no_copy (SMAX, V2DFmode, operands);
})

(define_insn "*smaxv2df3_finite"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(smax:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && flag_finite_math_only
   && ix86_binary_operator_ok (SMAX, V2DFmode, operands)"
  "maxpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "*smaxv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(smax:V2DF (match_operand:V2DF 1 "register_operand" "0")
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
```

```
  "TARGET_SSE2"
  "maxpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "*sse2_vmsmaxv2df3_finite"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (smax:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2 && flag_finite_math_only
   && ix86_binary_operator_ok (SMAX, V2DFmode, operands)"
  "maxsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_insn "sse2_vmsmaxv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (smax:V2DF (match_operand:V2DF 1 "register_operand" "0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2"
  "maxsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_expand "sminv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(smin:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
   (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
{
  if (!flag_finite_math_only)
    operands[1] = force_reg (V2DFmode, operands[1]);
  ix86_fixup_binary_operands_no_copy (SMIN, V2DFmode, operands);
})

(define_insn "*sminv2df3_finite"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(smin:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && flag_finite_math_only
   && ix86_binary_operator_ok (SMIN, V2DFmode, operands)"
  "minpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])
```

```
(define_insn "*sminv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(smin:V2DF (match_operand:V2DF 1 "register_operand" "0")
   (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "minpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "*sse2_vmsminv2df3_finite"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (smin:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2 && flag_finite_math_only
   && ix86_binary_operator_ok (SMIN, V2DFmode, operands)"
  "minsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_insn "sse2_vmsminv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (smin:V2DF (match_operand:V2DF 1 "register_operand" "0")
     (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2"
  "minsd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "DF")])

(define_insn "sse3_addsubv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (plus:V2DF
    (match_operand:V2DF 1 "register_operand" "0")
    (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (minus:V2DF (match_dup 1) (match_dup 2))
  (const_int 1)))]
  "TARGET_SSE3"
  "addsubpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "sse3_haddv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
```

```
(vec_concat:V2DF
  (plus:DF
    (vec_select:DF
      (match_operand:V2DF 1 "register_operand" "0")
      (parallel [(const_int 0)]))
    (vec_select:DF (match_dup 1) (parallel [(const_int 1)])))
  (plus:DF
    (vec_select:DF
      (match_operand:V2DF 2 "nonimmediate_operand" "xm")
      (parallel [(const_int 0)]))
    (vec_select:DF (match_dup 2) (parallel [(const_int 1)]))))]
  "TARGET_SSE3"
  "haddpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

(define_insn "sse3_hsubv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_concat:V2DF
  (minus:DF
    (vec_select:DF
      (match_operand:V2DF 1 "register_operand" "0")
      (parallel [(const_int 0)]))
    (vec_select:DF (match_dup 1) (parallel [(const_int 1)])))
  (minus:DF
    (vec_select:DF
      (match_operand:V2DF 2 "nonimmediate_operand" "xm")
      (parallel [(const_int 0)]))
    (vec_select:DF (match_dup 2) (parallel [(const_int 1)]))))]
  "TARGET_SSE3"
  "hsubpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseadd")
   (set_attr "mode" "V2DF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel double-precision floating point comparisons
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_maskcmpv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(match_operator:V2DF 3 "sse_comparison_operator"
[(match_operand:V2DF 1 "register_operand" "0")
 (match_operand:V2DF 2 "nonimmediate_operand" "xm")]))]
  "TARGET_SSE2"
  "cmp%D3pd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "V2DF")])
```

```
(define_insn "sse2_vmmaskcmpv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (match_operator:V2DF 3 "sse_comparison_operator"
[(match_operand:V2DF 1 "register_operand" "0")
 (match_operand:V2DF 2 "nonimmediate_operand" "xm")])
  (match_dup 1)
  (const_int 1)))]
  "TARGET_SSE2"
  "cmp%D3sd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "DF")])

(define_insn "sse2_comi"
  [(set (reg:CCFP FLAGS_REG)
(compare:CCFP
  (vec_select:DF
    (match_operand:V2DF 0 "register_operand" "x")
    (parallel [(const_int 0)]))
  (vec_select:DF
    (match_operand:V2DF 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0)])))))]
  "TARGET_SSE2"
  "comisd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecomi")
   (set_attr "mode" "DF")])

(define_insn "sse2_ucomi"
  [(set (reg:CCFPU FLAGS_REG)
(compare:CCFPU
  (vec_select:DF
    (match_operand:V2DF 0 "register_operand" "x")
    (parallel [(const_int 0)]))
  (vec_select:DF
    (match_operand:V2DF 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0)])))))]
  "TARGET_SSE2"
  "ucomisd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecomi")
   (set_attr "mode" "DF")])

(define_expand "vcondv2df"
  [(set (match_operand:V2DF 0 "register_operand" "")
        (if_then_else:V2DF
          (match_operator 3 ""
            [(match_operand:V2DF 4 "nonimmediate_operand" "")
             (match_operand:V2DF 5 "nonimmediate_operand" "")])
          (match_operand:V2DF 1 "general_operand" "")
          (match_operand:V2DF 2 "general_operand" "")))]
  "TARGET_SSE2"
```

```
{
  if (ix86_expand_fp_vcond (operands))
    DONE;
  else
    FAIL;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel double-precision floating point logical operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "andv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(and:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
  (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (AND, V2DFmode, operands);")

(define_insn "*andv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(and:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
  (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (AND, V2DFmode, operands)"
  "andpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_nandv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(and:V2DF (not:V2DF (match_operand:V2DF 1 "register_operand" "0"))
  (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "andnpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_expand "iorv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(ior:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
  (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (IOR, V2DFmode, operands);")

(define_insn "*iorv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(ior:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
  (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (IOR, V2DFmode, operands)"
```

```
  "orpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_expand "xorv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "")
(xor:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "")
  (match_operand:V2DF 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (XOR, V2DFmode, operands);")

(define_insn "*xorv2df3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(xor:V2DF (match_operand:V2DF 1 "nonimmediate_operand" "%0")
  (match_operand:V2DF 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (XOR, V2DFmode, operands)"
  "xorpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

;; Also define scalar versions.  These are used for abs, neg, and
;; conditional move.  Using subregs into vector modes causes register
;; allocation lossage.  These patterns do not allow memory operands
;; because the native instructions read the full 128-bits.

(define_insn "*anddf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(and:DF (match_operand:DF 1 "register_operand" "0")
(match_operand:DF 2 "register_operand" "x")))]
  "TARGET_SSE2"
  "andpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_insn "*nanddf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(and:DF (not:DF (match_operand:DF 1 "register_operand" "0"))
(match_operand:DF 2 "register_operand" "x")))]
  "TARGET_SSE2"
  "andnpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_insn "*iordf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(ior:DF (match_operand:DF 1 "register_operand" "0")
(match_operand:DF 2 "register_operand" "x")))]
  "TARGET_SSE2"
  "orpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
```

```
   (set_attr "mode" "V2DF")])

(define_insn "*xordf3"
  [(set (match_operand:DF 0 "register_operand" "=x")
(xor:DF (match_operand:DF 1 "register_operand" "0")
(match_operand:DF 2 "register_operand" "x")))]
  "TARGET_SSE2"
  "xorpd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel double-precision floating point conversion operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_cvtpi2pd"
  [(set (match_operand:V2DF 0 "register_operand" "=x,x")
(float:V2DF (match_operand:V2SI 1 "nonimmediate_operand" "y,m")))]
  "TARGET_SSE2"
  "cvtpi2pd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "unit" "mmx,*")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_cvtpd2pi"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(unspec:V2SI [(match_operand:V2DF 1 "nonimmediate_operand" "xm")]
     UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE2"
  "cvtpd2pi\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "unit" "mmx")
   (set_attr "mode" "DI")])

(define_insn "sse2_cvttpd2pi"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(fix:V2SI (match_operand:V2DF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "cvttpd2pi\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "unit" "mmx")
   (set_attr "mode" "TI")])

(define_insn "sse2_cvtsi2sd"
  [(set (match_operand:V2DF 0 "register_operand" "=x,x")
(vec_merge:V2DF
  (vec_duplicate:V2DF
    (float:DF (match_operand:SI 2 "nonimmediate_operand" "r,m")))
```

```
  (match_operand:V2DF 1 "register_operand" "0,0")
  (const_int 1)))]
  "TARGET_SSE2"
  "cvtsi2sd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,direct")])

(define_insn "sse2_cvtsi2sdq"
  [(set (match_operand:V2DF 0 "register_operand" "=x,x")
(vec_merge:V2DF
  (vec_duplicate:V2DF
    (float:DF (match_operand:DI 2 "nonimmediate_operand" "r,m")))
  (match_operand:V2DF 1 "register_operand" "0,0")
  (const_int 1)))]
  "TARGET_SSE2 && TARGET_64BIT"
  "cvtsi2sdq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DF")
   (set_attr "athlon_decode" "double,direct")])

(define_insn "sse2_cvtsd2si"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(unspec:SI
  [(vec_select:DF
     (match_operand:V2DF 1 "nonimmediate_operand" "x,m")
     (parallel [(const_int 0)]))]
  UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE2"
  "cvtsd2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "SI")])

(define_insn "sse2_cvtsd2siq"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(unspec:DI
  [(vec_select:DF
     (match_operand:V2DF 1 "nonimmediate_operand" "x,m")
     (parallel [(const_int 0)]))]
  UNSPEC_FIX_NOTRUNC))]
  "TARGET_SSE2 && TARGET_64BIT"
  "cvtsd2siq\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "athlon_decode" "double,vector")
   (set_attr "mode" "DI")])

(define_insn "sse2_cvttsd2si"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
(fix:SI
```

```
  (vec_select:DF
    (match_operand:V2DF 1 "nonimmediate_operand" "x,m")
    (parallel [(const_int 0)])))]
  "TARGET_SSE2"
  "cvttsd2si\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "SI")
   (set_attr "athlon_decode" "double,vector")])

(define_insn "sse2_cvttsd2siq"
  [(set (match_operand:DI 0 "register_operand" "=r,r")
(fix:DI
  (vec_select:DF
    (match_operand:V2DF 1 "nonimmediate_operand" "x,m")
    (parallel [(const_int 0)])))]
  "TARGET_SSE2 && TARGET_64BIT"
  "cvttsd2siq\t{%1, %0|%0, %1}"
  [(set_attr "type" "sseicvt")
   (set_attr "mode" "DI")
   (set_attr "athlon_decode" "double,vector")])

(define_insn "sse2_cvtdq2pd"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(float:V2DF
  (vec_select:V2SI
    (match_operand:V4SI 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0) (const_int 1)])))]
  "TARGET_SSE2"
  "cvtdq2pd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_expand "sse2_cvtpd2dq"
  [(set (match_operand:V4SI 0 "register_operand" "")
(vec_concat:V4SI
  (unspec:V2SI [(match_operand:V2DF 1 "nonimmediate_operand" "")]
       UNSPEC_FIX_NOTRUNC)
  (match_dup 2)))]
  "TARGET_SSE2"
  "operands[2] = CONST0_RTX (V2SImode);")

(define_insn "*sse2_cvtpd2dq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(vec_concat:V4SI
  (unspec:V2SI [(match_operand:V2DF 1 "nonimmediate_operand" "xm")]
       UNSPEC_FIX_NOTRUNC)
  (match_operand:V2SI 2 "const0_operand" "")))]
  "TARGET_SSE2"
  "cvtpd2dq\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
```

```
   (set_attr "mode" "TI")])

(define_expand "sse2_cvttpd2dq"
  [(set (match_operand:V4SI 0 "register_operand" "")
(vec_concat:V4SI
  (fix:V2SI (match_operand:V2DF 1 "nonimmediate_operand" ""))
  (match_dup 2)))]
  "TARGET_SSE2"
  "operands[2] = CONST0_RTX (V2SImode);")

(define_insn "*sse2_cvttpd2dq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(vec_concat:V4SI
  (fix:V2SI (match_operand:V2DF 1 "nonimmediate_operand" "xm"))
  (match_operand:V2SI 2 "const0_operand" "")))]
  "TARGET_SSE2"
  "cvttpd2dq\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

(define_insn "sse2_cvtsd2ss"
  [(set (match_operand:V4SF 0 "register_operand" "=x,x")
(vec_merge:V4SF
  (vec_duplicate:V4SF
    (float_truncate:V2SF
      (match_operand:V2DF 2 "nonimmediate_operand" "x,m")))
  (match_operand:V4SF 1 "register_operand" "0,0")
  (const_int 1)))]
  "TARGET_SSE2"
  "cvtsd2ss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecvt")
   (set_attr "athlon_decode" "vector,double")
   (set_attr "mode" "SF")])

(define_insn "sse2_cvtss2sd"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_merge:V2DF
  (float_extend:V2DF
    (vec_select:V2SF
      (match_operand:V4SF 2 "nonimmediate_operand" "xm")
      (parallel [(const_int 0) (const_int 1)])))
  (match_operand:V2DF 1 "register_operand" "0")
  (const_int 1)))]
  "TARGET_SSE2"
  "cvtss2sd\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "DF")])

(define_expand "sse2_cvtpd2ps"
  [(set (match_operand:V4SF 0 "register_operand" "")
```

```
(vec_concat:V4SF
  (float_truncate:V2SF
    (match_operand:V2DF 1 "nonimmediate_operand" "xm"))
  (match_dup 2)))]
  "TARGET_SSE2"
  "operands[2] = CONST0_RTX (V2SFmode);")

(define_insn "*sse2_cvtpd2ps"
  [(set (match_operand:V4SF 0 "register_operand" "=x")
(vec_concat:V4SF
  (float_truncate:V2SF
    (match_operand:V2DF 1 "nonimmediate_operand" "xm"))
  (match_operand:V2SF 2 "const0_operand" "")))]
  "TARGET_SSE2"
  "cvtpd2ps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V4SF")])

(define_insn "sse2_cvtps2pd"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(float_extend:V2DF
  (vec_select:V2SF
    (match_operand:V4SF 1 "nonimmediate_operand" "xm")
    (parallel [(const_int 0) (const_int 1)])))]
  "TARGET_SSE2"
  "cvtps2pd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel double-precision floating point element swizzling
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_unpckhpd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"     "=x,x,m")
(vec_select:V2DF
  (vec_concat:V4DF
    (match_operand:V2DF 1 "nonimmediate_operand" " 0,o,x")
    (match_operand:V2DF 2 "nonimmediate_operand" " x,0,0"))
  (parallel [(const_int 1)
     (const_int 3)])))]
  "TARGET_SSE2 && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   unpckhpd\t{%2, %0|%0, %2}
   movlpd\t{%H1, %0|%0, %H1}
   movhpd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog,ssemov,ssemov")
   (set_attr "mode" "V2DF,V1DF,V1DF")])
```

```
(define_insn "*sse3_movddup"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"     "=x,o")
(vec_select:V2DF
  (vec_concat:V4DF
    (match_operand:V2DF 1 "nonimmediate_operand" "xm,x")
    (match_dup 1))
  (parallel [(const_int 0)
     (const_int 2)])))]
  "TARGET_SSE3 && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   movddup\t{%1, %0|%0, %1}
   #"
  [(set_attr "type" "sselog,ssemov")
   (set_attr "mode" "V2DF")])

(define_split
  [(set (match_operand:V2DF 0 "memory_operand" "")
(vec_select:V2DF
  (vec_concat:V4DF
    (match_operand:V2DF 1 "register_operand" "")
    (match_dup 1))
  (parallel [(const_int 0)
     (const_int 2)])))]
  "TARGET_SSE3 && reload_completed"
  [(const_int 0)]
{
  rtx low = gen_rtx_REG (DFmode, REGNO (operands[1]));
  emit_move_insn (adjust_address (operands[0], DFmode, 0), low);
  emit_move_insn (adjust_address (operands[0], DFmode, 8), low);
  DONE;
})

(define_insn "sse2_unpcklpd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"     "=x,x,o")
(vec_select:V2DF
  (vec_concat:V4DF
    (match_operand:V2DF 1 "nonimmediate_operand" " 0,0,0")
    (match_operand:V2DF 2 "nonimmediate_operand" " x,m,x"))
  (parallel [(const_int 0)
     (const_int 2)])))]
  "TARGET_SSE2 && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   unpcklpd\t{%2, %0|%0, %2}
   movhpd\t{%2, %0|%0, %2}
   movlpd\t{%2, %H0|%H0, %2}"
  [(set_attr "type" "sselog,ssemov,ssemov")
   (set_attr "mode" "V2DF,V1DF,V1DF")])

(define_expand "sse2_shufpd"
```

```
  [(match_operand:V2DF 0 "register_operand" "")
   (match_operand:V2DF 1 "register_operand" "")
   (match_operand:V2DF 2 "nonimmediate_operand" "")
   (match_operand:SI 3 "const_int_operand" "")]
  "TARGET_SSE2"
{
  int mask = INTVAL (operands[3]);
  emit_insn (gen_sse2_shufpd_1 (operands[0], operands[1], operands[2],
GEN_INT (mask & 1),
GEN_INT (mask & 2 ? 3 : 2)));
  DONE;
})

(define_insn "sse2_shufpd_1"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_select:V2DF
  (vec_concat:V4DF
    (match_operand:V2DF 1 "register_operand" "0")
    (match_operand:V2DF 2 "nonimmediate_operand" "xm"))
  (parallel [(match_operand 3 "const_0_to_1_operand" "")
     (match_operand 4 "const_2_to_3_operand" "")]))))]
  "TARGET_SSE2"
{
  int mask;
  mask = INTVAL (operands[3]);
  mask |= (INTVAL (operands[4]) - 2) << 1;
  operands[3] = GEN_INT (mask);

  return "shufpd\t{%3, %2, %0|%0, %2, %3}";
}
  [(set_attr "type" "sselog")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_storehpd"
  [(set (match_operand:DF 0 "nonimmediate_operand"     "=m,x,x*fr")
(vec_select:DF
  (match_operand:V2DF 1 "nonimmediate_operand" " x,0,o")
  (parallel [(const_int 1)])))]
  "TARGET_SSE2 && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
   movhpd\t{%1, %0|%0, %1}
   unpckhpd\t%0, %0
   #"
  [(set_attr "type" "ssemov,sselog1,ssemov")
   (set_attr "mode" "V1DF,V2DF,DF")])

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(vec_select:DF
  (match_operand:V2DF 1 "memory_operand" "")
```

```
  (parallel [(const_int 1)]))]
  "TARGET_SSE2 && reload_completed"
  [(set (match_dup 0) (match_dup 1))]
{
  operands[1] = adjust_address (operands[1], DFmode, 8);
})

(define_insn "sse2_storelpd"
  [(set (match_operand:DF 0 "nonimmediate_operand"     "=m,x,x*fr")
(vec_select:DF
  (match_operand:V2DF 1 "nonimmediate_operand" " x,x,m")
  (parallel [(const_int 0)])))]
  "TARGET_SSE2 && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
   movlpd\t{%1, %0|%0, %1}
   #
   #"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "V1DF,DF,DF")])

(define_split
  [(set (match_operand:DF 0 "register_operand" "")
(vec_select:DF
  (match_operand:V2DF 1 "nonimmediate_operand" "")
  (parallel [(const_int 0)])))]
  "TARGET_SSE2 && reload_completed"
  [(const_int 0)]
{
  rtx op1 = operands[1];
  if (REG_P (op1))
    op1 = gen_rtx_REG (DFmode, REGNO (op1));
  else
    op1 = gen_lowpart (DFmode, op1);
  emit_move_insn (operands[0], op1);
  DONE;
})

(define_insn "sse2_loadhpd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"     "=x,x,x,o")
(vec_concat:V2DF
  (vec_select:DF
    (match_operand:V2DF 1 "nonimmediate_operand" " 0,0,x,0")
    (parallel [(const_int 0)]))
  (match_operand:DF 2 "nonimmediate_operand"     " m,x,0,x*fr")))]
  "TARGET_SSE2 && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   movhpd\t{%2, %0|%0, %2}
   unpcklpd\t{%2, %0|%0, %2}
   shufpd\t{$1, %1, %0|%0, %1, 1}
   #"
```

```
  [(set_attr "type" "ssemov,sselog,sselog,other")
   (set_attr "mode" "V1DF,V2DF,V2DF,DF")])

(define_split
  [(set (match_operand:V2DF 0 "memory_operand" "")
(vec_concat:V2DF
  (vec_select:DF (match_dup 0) (parallel [(const_int 0)]))
  (match_operand:DF 1 "register_operand" "")))]
  "TARGET_SSE2 && reload_completed"
  [(set (match_dup 0) (match_dup 1))]
{
  operands[0] = adjust_address (operands[0], DFmode, 8);
})

(define_insn "sse2_loadlpd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"    "=x,x,x,x,x,m")
(vec_concat:V2DF
  (match_operand:DF 2 "nonimmediate_operand"    " m,m,x,0,0,x*fr")
  (vec_select:DF
    (match_operand:V2DF 1 "vector_move_operand" " C,0,0,x,o,0")
    (parallel [(const_int 1)])))))]
  "TARGET_SSE2 && !(MEM_P (operands[1]) && MEM_P (operands[2]))"
  "@
   movsd\t{%2, %0|%0, %2}
   movlpd\t{%2, %0|%0, %2}
   movsd\t{%2, %0|%0, %2}
   shufpd\t{$2, %2, %0|%0, %2, 2}
   movhpd\t{%H1, %0|%0, %H1}
   #"
  [(set_attr "type" "ssemov,ssemov,ssemov,sselog,ssemov,other")
   (set_attr "mode" "DF,V1DF,V1DF,V2DF,V1DF,DF")])

(define_split
  [(set (match_operand:V2DF 0 "memory_operand" "")
(vec_concat:V2DF
  (match_operand:DF 1 "register_operand" "")
  (vec_select:DF (match_dup 0) (parallel [(const_int 1)]))))]
  "TARGET_SSE2 && reload_completed"
  [(set (match_dup 0) (match_dup 1))]
{
  operands[0] = adjust_address (operands[0], DFmode, 8);
})

(define_insn "sse2_movsd"
  [(set (match_operand:V2DF 0 "nonimmediate_operand"   "=x,x,m,x,x,o")
(vec_merge:V2DF
  (match_operand:V2DF 2 "nonimmediate_operand" " x,m,x,0,0,0")
  (match_operand:V2DF 1 "nonimmediate_operand" " 0,0,0,x,o,x")
  (const_int 1)))]
  "TARGET_SSE2"
```

```
  "@
   movsd\t{%2, %0|%0, %2}
   movlpd\t{%2, %0|%0, %2}
   movlpd\t{%2, %0|%0, %2}
   shufpd\t{$2, %2, %0|%0, %2, 2}
   movhps\t{%H1, %0|%0, %H1
   movhps\t{%1, %H0|%H0, %1"
  [(set_attr "type" "ssemov,ssemov,ssemov,sselog,ssemov,ssemov")
   (set_attr "mode" "DF,V1DF,V1DF,V2DF,V1DF,V1DF")])

(define_insn "*vec_dupv2df_sse3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_duplicate:V2DF
  (match_operand:DF 1 "nonimmediate_operand" "xm")))]
  "TARGET_SSE3"
  "movddup\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog1")
   (set_attr "mode" "DF")])

(define_insn "*vec_dupv2df"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_duplicate:V2DF
  (match_operand:DF 1 "register_operand" "0")))]
  "TARGET_SSE2"
  "unpcklpd\t%0, %0"
  [(set_attr "type" "sselog1")
   (set_attr "mode" "V4SF")])

(define_insn "*vec_concatv2df_sse3"
  [(set (match_operand:V2DF 0 "register_operand" "=x")
(vec_concat:V2DF
  (match_operand:DF 1 "nonimmediate_operand" "xm")
  (match_dup 1)))]
  "TARGET_SSE3"
  "movddup\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog1")
   (set_attr "mode" "DF")])

(define_insn "*vec_concatv2df"
  [(set (match_operand:V2DF 0 "register_operand"     "=Y,Y,Y,x,x")
(vec_concat:V2DF
  (match_operand:DF 1 "nonimmediate_operand" " 0,0,m,0,0")
  (match_operand:DF 2 "vector_move_operand"  " Y,m,C,x,m")))]
  "TARGET_SSE"
  "@
   unpcklpd\t{%2, %0|%0, %2}
   movhpd\t{%2, %0|%0, %2}
   movsd\t{%1, %0|%0, %1}
   movlhps\t{%2, %0|%0, %2}
   movhps\t{%2, %0|%0, %2}"
```

```
  [(set_attr "type" "sselog,ssemov,ssemov,ssemov,ssemov")
   (set_attr "mode" "V2DF,V1DF,DF,V4SF,V2SF")])

(define_expand "vec_setv2df"
  [(match_operand:V2DF 0 "register_operand" "")
   (match_operand:DF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv2df"
  [(match_operand:DF 0 "register_operand" "")
   (match_operand:V2DF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv2df"
  [(match_operand:V2DF 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral arithmetic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "neg<mode>2"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(minus:SSEMODEI
  (match_dup 2)
  (match_operand:SSEMODEI 1 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "operands[2] = force_reg (<MODE>mode, CONST0_RTX (<MODE>mode));")

(define_expand "add<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
```

```
(plus:SSEMODEI (match_operand:SSEMODEI 1 "nonimmediate_operand" "")
       (match_operand:SSEMODEI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (PLUS, <MODE>mode, operands);")

(define_insn "*add<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(plus:SSEMODEI
  (match_operand:SSEMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (PLUS, <MODE>mode, operands)"
  "padd<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_insn "sse2_ssadd<mode>3"
  [(set (match_operand:SSEMODE12 0 "register_operand" "=x")
(ss_plus:SSEMODE12
  (match_operand:SSEMODE12 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODE12 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (SS_PLUS, <MODE>mode, operands)"
  "padds<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_insn "sse2_usadd<mode>3"
  [(set (match_operand:SSEMODE12 0 "register_operand" "=x")
(us_plus:SSEMODE12
  (match_operand:SSEMODE12 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODE12 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (US_PLUS, <MODE>mode, operands)"
  "paddus<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "sub<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(minus:SSEMODEI (match_operand:SSEMODEI 1 "register_operand" "")
(match_operand:SSEMODEI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (MINUS, <MODE>mode, operands);")

(define_insn "*sub<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(minus:SSEMODEI
  (match_operand:SSEMODEI 1 "register_operand" "0")
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "psub<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
```

```
   (set_attr "mode" "TI")])

(define_insn "sse2_sssub<mode>3"
  [(set (match_operand:SSEMODE12 0 "register_operand" "=x")
(ss_minus:SSEMODE12
  (match_operand:SSEMODE12 1 "register_operand" "0")
  (match_operand:SSEMODE12 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "psubs<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_insn "sse2_ussub<mode>3"
  [(set (match_operand:SSEMODE12 0 "register_operand" "=x")
(us_minus:SSEMODE12
  (match_operand:SSEMODE12 1 "register_operand" "0")
  (match_operand:SSEMODE12 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "psubus<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "mulv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "")
(mult:V16QI (match_operand:V16QI 1 "register_operand" "")
    (match_operand:V16QI 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx t[12], op0;
  int i;

  for (i = 0; i < 12; ++i)
    t[i] = gen_reg_rtx (V16QImode);

  /* Unpack data such that we've got a source byte in each low byte of
     each word.  We don't care what goes into the high byte of each word.
     Rather than trying to get zero in there, most convenient is to let
     it be a copy of the low byte.  */
  emit_insn (gen_sse2_punpckhbw (t[0], operands[1], operands[1]));
  emit_insn (gen_sse2_punpckhbw (t[1], operands[2], operands[2]));
  emit_insn (gen_sse2_punpcklbw (t[2], operands[1], operands[1]));
  emit_insn (gen_sse2_punpcklbw (t[3], operands[2], operands[2]));

  /* Multiply words.  The end-of-line annotations here give a picture of what
     the output of that instruction looks like.  Dot means don't care; the
     letters are the bytes of the result with A being the most significant.  */
  emit_insn (gen_mulv8hi3 (gen_lowpart (V8HImode, t[4]), /* .A.B.C.D.E.F.G.H */
   gen_lowpart (V8HImode, t[0]),
   gen_lowpart (V8HImode, t[1])));
  emit_insn (gen_mulv8hi3 (gen_lowpart (V8HImode, t[5]), /* .I.J.K.L.M.N.O.P */
```

```
   gen_lowpart (V8HImode, t[2]),
   gen_lowpart (V8HImode, t[3])));

  /* Extract the relevant bytes and merge them back together.  */
  emit_insn (gen_sse2_punpckhbw (t[6], t[5], t[4])); /* ..AI..BJ..CK..DL */
  emit_insn (gen_sse2_punpcklbw (t[7], t[5], t[4])); /* ..EM..FN..GO..HP */
  emit_insn (gen_sse2_punpckhbw (t[8], t[7], t[6])); /* ....AEIM....BFJN */
  emit_insn (gen_sse2_punpcklbw (t[9], t[7], t[6])); /* ....CGKO....DHLP */
  emit_insn (gen_sse2_punpckhbw (t[10], t[9], t[8])); /* ........ACEGIKMO */
  emit_insn (gen_sse2_punpcklbw (t[11], t[9], t[8])); /* ........BDFHJLNP */

  op0 = operands[0];
  emit_insn (gen_sse2_punpcklbw (op0, t[11], t[10])); /* ABCDEFGHIJKLMNOP */
  DONE;
})

(define_expand "mulv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "")
(mult:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "")
   (match_operand:V8HI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (MULT, V8HImode, operands);")

(define_insn "*mulv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(mult:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "%0")
   (match_operand:V8HI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V8HImode, operands)"
  "pmullw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseimul")
   (set_attr "mode" "TI")])

(define_insn "sse2_smulv8hi3_highpart"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(truncate:V8HI
  (lshiftrt:V8SI
    (mult:V8SI
      (sign_extend:V8SI
(match_operand:V8HI 1 "nonimmediate_operand" "%0"))
      (sign_extend:V8SI
(match_operand:V8HI 2 "nonimmediate_operand" "xm")))
    (const_int 16))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V8HImode, operands)"
  "pmulhw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseimul")
   (set_attr "mode" "TI")])

(define_insn "sse2_umulv8hi3_highpart"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(truncate:V8HI
```

```
  (lshiftrt:V8SI
    (mult:V8SI
      (zero_extend:V8SI
(match_operand:V8HI 1 "nonimmediate_operand" "%0"))
      (zero_extend:V8SI
(match_operand:V8HI 2 "nonimmediate_operand" "xm")))
    (const_int 16))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V8HImode, operands)"
  "pmulhuw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseimul")
   (set_attr "mode" "TI")])

(define_insn "sse2_umulv2siv2di3"
  [(set (match_operand:V2DI 0 "register_operand" "=x")
(mult:V2DI
  (zero_extend:V2DI
    (vec_select:V2SI
      (match_operand:V4SI 1 "nonimmediate_operand" "%0")
      (parallel [(const_int 0) (const_int 2)])))
  (zero_extend:V2DI
    (vec_select:V2SI
      (match_operand:V4SI 2 "nonimmediate_operand" "xm")
      (parallel [(const_int 0) (const_int 2)])))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V8HImode, operands)"
  "pmuludq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseimul")
   (set_attr "mode" "TI")])

(define_insn "sse2_pmaddwd"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(plus:V4SI
  (mult:V4SI
    (sign_extend:V4SI
      (vec_select:V4HI
(match_operand:V8HI 1 "nonimmediate_operand" "%0")
(parallel [(const_int 0)
   (const_int 2)
   (const_int 4)
   (const_int 6)])))
    (sign_extend:V4SI
      (vec_select:V4HI
(match_operand:V8HI 2 "nonimmediate_operand" "xm")
(parallel [(const_int 0)
   (const_int 2)
   (const_int 4)
   (const_int 6)]))))
  (mult:V4SI
    (sign_extend:V4SI
      (vec_select:V4HI (match_dup 1)
(parallel [(const_int 1)
```

```
   (const_int 3)
   (const_int 5)
   (const_int 7)])))
    (sign_extend:V4SI
      (vec_select:V4HI (match_dup 2)
(parallel [(const_int 1)
   (const_int 3)
   (const_int 5)
   (const_int 7)])))))]
  "TARGET_SSE2"
  "pmaddwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "mulv4si3"
  [(set (match_operand:V4SI 0 "register_operand" "")
(mult:V4SI (match_operand:V4SI 1 "register_operand" "")
   (match_operand:V4SI 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx t1, t2, t3, t4, t5, t6, thirtytwo;
  rtx op0, op1, op2;

  op0 = operands[0];
  op1 = operands[1];
  op2 = operands[2];
  t1 = gen_reg_rtx (V4SImode);
  t2 = gen_reg_rtx (V4SImode);
  t3 = gen_reg_rtx (V4SImode);
  t4 = gen_reg_rtx (V4SImode);
  t5 = gen_reg_rtx (V4SImode);
  t6 = gen_reg_rtx (V4SImode);
  thirtytwo = GEN_INT (32);

  /* Multiply elements 2 and 0.  */
  emit_insn (gen_sse2_umulv2siv2di3 (gen_lowpart (V2DImode, t1), op1, op2));

  /* Shift both input vectors down one element, so that elements 3 and 1
     are now in the slots for elements 2 and 0.  For K8, at least, this is
     faster than using a shuffle.  */
  emit_insn (gen_sse2_lshrti3 (gen_lowpart (TImode, t2),
       gen_lowpart (TImode, op1), thirtytwo));
  emit_insn (gen_sse2_lshrti3 (gen_lowpart (TImode, t3),
       gen_lowpart (TImode, op2), thirtytwo));

  /* Multiply elements 3 and 1.  */
  emit_insn (gen_sse2_umulv2siv2di3 (gen_lowpart (V2DImode, t4), t2, t3));

  /* Move the results in element 2 down to element 1; we don't care what
     goes in elements 2 and 3.  */
```

```
  emit_insn (gen_sse2_pshufd_1 (t5, t1, const0_rtx, const2_rtx,
const0_rtx, const0_rtx));
  emit_insn (gen_sse2_pshufd_1 (t6, t4, const0_rtx, const2_rtx,
const0_rtx, const0_rtx));

  /* Merge the parts back together.  */
  emit_insn (gen_sse2_punpckldq (op0, t5, t6));
  DONE;
})

(define_expand "mulv2di3"
  [(set (match_operand:V2DI 0 "register_operand" "")
(mult:V2DI (match_operand:V2DI 1 "register_operand" "")
   (match_operand:V2DI 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx t1, t2, t3, t4, t5, t6, thirtytwo;
  rtx op0, op1, op2;

  op0 = operands[0];
  op1 = operands[1];
  op2 = operands[2];
  t1 = gen_reg_rtx (V2DImode);
  t2 = gen_reg_rtx (V2DImode);
  t3 = gen_reg_rtx (V2DImode);
  t4 = gen_reg_rtx (V2DImode);
  t5 = gen_reg_rtx (V2DImode);
  t6 = gen_reg_rtx (V2DImode);
  thirtytwo = GEN_INT (32);

  /* Multiply low parts.  */
  emit_insn (gen_sse2_umulv2siv2di3 (t1, gen_lowpart (V4SImode, op1),
     gen_lowpart (V4SImode, op2)));

  /* Shift input vectors left 32 bits so we can multiply high parts.  */
  emit_insn (gen_lshrv2di3 (t2, op1, thirtytwo));
  emit_insn (gen_lshrv2di3 (t3, op2, thirtytwo));

  /* Multiply high parts by low parts.  */
  emit_insn (gen_sse2_umulv2siv2di3 (t4, gen_lowpart (V4SImode, op1),
     gen_lowpart (V4SImode, t3)));
  emit_insn (gen_sse2_umulv2siv2di3 (t5, gen_lowpart (V4SImode, op2),
     gen_lowpart (V4SImode, t2)));

  /* Shift them back.  */
  emit_insn (gen_ashlv2di3 (t4, t4, thirtytwo));
  emit_insn (gen_ashlv2di3 (t5, t5, thirtytwo));

  /* Add the three parts together.  */
  emit_insn (gen_addv2di3 (t6, t1, t4));
```

```
  emit_insn (gen_addv2di3 (op0, t6, t5));
  DONE;
})

(define_insn "ashr<mode>3"
  [(set (match_operand:SSEMODE24 0 "register_operand" "=x")
(ashiftrt:SSEMODE24
  (match_operand:SSEMODE24 1 "register_operand" "0")
  (match_operand:SI 2 "nonmemory_operand" "xi")))]
  "TARGET_SSE2"
  "psra<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseishft")
   (set_attr "mode" "TI")])

(define_insn "lshr<mode>3"
  [(set (match_operand:SSEMODE248 0 "register_operand" "=x")
(lshiftrt:SSEMODE248
  (match_operand:SSEMODE248 1 "register_operand" "0")
  (match_operand:SI 2 "nonmemory_operand" "xi")))]
  "TARGET_SSE2"
  "psrl<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseishft")
   (set_attr "mode" "TI")])

(define_insn "ashl<mode>3"
  [(set (match_operand:SSEMODE248 0 "register_operand" "=x")
(ashift:SSEMODE248
  (match_operand:SSEMODE248 1 "register_operand" "0")
  (match_operand:SI 2 "nonmemory_operand" "xi")))]
  "TARGET_SSE2"
  "psll<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseishft")
   (set_attr "mode" "TI")])

(define_insn "sse2_ashlti3"
  [(set (match_operand:TI 0 "register_operand" "=x")
(ashift:TI (match_operand:TI 1 "register_operand" "0")
   (match_operand:SI 2 "const_0_to_255_mul_8_operand" "n")))]
  "TARGET_SSE2"
{
  operands[2] = GEN_INT (INTVAL (operands[2]) / 8);
  return "pslldq\t{%2, %0|%0, %2}";
}
  [(set_attr "type" "sseishft")
   (set_attr "mode" "TI")])

(define_expand "vec_shl_<mode>"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
        (ashift:TI (match_operand:SSEMODEI 1 "register_operand" "")
   (match_operand:SI 2 "general_operand" "")))]
```

```
  "TARGET_SSE2"
{
  if (!const_0_to_255_mul_8_operand (operands[2], SImode))
    FAIL;
  operands[0] = gen_lowpart (TImode, operands[0]);
  operands[1] = gen_lowpart (TImode, operands[1]);
})

(define_insn "sse2_lshrti3"
  [(set (match_operand:TI 0 "register_operand" "=x")
  (lshiftrt:TI (match_operand:TI 1 "register_operand" "0")
     (match_operand:SI 2 "const_0_to_255_mul_8_operand" "n")))]
  "TARGET_SSE2"
{
  operands[2] = GEN_INT (INTVAL (operands[2]) / 8);
  return "psrldq\t{%2, %0|%0, %2}";
}
  [(set_attr "type" "sseishft")
   (set_attr "mode" "TI")])

(define_expand "vec_shr_<mode>"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
        (lshiftrt:TI (match_operand:SSEMODEI 1 "register_operand" "")
     (match_operand:SI 2 "general_operand" "")))]
  "TARGET_SSE2"
{
  if (!const_0_to_255_mul_8_operand (operands[2], SImode))
    FAIL;
  operands[0] = gen_lowpart (TImode, operands[0]);
  operands[1] = gen_lowpart (TImode, operands[1]);
})

(define_expand "umaxv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "")
(umax:V16QI (match_operand:V16QI 1 "nonimmediate_operand" "")
    (match_operand:V16QI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (UMAX, V16QImode, operands);")

(define_insn "*umaxv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(umax:V16QI (match_operand:V16QI 1 "nonimmediate_operand" "%0")
    (match_operand:V16QI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (UMAX, V16QImode, operands)"
  "pmaxub\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "smaxv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "")
```

```
(smax:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "")
   (match_operand:V8HI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (SMAX, V8HImode, operands);")

(define_insn "*smaxv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(smax:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "%0")
   (match_operand:V8HI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (SMAX, V8HImode, operands)"
  "pmaxsw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "umaxv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(us_minus:V8HI (match_operand:V8HI 1 "register_operand" "0")
       (match_operand:V8HI 2 "nonimmediate_operand" "xm")))
   (set (match_dup 3)
(plus:V8HI (match_dup 0) (match_dup 2)))]
  "TARGET_SSE2"
{
  operands[3] = operands[0];
  if (rtx_equal_p (operands[0], operands[2]))
    operands[0] = gen_reg_rtx (V8HImode);
})

(define_expand "smax<mode>3"
  [(set (match_operand:SSEMODE14 0 "register_operand" "")
(smax:SSEMODE14 (match_operand:SSEMODE14 1 "register_operand" "")
(match_operand:SSEMODE14 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx xops[6];
  bool ok;

  xops[0] = operands[0];
  xops[1] = operands[1];
  xops[2] = operands[2];
  xops[3] = gen_rtx_GT (VOIDmode, operands[1], operands[2]);
  xops[4] = operands[1];
  xops[5] = operands[2];
  ok = ix86_expand_int_vcond (xops);
  gcc_assert (ok);
  DONE;
})

(define_expand "umaxv4si3"
  [(set (match_operand:V4SI 0 "register_operand" "")
(umax:V4SI (match_operand:V4SI 1 "register_operand" "")
```

```
   (match_operand:V4SI 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx xops[6];
  bool ok;

  xops[0] = operands[0];
  xops[1] = operands[1];
  xops[2] = operands[2];
  xops[3] = gen_rtx_GTU (VOIDmode, operands[1], operands[2]);
  xops[4] = operands[1];
  xops[5] = operands[2];
  ok = ix86_expand_int_vcond (xops);
  gcc_assert (ok);
  DONE;
})

(define_expand "uminv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "")
(umin:V16QI (match_operand:V16QI 1 "nonimmediate_operand" "")
    (match_operand:V16QI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (UMAX, V16QImode, operands);")

(define_insn "*uminv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(umin:V16QI (match_operand:V16QI 1 "nonimmediate_operand" "%0")
    (match_operand:V16QI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (UMIN, V16QImode, operands)"
  "pminub\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "sminv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "")
(smin:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "")
   (match_operand:V8HI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (SMIN, V8HImode, operands);")

(define_insn "*sminv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(smin:V8HI (match_operand:V8HI 1 "nonimmediate_operand" "%0")
   (match_operand:V8HI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (SMIN, V8HImode, operands)"
  "pminsw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_expand "smin<mode>3"
```

```
  [(set (match_operand:SSEMODE14 0 "register_operand" "")
(smin:SSEMODE14 (match_operand:SSEMODE14 1 "register_operand" "")
(match_operand:SSEMODE14 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx xops[6];
  bool ok;

  xops[0] = operands[0];
  xops[1] = operands[2];
  xops[2] = operands[1];
  xops[3] = gen_rtx_GT (VOIDmode, operands[1], operands[2]);
  xops[4] = operands[1];
  xops[5] = operands[2];
  ok = ix86_expand_int_vcond (xops);
  gcc_assert (ok);
  DONE;
})

(define_expand "umin<mode>3"
  [(set (match_operand:SSEMODE24 0 "register_operand" "")
(umin:SSEMODE24 (match_operand:SSEMODE24 1 "register_operand" "")
(match_operand:SSEMODE24 2 "register_operand" "")))]
  "TARGET_SSE2"
{
  rtx xops[6];
  bool ok;

  xops[0] = operands[0];
  xops[1] = operands[2];
  xops[2] = operands[1];
  xops[3] = gen_rtx_GTU (VOIDmode, operands[1], operands[2]);
  xops[4] = operands[1];
  xops[5] = operands[2];
  ok = ix86_expand_int_vcond (xops);
  gcc_assert (ok);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral comparisons
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_eq<mode>3"
  [(set (match_operand:SSEMODE124 0 "register_operand" "=x")
(eq:SSEMODE124
  (match_operand:SSEMODE124 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODE124 2 "nonimmediate_operand" "xm")))]
```

```
  "TARGET_SSE2 && ix86_binary_operator_ok (EQ, <MODE>mode, operands)"
  "pcmpeq<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "TI")])

(define_insn "sse2_gt<mode>3"
  [(set (match_operand:SSEMODE124 0 "register_operand" "=x")
(gt:SSEMODE124
  (match_operand:SSEMODE124 1 "register_operand" "0")
  (match_operand:SSEMODE124 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "pcmpgt<ssevecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssecmp")
   (set_attr "mode" "TI")])

(define_expand "vcond<mode>"
  [(set (match_operand:SSEMODE124 0 "register_operand" "")
        (if_then_else:SSEMODE124
          (match_operator 3 ""
            [(match_operand:SSEMODE124 4 "nonimmediate_operand" "")
             (match_operand:SSEMODE124 5 "nonimmediate_operand" "")])
          (match_operand:SSEMODE124 1 "general_operand" "")
          (match_operand:SSEMODE124 2 "general_operand" "")))]
  "TARGET_SSE2"
{
  if (ix86_expand_int_vcond (operands))
    DONE;
  else
    FAIL;
})

(define_expand "vcondu<mode>"
  [(set (match_operand:SSEMODE12 0 "register_operand" "")
        (if_then_else:SSEMODE12
          (match_operator 3 ""
            [(match_operand:SSEMODE12 4 "nonimmediate_operand" "")
             (match_operand:SSEMODE12 5 "nonimmediate_operand" "")])
          (match_operand:SSEMODE12 1 "general_operand" "")
          (match_operand:SSEMODE12 2 "general_operand" "")))]
  "TARGET_SSE2"
{
  if (ix86_expand_int_vcond (operands))
    DONE;
  else
    FAIL;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral logical operations
```

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_expand "one_cmpl<mode>2"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(xor:SSEMODEI (match_operand:SSEMODEI 1 "nonimmediate_operand" "")
      (match_dup 2)))]
  "TARGET_SSE2"
{
  int i, n = GET_MODE_NUNITS (<MODE>mode);
  rtvec v = rtvec_alloc (n);

  for (i = 0; i < n; ++i)
    RTVEC_ELT (v, i) = constm1_rtx;

  operands[2] = force_reg (<MODE>mode, gen_rtx_CONST_VECTOR (<MODE>mode, v));
})

(define_expand "and<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(and:SSEMODEI (match_operand:SSEMODEI 1 "nonimmediate_operand" "")
      (match_operand:SSEMODEI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (AND, <MODE>mode, operands);")

(define_insn "*and<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(and:SSEMODEI
  (match_operand:SSEMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (AND, <MODE>mode, operands)"
  "pand\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_nand<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(and:SSEMODEI
  (not:SSEMODEI (match_operand:SSEMODEI 1 "register_operand" "0"))
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2"
  "pandn\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "ior<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(ior:SSEMODEI (match_operand:SSEMODEI 1 "nonimmediate_operand" "")
      (match_operand:SSEMODEI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
```

```
  "ix86_fixup_binary_operands_no_copy (IOR, <MODE>mode, operands);")

(define_insn "*ior<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(ior:SSEMODEI
  (match_operand:SSEMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (IOR, <MODE>mode, operands)"
  "por\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "xor<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "")
(xor:SSEMODEI (match_operand:SSEMODEI 1 "nonimmediate_operand" "")
      (match_operand:SSEMODEI 2 "nonimmediate_operand" "")))]
  "TARGET_SSE2"
  "ix86_fixup_binary_operands_no_copy (XOR, <MODE>mode, operands);")

(define_insn "*xor<mode>3"
  [(set (match_operand:SSEMODEI 0 "register_operand" "=x")
(xor:SSEMODEI
  (match_operand:SSEMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:SSEMODEI 2 "nonimmediate_operand" "xm")))]
  "TARGET_SSE2 && ix86_binary_operator_ok (XOR, <MODE>mode, operands)"
  "pxor\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral element swizzling
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_packsswb"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(vec_concat:V16QI
  (ss_truncate:V8QI
    (match_operand:V8HI 1 "register_operand" "0"))
  (ss_truncate:V8QI
    (match_operand:V8HI 2 "nonimmediate_operand" "xm"))))]
  "TARGET_SSE2"
  "packsswb\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_packssdw"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_concat:V8HI
```

```
  (ss_truncate:V4HI
    (match_operand:V4SI 1 "register_operand" "0"))
  (ss_truncate:V4HI
    (match_operand:V4SI 2 "nonimmediate_operand" "xm"))))]
  "TARGET_SSE2"
  "packssdw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_packuswb"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(vec_concat:V16QI
  (us_truncate:V8QI
    (match_operand:V8HI 1 "register_operand" "0"))
  (us_truncate:V8QI
    (match_operand:V8HI 2 "nonimmediate_operand" "xm"))))]
  "TARGET_SSE2"
  "packuswb\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpckhbw"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(vec_select:V16QI
  (vec_concat:V32QI
    (match_operand:V16QI 1 "register_operand" "0")
    (match_operand:V16QI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 8)  (const_int 24)
     (const_int 9)  (const_int 25)
     (const_int 10) (const_int 26)
     (const_int 11) (const_int 27)
     (const_int 12) (const_int 28)
     (const_int 13) (const_int 29)
     (const_int 14) (const_int 30)
     (const_int 15) (const_int 31)]))))]
  "TARGET_SSE2"
  "punpckhbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpcklbw"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(vec_select:V16QI
  (vec_concat:V32QI
    (match_operand:V16QI 1 "register_operand" "0")
    (match_operand:V16QI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0) (const_int 16)
     (const_int 1) (const_int 17)
     (const_int 2) (const_int 18)
     (const_int 3) (const_int 19)
```

```
     (const_int 4) (const_int 20)
     (const_int 5) (const_int 21)
     (const_int 6) (const_int 22)
     (const_int 7) (const_int 23)]))))]
  "TARGET_SSE2"
  "punpcklbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpckhwd"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_select:V8HI
  (vec_concat:V16HI
    (match_operand:V8HI 1 "register_operand" "0")
    (match_operand:V8HI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 4) (const_int 12)
     (const_int 5) (const_int 13)
     (const_int 6) (const_int 14)
     (const_int 7) (const_int 15)]))))]
  "TARGET_SSE2"
  "punpckhwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpcklwd"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_select:V8HI
  (vec_concat:V16HI
    (match_operand:V8HI 1 "register_operand" "0")
    (match_operand:V8HI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0) (const_int 8)
     (const_int 1) (const_int 9)
     (const_int 2) (const_int 10)
     (const_int 3) (const_int 11)]))))]
  "TARGET_SSE2"
  "punpcklwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpckhdq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(vec_select:V4SI
  (vec_concat:V8SI
    (match_operand:V4SI 1 "register_operand" "0")
    (match_operand:V4SI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 2) (const_int 6)
     (const_int 3) (const_int 7)]))))]
  "TARGET_SSE2"
  "punpckhdq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
```

```
   (set_attr "mode" "TI")])

(define_insn "sse2_punpckldq"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(vec_select:V4SI
  (vec_concat:V8SI
    (match_operand:V4SI 1 "register_operand" "0")
    (match_operand:V4SI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0) (const_int 4)
     (const_int 1) (const_int 5)])))]
  "TARGET_SSE2"
  "punpckldq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpckhqdq"
  [(set (match_operand:V2DI 0 "register_operand" "=x")
(vec_select:V2DI
  (vec_concat:V4DI
    (match_operand:V2DI 1 "register_operand" "0")
    (match_operand:V2DI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 1)
     (const_int 3)])))]
  "TARGET_SSE2"
  "punpckhqdq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_punpcklqdq"
  [(set (match_operand:V2DI 0 "register_operand" "=x")
(vec_select:V2DI
  (vec_concat:V4DI
    (match_operand:V2DI 1 "register_operand" "0")
    (match_operand:V2DI 2 "nonimmediate_operand" "xm"))
  (parallel [(const_int 0)
     (const_int 2)])))]
  "TARGET_SSE2"
  "punpcklqdq\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "sse2_pinsrw"
  [(set (match_operand:V8HI 0 "register_operand" "")
(vec_merge:V8HI
  (vec_duplicate:V8HI
    (match_operand:SI 2 "nonimmediate_operand" ""))
  (match_operand:V8HI 1 "register_operand" "")
  (match_operand:SI 3 "const_0_to_7_operand" "")))]
  "TARGET_SSE2"
{
```

```
  operands[2] = gen_lowpart (HImode, operands[2]);
  operands[3] = GEN_INT ((1 << INTVAL (operands[3])));
})

(define_insn "*sse2_pinsrw"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_merge:V8HI
  (vec_duplicate:V8HI
    (match_operand:HI 2 "nonimmediate_operand" "rm"))
  (match_operand:V8HI 1 "register_operand" "0")
  (match_operand:SI 3 "const_pow2_1_to_128_operand" "n")))]
  "TARGET_SSE2"
{
  operands[3] = GEN_INT (exact_log2 (INTVAL (operands[3])));
  return "pinsrw\t{%3, %k2, %0|%0, %k2, %3}";
}
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_insn "sse2_pextrw"
  [(set (match_operand:SI 0 "register_operand" "=r")
(zero_extend:SI
  (vec_select:HI
    (match_operand:V8HI 1 "register_operand" "x")
    (parallel [(match_operand:SI 2 "const_0_to_7_operand" "n")])))]
  "TARGET_SSE2"
  "pextrw\t{%2, %1, %0|%0, %1, %2}"
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "sse2_pshufd"
  [(match_operand:V4SI 0 "register_operand" "")
   (match_operand:V4SI 1 "nonimmediate_operand" "")
   (match_operand:SI 2 "const_int_operand" "")]
  "TARGET_SSE2"
{
  int mask = INTVAL (operands[2]);
  emit_insn (gen_sse2_pshufd_1 (operands[0], operands[1],
GEN_INT ((mask >> 0) & 3),
GEN_INT ((mask >> 2) & 3),
GEN_INT ((mask >> 4) & 3),
GEN_INT ((mask >> 6) & 3)));
  DONE;
})

(define_insn "sse2_pshufd_1"
  [(set (match_operand:V4SI 0 "register_operand" "=x")
(vec_select:V4SI
  (match_operand:V4SI 1 "nonimmediate_operand" "xm")
  (parallel [(match_operand 2 "const_0_to_3_operand" "")
```

```
     (match_operand 3 "const_0_to_3_operand" "")
     (match_operand 4 "const_0_to_3_operand" "")
     (match_operand 5 "const_0_to_3_operand" "")]))))]
  "TARGET_SSE2"
{
  int mask = 0;
  mask |= INTVAL (operands[2]) << 0;
  mask |= INTVAL (operands[3]) << 2;
  mask |= INTVAL (operands[4]) << 4;
  mask |= INTVAL (operands[5]) << 6;
  operands[2] = GEN_INT (mask);

  return "pshufd\t{%2, %1, %0|%0, %1, %2}";
}
  [(set_attr "type" "sselog1")
   (set_attr "mode" "TI")])

(define_expand "sse2_pshuflw"
  [(match_operand:V8HI 0 "register_operand" "")
   (match_operand:V8HI 1 "nonimmediate_operand" "")
   (match_operand:SI 2 "const_int_operand" "")]
  "TARGET_SSE2"
{
  int mask = INTVAL (operands[2]);
  emit_insn (gen_sse2_pshuflw_1 (operands[0], operands[1],
 GEN_INT ((mask >> 0) & 3),
 GEN_INT ((mask >> 2) & 3),
 GEN_INT ((mask >> 4) & 3),
 GEN_INT ((mask >> 6) & 3)));
  DONE;
})

(define_insn "sse2_pshuflw_1"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_select:V8HI
  (match_operand:V8HI 1 "nonimmediate_operand" "xm")
  (parallel [(match_operand 2 "const_0_to_3_operand" "")
     (match_operand 3 "const_0_to_3_operand" "")
     (match_operand 4 "const_0_to_3_operand" "")
     (match_operand 5 "const_0_to_3_operand" "")
     (const_int 4)
     (const_int 5)
     (const_int 6)
     (const_int 7)]))))]
  "TARGET_SSE2"
{
  int mask = 0;
  mask |= INTVAL (operands[2]) << 0;
  mask |= INTVAL (operands[3]) << 2;
  mask |= INTVAL (operands[4]) << 4;
```

```
  mask |= INTVAL (operands[5]) << 6;
  operands[2] = GEN_INT (mask);

  return "pshuflw\t{%2, %1, %0|%0, %1, %2}";
}
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "sse2_pshufhw"
  [(match_operand:V8HI 0 "register_operand" "")
   (match_operand:V8HI 1 "nonimmediate_operand" "")
   (match_operand:SI 2 "const_int_operand" "")]
  "TARGET_SSE2"
{
  int mask = INTVAL (operands[2]);
  emit_insn (gen_sse2_pshufhw_1 (operands[0], operands[1],
 GEN_INT (((mask >> 0) & 3) + 4),
 GEN_INT (((mask >> 2) & 3) + 4),
 GEN_INT (((mask >> 4) & 3) + 4),
 GEN_INT (((mask >> 6) & 3) + 4)));
  DONE;
})

(define_insn "sse2_pshufhw_1"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(vec_select:V8HI
  (match_operand:V8HI 1 "nonimmediate_operand" "xm")
  (parallel [(const_int 0)
     (const_int 1)
     (const_int 2)
     (const_int 3)
     (match_operand 2 "const_4_to_7_operand" "")
     (match_operand 3 "const_4_to_7_operand" "")
     (match_operand 4 "const_4_to_7_operand" "")
     (match_operand 5 "const_4_to_7_operand" "")]))))]
  "TARGET_SSE2"
{
  int mask = 0;
  mask |= (INTVAL (operands[2]) - 4) << 0;
  mask |= (INTVAL (operands[3]) - 4) << 2;
  mask |= (INTVAL (operands[4]) - 4) << 4;
  mask |= (INTVAL (operands[5]) - 4) << 6;
  operands[2] = GEN_INT (mask);

  return "pshufhw\t{%2, %1, %0|%0, %1, %2}";
}
  [(set_attr "type" "sselog")
   (set_attr "mode" "TI")])

(define_expand "sse2_loadd"
```

```
  [(set (match_operand:V4SI 0 "register_operand" "")
(vec_merge:V4SI
  (vec_duplicate:V4SI
    (match_operand:SI 1 "nonimmediate_operand" ""))
  (match_dup 2)
  (const_int 1)))]
  "TARGET_SSE"
  "operands[2] = CONST0_RTX (V4SImode);")

(define_insn "sse2_loadld"
  [(set (match_operand:V4SI 0 "register_operand"       "=Y,x,x")
(vec_merge:V4SI
  (vec_duplicate:V4SI
    (match_operand:SI 2 "nonimmediate_operand" "mr,m,x"))
  (match_operand:V4SI 1 "vector_move_operand"  " C,C,0")
  (const_int 1)))]
  "TARGET_SSE"
  "@
   movd\t{%2, %0|%0, %2}
   movss\t{%2, %0|%0, %2}
   movss\t{%2, %0|%0, %2}"
  [(set_attr "type" "ssemov")
   (set_attr "mode" "TI,V4SF,SF")])

;; ??? The hardware supports more, but TARGET_INTER_UNIT_MOVES must
;; be taken into account, and movdi isn't fully populated even without.
(define_insn_and_split "sse2_stored"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=mx")
(vec_select:SI
  (match_operand:V4SI 1 "register_operand" "x")
  (parallel [(const_int 0)])))]
  "TARGET_SSE"
  "#"
  "&& reload_completed"
  [(set (match_dup 0) (match_dup 1))]
{
  operands[1] = gen_rtx_REG (SImode, REGNO (operands[1]));
})

(define_expand "sse_storeq"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(vec_select:DI
  (match_operand:V2DI 1 "register_operand" "")
  (parallel [(const_int 0)])))]
  "TARGET_SSE"
  "")

;; ??? The hardware supports more, but TARGET_INTER_UNIT_MOVES must
;; be taken into account, and movdi isn't fully populated even without.
(define_insn "*sse2_storeq"
```

```
  [(set (match_operand:DI 0 "nonimmediate_operand" "=mx")
(vec_select:DI
  (match_operand:V2DI 1 "register_operand" "x")
  (parallel [(const_int 0)])))]
  "TARGET_SSE"
  "#")

(define_split
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
(vec_select:DI
  (match_operand:V2DI 1 "register_operand" "")
  (parallel [(const_int 0)])))]
  "TARGET_SSE && reload_completed"
  [(set (match_dup 0) (match_dup 1))]
{
  operands[1] = gen_rtx_REG (DImode, REGNO (operands[1]));
})

(define_insn "*vec_dupv4si"
  [(set (match_operand:V4SI 0 "register_operand" "=Y,x")
(vec_duplicate:V4SI
  (match_operand:SI 1 "register_operand" " Y,0")))]
  "TARGET_SSE"
  "@
   pshufd\t{$0, %1, %0|%0, %1, 0}
   shufps\t{$0, %0, %0|%0, %0, 0}"
  [(set_attr "type" "sselog1")
   (set_attr "mode" "TI,V4SF")])

(define_insn "*vec_dupv2di"
  [(set (match_operand:V2DI 0 "register_operand" "=Y,x")
(vec_duplicate:V2DI
  (match_operand:DI 1 "register_operand" " 0,0")))]
  "TARGET_SSE"
  "@
   punpcklqdq\t%0, %0
   movlhps\t%0, %0"
  [(set_attr "type" "sselog1,ssemov")
   (set_attr "mode" "TI,V4SF")])

;; ??? In theory we can match memory for the MMX alternative, but allowing
;; nonimmediate_operand for operand 2 and *not* allowing memory for the SSE
;; alternatives pretty much forces the MMX alternative to be chosen.
(define_insn "*sse2_concatv2si"
  [(set (match_operand:V2SI 0 "register_operand"     "=Y, Y,*y,*y")
(vec_concat:V2SI
  (match_operand:SI 1 "nonimmediate_operand" " 0,rm, 0,rm")
  (match_operand:SI 2 "reg_or_0_operand"     " Y, C,*y, C")))]
  "TARGET_SSE2"
  "@
```

```
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog,ssemov,mmxcvt,mmxmov")
   (set_attr "mode" "TI,TI,DI,DI")])

(define_insn "*sse1_concatv2si"
  [(set (match_operand:V2SI 0 "register_operand"     "=x,x,*y,*y")
(vec_concat:V2SI
  (match_operand:SI 1 "nonimmediate_operand" " 0,m, 0,*rm")
  (match_operand:SI 2 "reg_or_0_operand"     " x,C,*y,C")))]
  "TARGET_SSE"
  "@
   unpcklps\t{%2, %0|%0, %2}
   movss\t{%1, %0|%0, %1}
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "sselog,ssemov,mmxcvt,mmxmov")
   (set_attr "mode" "V4SF,V4SF,DI,DI")])

(define_insn "*vec_concatv4si_1"
  [(set (match_operand:V4SI 0 "register_operand"       "=Y,x,x")
(vec_concat:V4SI
  (match_operand:V2SI 1 "register_operand"     " 0,0,0")
  (match_operand:V2SI 2 "nonimmediate_operand" " Y,x,m")))]
  "TARGET_SSE"
  "@
   punpcklqdq\t{%2, %0|%0, %2}
   movlhps\t{%2, %0|%0, %2}
   movhps\t{%2, %0|%0, %2}"
  [(set_attr "type" "sselog,ssemov,ssemov")
   (set_attr "mode" "TI,V4SF,V2SF")])

(define_insn "*vec_concatv2di"
  [(set (match_operand:V2DI 0 "register_operand"     "=Y,?Y,Y,x,x,x")
(vec_concat:V2DI
  (match_operand:DI 1 "nonimmediate_operand" " m,*y,0,0,0,m")
  (match_operand:DI 2 "vector_move_operand"  " C, C,Y,x,m,0")))]
  "TARGET_SSE"
  "@
   movq\t{%1, %0|%0, %1}
   movq2dq\t{%1, %0|%0, %1}
   punpcklqdq\t{%2, %0|%0, %2}
   movlhps\t{%2, %0|%0, %2}
   movhps\t{%2, %0|%0, %2}
   movlps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssemov,ssemov,sselog,ssemov,ssemov,ssemov")
   (set_attr "mode" "TI,TI,TI,V4SF,V2SF,V2SF")])
```

```
(define_expand "vec_setv2di"
  [(match_operand:V2DI 0 "register_operand" "")
   (match_operand:DI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv2di"
  [(match_operand:DI 0 "register_operand" "")
   (match_operand:V2DI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv2di"
  [(match_operand:V2DI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

(define_expand "vec_setv4si"
  [(match_operand:V4SI 0 "register_operand" "")
   (match_operand:SI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv4si"
  [(match_operand:SI 0 "register_operand" "")
   (match_operand:V4SI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
```

```
  DONE;
})

(define_expand "vec_initv4si"
  [(match_operand:V4SI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

(define_expand "vec_setv8hi"
  [(match_operand:V8HI 0 "register_operand" "")
   (match_operand:HI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv8hi"
  [(match_operand:HI 0 "register_operand" "")
   (match_operand:V8HI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv8hi"
  [(match_operand:V8HI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

(define_expand "vec_setv16qi"
  [(match_operand:V16QI 0 "register_operand" "")
   (match_operand:QI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
```

```
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv16qi"
  [(match_operand:QI 0 "register_operand" "")
   (match_operand:V16QI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv16qi"
  [(match_operand:V16QI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Miscellaneous
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "sse2_uavgv16qi3"
  [(set (match_operand:V16QI 0 "register_operand" "=x")
(truncate:V16QI
  (lshiftrt:V16HI
    (plus:V16HI
      (plus:V16HI
(zero_extend:V16HI
  (match_operand:V16QI 1 "nonimmediate_operand" "%0"))
(zero_extend:V16HI
  (match_operand:V16QI 2 "nonimmediate_operand" "xm")))
      (const_vector:V16QI [(const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)
   (const_int 1) (const_int 1)]))
    (const_int 1))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (PLUS, V16QImode, operands)"
```

```
  "pavgb\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_insn "sse2_uavgv8hi3"
  [(set (match_operand:V8HI 0 "register_operand" "=x")
(truncate:V8HI
  (lshiftrt:V8SI
    (plus:V8SI
      (plus:V8SI
(zero_extend:V8SI
  (match_operand:V8HI 1 "nonimmediate_operand" "%0"))
(zero_extend:V8SI
  (match_operand:V8HI 2 "nonimmediate_operand" "xm")))
      (const_vector:V8HI [(const_int 1) (const_int 1)
  (const_int 1) (const_int 1)
  (const_int 1) (const_int 1)
  (const_int 1) (const_int 1)]))
    (const_int 1))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (PLUS, V8HImode, operands)"
  "pavgw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

;; The correct representation for this is absolutely enormous, and
;; surely not generally useful.
(define_insn "sse2_psadbw"
  [(set (match_operand:V2DI 0 "register_operand" "=x")
(unspec:V2DI [(match_operand:V16QI 1 "register_operand" "0")
      (match_operand:V16QI 2 "nonimmediate_operand" "xm")]
     UNSPEC_PSADBW))]
  "TARGET_SSE2"
  "psadbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "sseiadd")
   (set_attr "mode" "TI")])

(define_insn "sse_movmskps"
  [(set (match_operand:SI 0 "register_operand" "=r")
(unspec:SI [(match_operand:V4SF 1 "register_operand" "x")]
   UNSPEC_MOVMSK))]
  "TARGET_SSE"
  "movmskps\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V4SF")])

(define_insn "sse2_movmskpd"
  [(set (match_operand:SI 0 "register_operand" "=r")
(unspec:SI [(match_operand:V2DF 1 "register_operand" "x")]
   UNSPEC_MOVMSK))]
  "TARGET_SSE2"
```

```
  "movmskpd\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_insn "sse2_pmovmskb"
  [(set (match_operand:SI 0 "register_operand" "=r")
(unspec:SI [(match_operand:V16QI 1 "register_operand" "x")]
   UNSPEC_MOVMSK))]
  "TARGET_SSE2"
  "pmovmskb\t{%1, %0|%0, %1}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "V2DF")])

(define_expand "sse2_maskmovdqu"
  [(set (match_operand:V16QI 0 "memory_operand" "")
(unspec:V16QI [(match_operand:V16QI 1 "register_operand" "x")
       (match_operand:V16QI 2 "register_operand" "x")
       (match_dup 0)]
      UNSPEC_MASKMOV))]
  "TARGET_SSE2"
  "")

(define_insn "*sse2_maskmovdqu"
  [(set (mem:V16QI (match_operand:SI 0 "register_operand" "D"))
(unspec:V16QI [(match_operand:V16QI 1 "register_operand" "x")
       (match_operand:V16QI 2 "register_operand" "x")
       (mem:V16QI (match_dup 0))]
      UNSPEC_MASKMOV))]
  "TARGET_SSE2 && !TARGET_64BIT"
  ;; @@@ check ordering of operands in intel/nonintel syntax
  "maskmovdqu\t{%2, %1|%1, %2}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

(define_insn "*sse2_maskmovdqu_rex64"
  [(set (mem:V16QI (match_operand:DI 0 "register_operand" "D"))
(unspec:V16QI [(match_operand:V16QI 1 "register_operand" "x")
       (match_operand:V16QI 2 "register_operand" "x")
       (mem:V16QI (match_dup 0))]
      UNSPEC_MASKMOV))]
  "TARGET_SSE2 && TARGET_64BIT"
  ;; @@@ check ordering of operands in intel/nonintel syntax
  "maskmovdqu\t{%2, %1|%1, %2}"
  [(set_attr "type" "ssecvt")
   (set_attr "mode" "TI")])

(define_insn "sse_ldmxcsr"
  [(unspec_volatile [(match_operand:SI 0 "memory_operand" "m")]
    UNSPECV_LDMXCSR)]
  "TARGET_SSE"
```

```
  "ldmxcsr\t%0"
  [(set_attr "type" "sse")
   (set_attr "memory" "load")])

(define_insn "sse_stmxcsr"
  [(set (match_operand:SI 0 "memory_operand" "=m")
(unspec_volatile:SI [(const_int 0)] UNSPECV_STMXCSR))]
  "TARGET_SSE"
  "stmxcsr\t%0"
  [(set_attr "type" "sse")
   (set_attr "memory" "store")])

(define_expand "sse_sfence"
  [(set (match_dup 0)
(unspec:BLK [(match_dup 0)] UNSPEC_SFENCE))]
  "TARGET_SSE || TARGET_3DNOW_A"
{
  operands[0] = gen_rtx_MEM (BLKmode, gen_rtx_SCRATCH (Pmode));
  MEM_VOLATILE_P (operands[0]) = 1;
})

(define_insn "*sse_sfence"
  [(set (match_operand:BLK 0 "" "")
(unspec:BLK [(match_dup 0)] UNSPEC_SFENCE))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "sfence"
  [(set_attr "type" "sse")
   (set_attr "memory" "unknown")])

(define_insn "sse2_clflush"
  [(unspec_volatile [(match_operand 0 "address_operand" "p")]
    UNSPECV_CLFLUSH)]
  "TARGET_SSE2"
  "clflush\t%a0"
  [(set_attr "type" "sse")
   (set_attr "memory" "unknown")])

(define_expand "sse2_mfence"
  [(set (match_dup 0)
(unspec:BLK [(match_dup 0)] UNSPEC_MFENCE))]
  "TARGET_SSE2"
{
  operands[0] = gen_rtx_MEM (BLKmode, gen_rtx_SCRATCH (Pmode));
  MEM_VOLATILE_P (operands[0]) = 1;
})

(define_insn "*sse2_mfence"
  [(set (match_operand:BLK 0 "" "")
(unspec:BLK [(match_dup 0)] UNSPEC_MFENCE))]
  "TARGET_SSE2"
```

```
  "mfence"
  [(set_attr "type" "sse")
   (set_attr "memory" "unknown")])

(define_expand "sse2_lfence"
  [(set (match_dup 0)
(unspec:BLK [(match_dup 0)] UNSPEC_LFENCE))]
  "TARGET_SSE2"
{
  operands[0] = gen_rtx_MEM (BLKmode, gen_rtx_SCRATCH (Pmode));
  MEM_VOLATILE_P (operands[0]) = 1;
})

(define_insn "*sse2_lfence"
  [(set (match_operand:BLK 0 "" "")
(unspec:BLK [(match_dup 0)] UNSPEC_LFENCE))]
  "TARGET_SSE2"
  "lfence"
  [(set_attr "type" "sse")
   (set_attr "memory" "unknown")])

(define_insn "sse3_mwait"
  [(unspec_volatile [(match_operand:SI 0 "register_operand" "a")
     (match_operand:SI 1 "register_operand" "c")]
    UNSPECV_MWAIT)]
  "TARGET_SSE3"
  "mwait\t%0, %1"
  [(set_attr "length" "3")])

(define_insn "sse3_monitor"
  [(unspec_volatile [(match_operand:SI 0 "register_operand" "a")
     (match_operand:SI 1 "register_operand" "c")
     (match_operand:SI 2 "register_operand" "d")]
    UNSPECV_MONITOR)]
  "TARGET_SSE3"
  "monitor\t%0, %1, %2"
  [(set_attr "length" "3")])
```

1.9 GCC machine description for MMX and 3dNOW! instructions

;;(include "mmx.md")

```
;; The MMX and 3dNOW! patterns are in the same file because they use
;; the same register file, and 3dNOW! adds a number of extensions to
;; the base integer MMX isa.

;; Note!  Except for the basic move instructions, *all* of these
;; patterns are outside the normal optabs namespace.  This is because
;; use of these registers requires the insertion of emms or femms
```

```
;; instructions to return to normal fpu mode.  The compiler doesn't
;; know how to do that itself, which means it's up to the user.  Which
;; means that we should never use any of these patterns except at the
;; direction of the user via a builtin.

;; 8 byte integral modes handled by MMX (and by extension, SSE)
(define_mode_macro MMXMODEI [V8QI V4HI V2SI])

;; All 8-byte vector modes handled by MMX
(define_mode_macro MMXMODE [V8QI V4HI V2SI V2SF])

;; Mix-n-match
(define_mode_macro MMXMODE12 [V8QI V4HI])
(define_mode_macro MMXMODE24 [V4HI V2SI])

;; Mapping from integer vector mode to mnemonic suffix
(define_mode_attr mmxvecsize [(V8QI "b") (V4HI "w") (V2SI "d") (DI "q")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Move patterns
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; All of these patterns are enabled for MMX as well as 3dNOW.
;; This is essential for maintaining stable calling conventions.

(define_expand "mov<mode>"
  [(set (match_operand:MMXMODEI 0 "nonimmediate_operand" "")
(match_operand:MMXMODEI 1 "nonimmediate_operand" ""))]
  "TARGET_MMX"
{
  ix86_expand_vector_move (<MODE>mode, operands);
  DONE;
})

(define_insn "*mov<mode>_internal_rex64"
  [(set (match_operand:MMXMODEI 0 "nonimmediate_operand"
"=rm,r,*y,*y ,m ,*y,Y ,x,x ,m,r,x")
(match_operand:MMXMODEI 1 "vector_move_operand"
"Cr ,m,C ,*ym,*y,Y ,*y,C,xm,x,x,r"))]
  "TARGET_64BIT && TARGET_MMX
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    movdq2q\t{%1, %0|%0, %1}
```

```
    movq2dq\t{%1, %0|%0, %1}
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    movd\t{%1, %0|%0, %1}
    movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "imov,imov,mmxmov,mmxmov,mmxmov,ssecvt,ssecvt,ssemov,ssemov,ssemov,ssemov,ssemov")
   (set_attr "unit" "*,*,*,*,*,mmx,mmx,*,*,*,*,*")
   (set_attr "mode" "DI")])

(define_insn "*mov<mode>_internal"
  [(set (match_operand:MMXMODEI 0 "nonimmediate_operand"
"=*y,*y ,m ,*y,*Y,*Y,*Y ,m ,*x,*x,*x,m ,?r ,?m")
(match_operand:MMXMODEI 1 "vector_move_operand"
"C  ,*ym,*y,*Y,*y,C ,*Ym,*Y,C ,*x,m ,*x,irm,r"))]
  "TARGET_MMX
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    movdq2q\t{%1, %0|%0, %1}
    movq2dq\t{%1, %0|%0, %1}
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    xorps\t%0, %0
    movaps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    #
    #"
  [(set_attr "type" "mmxmov,mmxmov,mmxmov,ssecvt,ssecvt,ssemov,ssemov,ssemov,ssemov,ssemov,ssemov,ssem
   (set_attr "unit" "*,*,*,mmx,mmx,*,*,*,*,*,*,*,*")
   (set_attr "mode" "DI,DI,DI,DI,DI,TI,DI,DI,V4SF,V4SF,V2SF,V2SF,DI,DI")])

(define_expand "movv2sf"
  [(set (match_operand:V2SF 0 "nonimmediate_operand" "")
(match_operand:V2SF 1 "nonimmediate_operand" ""))]
  "TARGET_MMX"
{
  ix86_expand_vector_move (V2SFmode, operands);
  DONE;
})

(define_insn "*movv2sf_internal_rex64"
  [(set (match_operand:V2SF 0 "nonimmediate_operand"
"=rm,r,*y ,*y ,m ,*y,Y ,x,x,x,m,r,x")
        (match_operand:V2SF 1 "vector_move_operand"
"Cr ,m ,C ,*ym,*y,Y ,*y,C,x,m,x,x,r"))]
```

```
  "TARGET_64BIT && TARGET_MMX
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    movdq2q\t{%1, %0|%0, %1}
    movq2dq\t{%1, %0|%0, %1}
    xorps\t%0, %0
    movaps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    movd\t{%1, %0|%0, %1}
    movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "imov,imov,mmxmov,mmxmov,mmxmov,ssecvt,ssecvt,ssemov,ssemov,ssemov,ssemov,ssemov,s
   (set_attr "unit" "*,*,*,*,*,mmx,mmx,*,*,*,*,*,*")
   (set_attr "mode" "DI,DI,DI,DI,DI,DI,DI,V4SF,V4SF,V2SF,V2SF,DI,DI")])

(define_insn "*movv2sf_internal"
  [(set (match_operand:V2SF 0 "nonimmediate_operand"
"=*y,*y ,m,*y,*Y,*x,*x,*x,m ,?r ,?m")
        (match_operand:V2SF 1 "vector_move_operand"
"C ,*ym,*y,*Y,*y,C ,*x,m ,*x,irm,r"))]
  "TARGET_MMX
   && (GET_CODE (operands[0]) != MEM || GET_CODE (operands[1]) != MEM)"
  "@
    pxor\t%0, %0
    movq\t{%1, %0|%0, %1}
    movq\t{%1, %0|%0, %1}
    movdq2q\t{%1, %0|%0, %1}
    movq2dq\t{%1, %0|%0, %1}
    xorps\t%0, %0
    movaps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    movlps\t{%1, %0|%0, %1}
    #
    #"
  [(set_attr "type" "mmxmov,mmxmov,mmxmov,ssecvt,ssecvt,ssemov,ssemov,ssemov,ssemov,*,*")
   (set_attr "unit" "*,*,*,mmx,mmx,*,*,*,*,*,*")
   (set_attr "mode" "DI,DI,DI,DI,DI,V4SF,V4SF,V2SF,V2SF,DI,DI")])

;; %%% This multiword shite has got to go.
(define_split
  [(set (match_operand:MMXMODE 0 "nonimmediate_operand" "")
        (match_operand:MMXMODE 1 "general_operand" ""))]
  "!TARGET_64BIT && reload_completed
   && (!MMX_REG_P (operands[0]) && !SSE_REG_P (operands[0]))
   && (!MMX_REG_P (operands[1]) && !SSE_REG_P (operands[1]))"
```

```
  [(const_int 0)]
  "ix86_split_long_move (operands); DONE;")

(define_expand "push<mode>1"
  [(match_operand:MMXMODE 0 "register_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_push (<MODE>mode, operands[0]);
  DONE;
})

(define_expand "movmisalign<mode>"
  [(set (match_operand:MMXMODE 0 "nonimmediate_operand" "")
(match_operand:MMXMODE 1 "nonimmediate_operand" ""))]
  "TARGET_MMX"
{
  ix86_expand_vector_move (<MODE>mode, operands);
  DONE;
})

(define_insn "sse_movntdi"
  [(set (match_operand:DI 0 "memory_operand" "=m")
(unspec:DI [(match_operand:DI 1 "register_operand" "y")]
   UNSPEC_MOVNT))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "movntq\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxmov")
   (set_attr "mode" "DI")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point arithmetic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_addv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(plus:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "%0")
   (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW && ix86_binary_operator_ok (PLUS, V2SFmode, operands)"
  "pfadd\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_subv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y,y")
        (minus:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "0,ym")
    (match_operand:V2SF 2 "nonimmediate_operand" "ym,0")))]
  "TARGET_3DNOW && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
```

```
   pfsub\\t{%2, %0|%0, %2}
   pfsubr\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_expand "mmx_subrv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "")
        (minus:V2SF (match_operand:V2SF 2 "nonimmediate_operand" "")
    (match_operand:V2SF 1 "nonimmediate_operand" "")))]
  "TARGET_3DNOW && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "")

(define_insn "mmx_mulv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(mult:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "%0")
   (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW && ix86_binary_operator_ok (MULT, V2SFmode, operands)"
  "pfmul\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_smaxv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
        (smax:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "%0")
                   (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW && ix86_binary_operator_ok (SMAX, V2SFmode, operands)"
  "pfmax\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_sminv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
        (smin:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "%0")
                   (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW && ix86_binary_operator_ok (SMIN, V2SFmode, operands)"
  "pfmin\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_rcpv2sf2"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
        (unspec:V2SF [(match_operand:V2SF 1 "nonimmediate_operand" "ym")]
     UNSPEC_PFRCP))]
  "TARGET_3DNOW"
  "pfrcp\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmx")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_rcpit1v2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
```

```
(unspec:V2SF [(match_operand:V2SF 1 "register_operand" "0")
      (match_operand:V2SF 2 "nonimmediate_operand" "ym")]
     UNSPEC_PFRCPIT1))]
  "TARGET_3DNOW"
  "pfrcpit1\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmx")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_rcpit2v2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(unspec:V2SF [(match_operand:V2SF 1 "register_operand" "0")
      (match_operand:V2SF 2 "nonimmediate_operand" "ym")]
     UNSPEC_PFRCPIT2))]
  "TARGET_3DNOW"
  "pfrcpit2\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmx")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_rsqrtv2sf2"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(unspec:V2SF [(match_operand:V2SF 1 "nonimmediate_operand" "ym")]
     UNSPEC_PFRSQRT))]
  "TARGET_3DNOW"
  "pfrsqrt\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmx")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_rsqit1v2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(unspec:V2SF [(match_operand:V2SF 1 "register_operand" "0")
      (match_operand:V2SF 2 "nonimmediate_operand" "ym")]
     UNSPEC_PFRSQIT1))]
  "TARGET_3DNOW"
  "pfrsqit1\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmx")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_haddv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(vec_concat:V2SF
  (plus:SF
    (vec_select:SF
      (match_operand:V2SF 1 "register_operand" "0")
      (parallel [(const_int  0)]))
    (vec_select:SF (match_dup 1) (parallel [(const_int 1)])))
  (plus:SF
            (vec_select:SF
      (match_operand:V2SF 2 "nonimmediate_operand" "ym")
      (parallel [(const_int  0)]))
    (vec_select:SF (match_dup 2) (parallel [(const_int 1)])))))]
```

```
  "TARGET_3DNOW"
  "pfacc\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_hsubv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(vec_concat:V2SF
  (minus:SF
    (vec_select:SF
      (match_operand:V2SF 1 "register_operand" "0")
      (parallel [(const_int  0)]))
    (vec_select:SF (match_dup 1) (parallel [(const_int 1)])))
  (minus:SF
           (vec_select:SF
      (match_operand:V2SF 2 "nonimmediate_operand" "ym")
      (parallel [(const_int  0)]))
    (vec_select:SF (match_dup 2) (parallel [(const_int 1)])))))]
  "TARGET_3DNOW_A"
  "pfnacc\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_addsubv2sf3"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
        (vec_merge:V2SF
          (plus:V2SF
            (match_operand:V2SF 1 "register_operand" "0")
            (match_operand:V2SF 2 "nonimmediate_operand" "ym"))
          (minus:V2SF (match_dup 1) (match_dup 2))
          (const_int 1)))]
  "TARGET_3DNOW_A"
  "pfpnacc\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "V2SF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point comparisons
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_gtv2sf3"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(gt:V2SI (match_operand:V2SF 1 "register_operand" "0")
 (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW"
  "pfcmpgt\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcmp")
   (set_attr "mode" "V2SF")])
```

```
(define_insn "mmx_gev2sf3"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(ge:V2SI (match_operand:V2SF 1 "register_operand" "0")
 (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW"
  "pfcmpge\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcmp")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_eqv2sf3"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(eq:V2SI (match_operand:V2SF 1 "nonimmediate_operand" "%0")
 (match_operand:V2SF 2 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW && ix86_binary_operator_ok (EQ, V2SFmode, operands)"
  "pfcmpeq\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcmp")
   (set_attr "mode" "V2SF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point conversion operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_pf2id"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(fix:V2SI (match_operand:V2SF 1 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW"
  "pf2id\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_pf2iw"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(sign_extend:V2SI
  (ss_truncate:V2HI
    (fix:V2SI
      (match_operand:V2SF 1 "nonimmediate_operand" "ym")))))]
  "TARGET_3DNOW_A"
  "pf2iw\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_pi2fw"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(float:V2SF
  (sign_extend:V2SI
    (truncate:V2HI
      (match_operand:V2SI 1 "nonimmediate_operand" "ym")))))]
```

```
  "TARGET_3DNOW_A"
  "pi2fw\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "V2SF")])

(define_insn "mmx_floatv2si2"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(float:V2SF (match_operand:V2SI 1 "nonimmediate_operand" "ym")))]
  "TARGET_3DNOW"
  "pi2fd\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "V2SF")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel single-precision floating point element swizzling
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_pswapdv2sf2"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(vec_select:V2SF (match_operand:V2SF 1 "nonimmediate_operand" "ym")
 (parallel [(const_int 1) (const_int 0)])))]
  "TARGET_3DNOW_A"
  "pswapd\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "V2SF")])

(define_insn "*vec_dupv2sf"
  [(set (match_operand:V2SF 0 "register_operand" "=y")
(vec_duplicate:V2SF
  (match_operand:SF 1 "register_operand" "0")))]
  "TARGET_MMX"
  "punpckldq\t%0, %0"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "*mmx_concatv2sf"
  [(set (match_operand:V2SF 0 "register_operand"     "=y,y")
(vec_concat:V2SF
  (match_operand:SF 1 "nonimmediate_operand" " 0,rm")
  (match_operand:SF 2 "vector_move_operand"  "ym,C")))]
  "TARGET_MMX && !TARGET_SSE"
  "@
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt,mmxmov")
   (set_attr "mode" "DI")])

(define_expand "vec_setv2sf"
```

```
  [(match_operand:V2SF 0 "register_operand" "")
   (match_operand:SF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_insn_and_split "*vec_extractv2sf_0"
  [(set (match_operand:SF 0 "nonimmediate_operand"     "=x,y,m,m,frxy")
(vec_select:SF
  (match_operand:V2SF 1 "nonimmediate_operand" " x,y,x,y,m")
  (parallel [(const_int 0)])))]
  "TARGET_MMX && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  rtx op1 = operands[1];
  if (REG_P (op1))
    op1 = gen_rtx_REG (SFmode, REGNO (op1));
  else
    op1 = gen_lowpart (SFmode, op1);
  emit_move_insn (operands[0], op1);
  DONE;
})

(define_insn "*vec_extractv2sf_1"
  [(set (match_operand:SF 0 "nonimmediate_operand"     "=y,x,frxy")
(vec_select:SF
  (match_operand:V2SF 1 "nonimmediate_operand" " 0,0,o")
  (parallel [(const_int 1)])))]
  "TARGET_MMX && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
   punpckhdq\t%0, %0
   unpckhps\t%0, %0
   #"
  [(set_attr "type" "mmxcvt,sselog1,*")
   (set_attr "mode" "DI,V4SF,SI")])

(define_split
  [(set (match_operand:SF 0 "register_operand" "")
(vec_select:SF
  (match_operand:V2SF 1 "memory_operand" "")
  (parallel [(const_int 1)])))]
  "TARGET_MMX && reload_completed"
  [(const_int 0)]
{
```

```
  operands[1] = adjust_address (operands[1], SFmode, 4);
  emit_move_insn (operands[0], operands[1]);
  DONE;
})

(define_expand "vec_extractv2sf"
  [(match_operand:SF 0 "register_operand" "")
   (match_operand:V2SF 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv2sf"
  [(match_operand:V2SF 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral arithmetic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_add<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
        (plus:MMXMODEI
  (match_operand:MMXMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (PLUS, <MODE>mode, operands)"
  "padd<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_adddi3"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (unspec:DI
 [(plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0")
   (match_operand:DI 2 "nonimmediate_operand" "ym"))]
 UNSPEC_NOP))]
  "TARGET_MMX && ix86_binary_operator_ok (PLUS, DImode, operands)"
  "paddq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
```

```
   (set_attr "mode" "DI")])

(define_insn "mmx_ssadd<mode>3"
  [(set (match_operand:MMXMODE12 0 "register_operand" "=y")
        (ss_plus:MMXMODE12
  (match_operand:MMXMODE12 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODE12 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "padds<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_usadd<mode>3"
  [(set (match_operand:MMXMODE12 0 "register_operand" "=y")
        (us_plus:MMXMODE12
  (match_operand:MMXMODE12 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODE12 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "paddus<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_sub<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
        (minus:MMXMODEI
  (match_operand:MMXMODEI 1 "register_operand" "0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "psub<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_subdi3"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (unspec:DI
 [(minus:DI (match_operand:DI 1 "register_operand" "0")
    (match_operand:DI 2 "nonimmediate_operand" "ym"))]
 UNSPEC_NOP))]
  "TARGET_MMX"
  "psubq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_sssub<mode>3"
  [(set (match_operand:MMXMODE12 0 "register_operand" "=y")
        (ss_minus:MMXMODE12
  (match_operand:MMXMODE12 1 "register_operand" "0")
  (match_operand:MMXMODE12 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "psubs<mmxvecsize>\t{%2, %0|%0, %2}"
```

```
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_ussub<mode>3"
  [(set (match_operand:MMXMODE12 0 "register_operand" "=y")
        (us_minus:MMXMODE12
  (match_operand:MMXMODE12 1 "register_operand" "0")
  (match_operand:MMXMODE12 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "psubus<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_mulv4hi3"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
        (mult:V4HI (match_operand:V4HI 1 "nonimmediate_operand" "%0")
   (match_operand:V4HI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (MULT, V4HImode, operands)"
  "pmullw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])

(define_insn "mmx_smulv4hi3_highpart"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(truncate:V4HI
 (lshiftrt:V4SI
  (mult:V4SI (sign_extend:V4SI
      (match_operand:V4HI 1 "nonimmediate_operand" "%0"))
     (sign_extend:V4SI
      (match_operand:V4HI 2 "nonimmediate_operand" "ym")))
  (const_int 16))))]
  "TARGET_MMX && ix86_binary_operator_ok (MULT, V4HImode, operands)"
  "pmulhw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])

(define_insn "mmx_umulv4hi3_highpart"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(truncate:V4HI
 (lshiftrt:V4SI
  (mult:V4SI (zero_extend:V4SI
      (match_operand:V4HI 1 "nonimmediate_operand" "%0"))
     (zero_extend:V4SI
      (match_operand:V4HI 2 "nonimmediate_operand" "ym")))
  (const_int 16))))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (MULT, V4HImode, operands)"
  "pmulhuw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])
```

```
(define_insn "mmx_pmaddwd"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
        (plus:V2SI
  (mult:V2SI
    (sign_extend:V2SI
      (vec_select:V2HI
(match_operand:V4HI 1 "nonimmediate_operand" "%0")
(parallel [(const_int 0) (const_int 2)])))
    (sign_extend:V2SI
      (vec_select:V2HI
(match_operand:V4HI 2 "nonimmediate_operand" "ym")
(parallel [(const_int 0) (const_int 2)])))
  (mult:V2SI
    (sign_extend:V2SI
      (vec_select:V2HI (match_dup 1)
(parallel [(const_int 1) (const_int 3)])))
    (sign_extend:V2SI
      (vec_select:V2HI (match_dup 2)
(parallel [(const_int 1) (const_int 3)])))))]
  "TARGET_MMX && ix86_binary_operator_ok (MULT, V4HImode, operands)"
  "pmaddwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])

(define_insn "mmx_pmulhrwv4hi3"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(truncate:V4HI
  (lshiftrt:V4SI
    (plus:V4SI
      (mult:V4SI
        (sign_extend:V4SI
  (match_operand:V4HI 1 "nonimmediate_operand" "%0"))
        (sign_extend:V4SI
  (match_operand:V4HI 2 "nonimmediate_operand" "ym")))
      (const_vector:V4SI [(const_int 32768) (const_int 32768)
  (const_int 32768) (const_int 32768)]))
    (const_int 16))))]
  "TARGET_3DNOW && ix86_binary_operator_ok (MULT, V4HImode, operands)"
  "pmulhrw\\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])

(define_insn "sse2_umulsidi3"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (mult:DI
  (zero_extend:DI
    (vec_select:SI
      (match_operand:V2SI 1 "nonimmediate_operand" "%0")
      (parallel [(const_int 0)]))))
```

```
  (zero_extend:DI
    (vec_select:SI
      (match_operand:V2SI 2 "nonimmediate_operand" "ym")
      (parallel [(const_int 0)]))))]
  "TARGET_SSE2 && ix86_binary_operator_ok (MULT, V2SImode, operands)"
  "pmuludq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxmul")
   (set_attr "mode" "DI")])

(define_insn "mmx_umaxv8qi3"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
        (umax:V8QI (match_operand:V8QI 1 "nonimmediate_operand" "%0")
   (match_operand:V8QI 2 "nonimmediate_operand" "ym")))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (UMAX, V8QImode, operands)"
  "pmaxub\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_smaxv4hi3"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
        (smax:V4HI (match_operand:V4HI 1 "nonimmediate_operand" "%0")
   (match_operand:V4HI 2 "nonimmediate_operand" "ym")))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (SMAX, V4HImode, operands)"
  "pmaxsw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_uminv8qi3"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
        (umin:V8QI (match_operand:V8QI 1 "nonimmediate_operand" "%0")
   (match_operand:V8QI 2 "nonimmediate_operand" "ym")))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (UMIN, V8QImode, operands)"
  "pminub\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_sminv4hi3"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
        (smin:V4HI (match_operand:V4HI 1 "nonimmediate_operand" "%0")
   (match_operand:V4HI 2 "nonimmediate_operand" "ym")))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (SMIN, V4HImode, operands)"
  "pminsw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_ashr<mode>3"
```

```
  [(set (match_operand:MMXMODE24 0 "register_operand" "=y")
        (ashiftrt:MMXMODE24
  (match_operand:MMXMODE24 1 "register_operand" "0")
  (match_operand:DI 2 "nonmemory_operand" "yi")))]
  "TARGET_MMX"
  "psra<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_lshr<mode>3"
  [(set (match_operand:MMXMODE24 0 "register_operand" "=y")
        (lshiftrt:MMXMODE24
  (match_operand:MMXMODE24 1 "register_operand" "0")
  (match_operand:DI 2 "nonmemory_operand" "yi")))]
  "TARGET_MMX"
  "psrl<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_lshrdi3"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (unspec:DI
  [(lshiftrt:DI (match_operand:DI 1 "register_operand" "0")
       (match_operand:DI 2 "nonmemory_operand" "yi"))]
  UNSPEC_NOP))]
  "TARGET_MMX"
  "psrlq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_ashl<mode>3"
  [(set (match_operand:MMXMODE24 0 "register_operand" "=y")
        (ashift:MMXMODE24
  (match_operand:MMXMODE24 1 "register_operand" "0")
  (match_operand:DI 2 "nonmemory_operand" "yi")))]
  "TARGET_MMX"
  "psll<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_ashldi3"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (unspec:DI
 [(ashift:DI (match_operand:DI 1 "register_operand" "0")
     (match_operand:DI 2 "nonmemory_operand" "yi"))]
 UNSPEC_NOP))]
  "TARGET_MMX"
  "psllq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral comparisons
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_eq<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
        (eq:MMXMODEI
  (match_operand:MMXMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (EQ, <MODE>mode, operands)"
  "pcmpeq<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcmp")
   (set_attr "mode" "DI")])

(define_insn "mmx_gt<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
        (gt:MMXMODEI
  (match_operand:MMXMODEI 1 "register_operand" "0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "pcmpgt<mmxvecsize>\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcmp")
   (set_attr "mode" "DI")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral logical operations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_and<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
(and:MMXMODEI
  (match_operand:MMXMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (AND, <MODE>mode, operands)"
  "pand\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_nand<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
(and:MMXMODEI
  (not:MMXMODEI (match_operand:MMXMODEI 1 "register_operand" "0"))
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX"
  "pandn\t{%2, %0|%0, %2}"
```

```
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_ior<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
        (ior:MMXMODEI
  (match_operand:MMXMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (IOR, <MODE>mode, operands)"
  "por\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")])

(define_insn "mmx_xor<mode>3"
  [(set (match_operand:MMXMODEI 0 "register_operand" "=y")
(xor:MMXMODEI
  (match_operand:MMXMODEI 1 "nonimmediate_operand" "%0")
  (match_operand:MMXMODEI 2 "nonimmediate_operand" "ym")))]
  "TARGET_MMX && ix86_binary_operator_ok (XOR, <MODE>mode, operands)"
  "pxor\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxadd")
   (set_attr "mode" "DI")
   (set_attr "memory" "none")])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parallel integral element swizzling
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_packsswb"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
(vec_concat:V8QI
  (ss_truncate:V4QI
    (match_operand:V4HI 1 "register_operand" "0"))
  (ss_truncate:V4QI
    (match_operand:V4HI 2 "nonimmediate_operand" "ym"))))]
  "TARGET_MMX"
  "packsswb\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_packssdw"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(vec_concat:V4HI
  (ss_truncate:V2HI
    (match_operand:V2SI 1 "register_operand" "0"))
  (ss_truncate:V2HI
    (match_operand:V2SI 2 "nonimmediate_operand" "ym"))))]
  "TARGET_MMX"
```

```
  "packssdw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_packuswb"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
(vec_concat:V8QI
  (us_truncate:V4QI
    (match_operand:V4HI 1 "register_operand" "0"))
  (us_truncate:V4QI
    (match_operand:V4HI 2 "nonimmediate_operand" "ym"))))]
  "TARGET_MMX"
  "packuswb\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpckhbw"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
(vec_select:V8QI
  (vec_concat:V16QI
    (match_operand:V8QI 1 "register_operand" "0")
    (match_operand:V8QI 2 "nonimmediate_operand" "ym"))
          (parallel [(const_int 4) (const_int 12)
                     (const_int 5) (const_int 13)
                     (const_int 6) (const_int 14)
                     (const_int 7) (const_int 15)]))))]
  "TARGET_MMX"
  "punpckhbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpcklbw"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
(vec_select:V8QI
  (vec_concat:V16QI
    (match_operand:V8QI 1 "register_operand" "0")
    (match_operand:V8QI 2 "nonimmediate_operand" "ym"))
          (parallel [(const_int 0) (const_int 8)
                     (const_int 1) (const_int 9)
                     (const_int 2) (const_int 10)
                     (const_int 3) (const_int 11)]))))]
  "TARGET_MMX"
  "punpcklbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpckhwd"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(vec_select:V4HI
  (vec_concat:V8HI
```

```
    (match_operand:V4HI 1 "register_operand" "0")
    (match_operand:V4HI 2 "nonimmediate_operand" "ym"))
          (parallel [(const_int 2) (const_int 6)
                     (const_int 3) (const_int 7)]))))]
  "TARGET_MMX"
  "punpckhwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpcklwd"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(vec_select:V4HI
  (vec_concat:V8HI
    (match_operand:V4HI 1 "register_operand" "0")
    (match_operand:V4HI 2 "nonimmediate_operand" "ym"))
          (parallel [(const_int 0) (const_int 4)
                     (const_int 1) (const_int 5)]))))]
  "TARGET_MMX"
  "punpcklwd\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpckhdq"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_select:V2SI
  (vec_concat:V4SI
    (match_operand:V2SI 1 "register_operand" "0")
    (match_operand:V2SI 2 "nonimmediate_operand" "ym"))
  (parallel [(const_int 1)
     (const_int 3)])))]
  "TARGET_MMX"
  "punpckhdq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_punpckldq"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_select:V2SI
  (vec_concat:V4SI
    (match_operand:V2SI 1 "register_operand" "0")
    (match_operand:V2SI 2 "nonimmediate_operand" "ym"))
  (parallel [(const_int 0)
     (const_int 2)])))]
  "TARGET_MMX"
  "punpckldq\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_expand "mmx_pinsrw"
  [(set (match_operand:V4HI 0 "register_operand" "")
```

```
        (vec_merge:V4HI
          (vec_duplicate:V4HI
            (match_operand:SI 2 "nonimmediate_operand" ""))
  (match_operand:V4HI 1 "register_operand" "")
          (match_operand:SI 3 "const_0_to_3_operand" "")))]
  "TARGET_SSE || TARGET_3DNOW_A"
{
  operands[2] = gen_lowpart (HImode, operands[2]);
  operands[3] = GEN_INT (1 << INTVAL (operands[3]));
})

(define_insn "*mmx_pinsrw"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
        (vec_merge:V4HI
          (vec_duplicate:V4HI
            (match_operand:HI 2 "nonimmediate_operand" "rm"))
  (match_operand:V4HI 1 "register_operand" "0")
          (match_operand:SI 3 "const_pow2_1_to_8_operand" "n")))]
  "TARGET_SSE || TARGET_3DNOW_A"
{
  operands[3] = GEN_INT (exact_log2 (INTVAL (operands[3])));
  return "pinsrw\t{%3, %k2, %0|%0, %k2, %3}";
}
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_pextrw"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (zero_extend:SI
  (vec_select:HI
    (match_operand:V4HI 1 "register_operand" "y")
    (parallel [(match_operand:SI 2 "const_0_to_3_operand" "n")])))))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "pextrw\t{%2, %1, %0|%0, %1, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_expand "mmx_pshufw"
  [(match_operand:V4HI 0 "register_operand" "")
   (match_operand:V4HI 1 "nonimmediate_operand" "")
   (match_operand:SI 2 "const_int_operand" "")]
  "TARGET_SSE || TARGET_3DNOW_A"
{
  int mask = INTVAL (operands[2]);
  emit_insn (gen_mmx_pshufw_1 (operands[0], operands[1],
                               GEN_INT ((mask >> 0) & 3),
                               GEN_INT ((mask >> 2) & 3),
                               GEN_INT ((mask >> 4) & 3),
                               GEN_INT ((mask >> 6) & 3)));
  DONE;
```

```
})

(define_insn "mmx_pshufw_1"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
        (vec_select:V4HI
          (match_operand:V4HI 1 "nonimmediate_operand" "ym")
          (parallel [(match_operand 2 "const_0_to_3_operand" "")
                     (match_operand 3 "const_0_to_3_operand" "")
                     (match_operand 4 "const_0_to_3_operand" "")
                     (match_operand 5 "const_0_to_3_operand" "")])))]
  "TARGET_SSE || TARGET_3DNOW_A"
{
  int mask = 0;
  mask |= INTVAL (operands[2]) << 0;
  mask |= INTVAL (operands[3]) << 2;
  mask |= INTVAL (operands[4]) << 4;
  mask |= INTVAL (operands[5]) << 6;
  operands[2] = GEN_INT (mask);

  return "pshufw\t{%2, %1, %0|%0, %1, %2}";
}
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_pswapdv2si2"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_select:V2SI
  (match_operand:V2SI 1 "nonimmediate_operand" "ym")
  (parallel [(const_int 1) (const_int 0)])))]
  "TARGET_3DNOW_A"
  "pswapd\\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "*vec_dupv4hi"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(vec_duplicate:V4HI
  (truncate:HI
    (match_operand:SI 1 "register_operand" "0"))))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "pshufw\t{$0, %0, %0|%0, %0, 0}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "*vec_dupv2si"
  [(set (match_operand:V2SI 0 "register_operand" "=y")
(vec_duplicate:V2SI
  (match_operand:SI 1 "register_operand" "0")))]
  "TARGET_MMX"
  "punpckldq\t%0, %0"
```

```
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "*mmx_concatv2si"
  [(set (match_operand:V2SI 0 "register_operand"     "=y,y")
(vec_concat:V2SI
  (match_operand:SI 1 "nonimmediate_operand" " 0,rm")
  (match_operand:SI 2 "vector_move_operand"  "ym,C")))]
  "TARGET_MMX && !TARGET_SSE"
  "@
   punpckldq\t{%2, %0|%0, %2}
   movd\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt,mmxmov")
   (set_attr "mode" "DI")])

(define_expand "vec_setv2si"
  [(match_operand:V2SI 0 "register_operand" "")
   (match_operand:SI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_insn_and_split "*vec_extractv2si_0"
  [(set (match_operand:SI 0 "nonimmediate_operand"     "=x,y,m,m,frxy")
(vec_select:SI
  (match_operand:V2SI 1 "nonimmediate_operand" " x,y,x,y,m")
  (parallel [(const_int 0)])))]
  "TARGET_MMX && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "#"
  "&& reload_completed"
  [(const_int 0)]
{
  rtx op1 = operands[1];
  if (REG_P (op1))
    op1 = gen_rtx_REG (SImode, REGNO (op1));
  else
    op1 = gen_lowpart (SImode, op1);
  emit_move_insn (operands[0], op1);
  DONE;
})

(define_insn "*vec_extractv2si_1"
  [(set (match_operand:SI 0 "nonimmediate_operand"     "=y,Y,Y,x,frxy")
(vec_select:SI
  (match_operand:V2SI 1 "nonimmediate_operand" " 0,0,Y,0,o")
  (parallel [(const_int 1)])))]
```

```
  "TARGET_MMX && !(MEM_P (operands[0]) && MEM_P (operands[1]))"
  "@
   punpckhdq\t%0, %0
   punpckhdq\t%0, %0
   pshufd\t{$85, %1, %0|%0, %1, 85}
   unpckhps\t%0, %0
   #"
  [(set_attr "type" "mmxcvt,sselog1,sselog1,sselog1,*")
   (set_attr "mode" "DI,TI,TI,V4SF,SI")])

(define_split
  [(set (match_operand:SI 0 "register_operand" "")
(vec_select:SI
  (match_operand:V2SI 1 "memory_operand" "")
  (parallel [(const_int 1)])))]
  "TARGET_MMX && reload_completed"
  [(const_int 0)]
{
  operands[1] = adjust_address (operands[1], SImode, 4);
  emit_move_insn (operands[0], operands[1]);
  DONE;
})

(define_expand "vec_extractv2si"
  [(match_operand:SI 0 "register_operand" "")
   (match_operand:V2SI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv2si"
  [(match_operand:V2SI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

(define_expand "vec_setv4hi"
  [(match_operand:V4HI 0 "register_operand" "")
   (match_operand:HI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
```

```
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv4hi"
  [(match_operand:HI 0 "register_operand" "")
   (match_operand:V4HI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv4hi"
  [(match_operand:V4HI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

(define_expand "vec_setv8qi"
  [(match_operand:V8QI 0 "register_operand" "")
   (match_operand:QI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_set (false, operands[0], operands[1],
  INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_extractv8qi"
  [(match_operand:QI 0 "register_operand" "")
   (match_operand:V8QI 1 "register_operand" "")
   (match_operand 2 "const_int_operand" "")]
  "TARGET_MMX"
{
  ix86_expand_vector_extract (false, operands[0], operands[1],
      INTVAL (operands[2]));
  DONE;
})

(define_expand "vec_initv8qi"
  [(match_operand:V8QI 0 "register_operand" "")
   (match_operand 1 "" "")]
  "TARGET_SSE"
```

```
{
  ix86_expand_vector_init (false, operands[0], operands[1]);
  DONE;
})

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Miscellaneous
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define_insn "mmx_uavgv8qi3"
  [(set (match_operand:V8QI 0 "register_operand" "=y")
(truncate:V8QI
  (lshiftrt:V8HI
    (plus:V8HI
      (plus:V8HI
(zero_extend:V8HI
  (match_operand:V8QI 1 "nonimmediate_operand" "%0"))
(zero_extend:V8HI
  (match_operand:V8QI 2 "nonimmediate_operand" "ym")))
      (const_vector:V8HI [(const_int 1) (const_int 1)
  (const_int 1) (const_int 1)
  (const_int 1) (const_int 1)
  (const_int 1) (const_int 1)]))
    (const_int 1))))]
  "(TARGET_SSE || TARGET_3DNOW)
   && ix86_binary_operator_ok (PLUS, V8QImode, operands)"
{
  /* These two instructions have the same operation, but their encoding
     is different.  Prefer the one that is de facto standard.  */
  if (TARGET_SSE || TARGET_3DNOW_A)
    return "pavgb\t{%2, %0|%0, %2}";
  else
    return "pavgusb\\t{%2, %0|%0, %2}";
}
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_uavgv4hi3"
  [(set (match_operand:V4HI 0 "register_operand" "=y")
(truncate:V4HI
  (lshiftrt:V4SI
    (plus:V4SI
      (plus:V4SI
(zero_extend:V4SI
  (match_operand:V4HI 1 "nonimmediate_operand" "%0"))
(zero_extend:V4SI
  (match_operand:V4HI 2 "nonimmediate_operand" "ym")))
      (const_vector:V4SI [(const_int 1) (const_int 1)
```

```
  (const_int 1) (const_int 1)]))
    (const_int 1))))]
  "(TARGET_SSE || TARGET_3DNOW_A)
   && ix86_binary_operator_ok (PLUS, V4HImode, operands)"
  "pavgw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_psadbw"
  [(set (match_operand:DI 0 "register_operand" "=y")
        (unspec:DI [(match_operand:V8QI 1 "register_operand" "0")
    (match_operand:V8QI 2 "nonimmediate_operand" "ym")]
   UNSPEC_PSADBW))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "psadbw\t{%2, %0|%0, %2}"
  [(set_attr "type" "mmxshft")
   (set_attr "mode" "DI")])

(define_insn "mmx_pmovmskb"
  [(set (match_operand:SI 0 "register_operand" "=r")
(unspec:SI [(match_operand:V8QI 1 "register_operand" "y")]
   UNSPEC_MOVMSK))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "pmovmskb\t{%1, %0|%0, %1}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_expand "mmx_maskmovq"
  [(set (match_operand:V8QI 0 "memory_operand" "")
(unspec:V8QI [(match_operand:V8QI 1 "register_operand" "y")
      (match_operand:V8QI 2 "register_operand" "y")
      (match_dup 0)]
     UNSPEC_MASKMOV))]
  "TARGET_SSE || TARGET_3DNOW_A"
  "")

(define_insn "*mmx_maskmovq"
  [(set (mem:V8QI (match_operand:SI 0 "register_operand" "D"))
(unspec:V8QI [(match_operand:V8QI 1 "register_operand" "y")
      (match_operand:V8QI 2 "register_operand" "y")
      (mem:V8QI (match_dup 0))]
     UNSPEC_MASKMOV))]
  "(TARGET_SSE || TARGET_3DNOW_A) && !TARGET_64BIT"
  ;; @@@ check ordering of operands in intel/nonintel syntax
  "maskmovq\t{%2, %1|%1, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "*mmx_maskmovq_rex"
  [(set (mem:V8QI (match_operand:DI 0 "register_operand" "D"))
```

```
(unspec:V8QI [(match_operand:V8QI 1 "register_operand" "y")
      (match_operand:V8QI 2 "register_operand" "y")
      (mem:V8QI (match_dup 0))]
     UNSPEC_MASKMOV))]
  "(TARGET_SSE || TARGET_3DNOW_A) && TARGET_64BIT"
  ;; @@@ check ordering of operands in intel/nonintel syntax
  "maskmovq\t{%2, %1|%1, %2}"
  [(set_attr "type" "mmxcvt")
   (set_attr "mode" "DI")])

(define_insn "mmx_emms"
  [(unspec_volatile [(const_int 0)] UNSPECV_EMMS)
   (clobber (reg:XF 8))
   (clobber (reg:XF 9))
   (clobber (reg:XF 10))
   (clobber (reg:XF 11))
   (clobber (reg:XF 12))
   (clobber (reg:XF 13))
   (clobber (reg:XF 14))
   (clobber (reg:XF 15))
   (clobber (reg:DI 29))
   (clobber (reg:DI 30))
   (clobber (reg:DI 31))
   (clobber (reg:DI 32))
   (clobber (reg:DI 33))
   (clobber (reg:DI 34))
   (clobber (reg:DI 35))
   (clobber (reg:DI 36))]
  "TARGET_MMX"
  "emms"
  [(set_attr "type" "mmx")
   (set_attr "memory" "unknown")])

(define_insn "mmx_femms"
  [(unspec_volatile [(const_int 0)] UNSPECV_FEMMS)
   (clobber (reg:XF 8))
   (clobber (reg:XF 9))
   (clobber (reg:XF 10))
   (clobber (reg:XF 11))
   (clobber (reg:XF 12))
   (clobber (reg:XF 13))
   (clobber (reg:XF 14))
   (clobber (reg:XF 15))
   (clobber (reg:DI 29))
   (clobber (reg:DI 30))
   (clobber (reg:DI 31))
   (clobber (reg:DI 32))
   (clobber (reg:DI 33))
   (clobber (reg:DI 34))
   (clobber (reg:DI 35))
```

```
   (clobber (reg:DI 36))]
  "TARGET_3DNOW"
  "femms"
  [(set_attr "type" "mmx")
   (set_attr "memory" "none")])
```

1.10 GCC machine description for i386 synchronization instructions.

;;(include "sync.md")

```
(define_mode_macro IMODE [QI HI SI (DI "TARGET_64BIT")])
(define_mode_attr modesuffix [(QI "b") (HI "w") (SI "l") (DI "q")])
(define_mode_attr modeconstraint [(QI "q") (HI "r") (SI "r") (DI "r")])
(define_mode_attr immconstraint [(QI "i") (HI "i") (SI "i") (DI "e")])

;; ??? It would be possible to use cmpxchg8b on pentium for DImode
;; changes.  It's complicated because the insn uses ecx:ebx as the
;; new value; note that the registers are reversed from the order
;; that they'd be in with (reg:DI 2 ecx).  Similarly for TImode
;; data in 64-bit mode.

(define_insn "sync_compare_and_swap<mode>"
  [(set (match_operand:IMODE 0 "register_operand" "=a")
(match_operand:IMODE 1 "memory_operand" "+m"))
   (set (match_dup 1)
(unspec_volatile:IMODE
  [(match_dup 1)
   (match_operand:IMODE 2 "register_operand" "a")
   (match_operand:IMODE 3 "register_operand" "<modeconstraint>")]
  UNSPECV_CMPXCHG_1))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_CMPXCHG"
  "lock\;cmpxchg{<modesuffix>}\t{%3, %1|%1, %3}")

(define_expand "sync_compare_and_swap_cc<mode>"
  [(parallel
    [(set (match_operand:IMODE 0 "register_operand" "")
  (match_operand:IMODE 1 "memory_operand" ""))
     (set (match_dup 1)
  (unspec_volatile:IMODE
    [(match_dup 1)
     (match_operand:IMODE 2 "register_operand" "")
     (match_operand:IMODE 3 "register_operand" "")]
    UNSPECV_CMPXCHG_1))
     (set (match_dup 4)
  (compare:CCZ
    (unspec_volatile:IMODE
      [(match_dup 1) (match_dup 2) (match_dup 3)] UNSPECV_CMPXCHG_2)
```

```
    (match_dup 2)))])]
  "TARGET_CMPXCHG"
{
  operands[4] = gen_rtx_REG (CCZmode, FLAGS_REG);
  ix86_compare_op0 = operands[3];
  ix86_compare_op1 = NULL;
  ix86_compare_emitted = operands[4];
})

(define_insn "*sync_compare_and_swap_cc<mode>"
  [(set (match_operand:IMODE 0 "register_operand" "=a")
(match_operand:IMODE 1 "memory_operand" "+m"))
   (set (match_dup 1)
(unspec_volatile:IMODE
  [(match_dup 1)
   (match_operand:IMODE 2 "register_operand" "a")
   (match_operand:IMODE 3 "register_operand" "<modeconstraint>")]
  UNSPECV_CMPXCHG_1))
   (set (reg:CCZ FLAGS_REG)
(compare:CCZ
  (unspec_volatile:IMODE
    [(match_dup 1) (match_dup 2) (match_dup 3)] UNSPECV_CMPXCHG_2)
  (match_dup 2)))]
  "TARGET_CMPXCHG"
  "lock\;cmpxchg{<modesuffix>}\t{%3, %1|%1, %3}")

(define_insn "sync_old_add<mode>"
  [(set (match_operand:IMODE 0 "register_operand" "=<modeconstraint>")
(unspec_volatile:IMODE
  [(match_operand:IMODE 1 "memory_operand" "+m")] UNSPECV_XCHG))
   (set (match_dup 1)
(plus:IMODE (match_dup 1)
    (match_operand:IMODE 2 "register_operand" "0")))
   (clobber (reg:CC FLAGS_REG))]
  "TARGET_XADD"
  "lock\;xadd{<modesuffix>}\t{%0, %1|%1, %0}")

;; Recall that xchg implicitly sets LOCK#, so adding it again wastes space.
(define_insn "sync_lock_test_and_set<mode>"
  [(set (match_operand:IMODE 0 "register_operand" "=<modeconstraint>")
(unspec_volatile:IMODE
  [(match_operand:IMODE 1 "memory_operand" "+m")] UNSPECV_XCHG))
   (set (match_dup 1)
(match_operand:IMODE 2 "register_operand" "0"))]
  ""
  "xchg{<modesuffix>}\t{%1, %0|%0, %1}")

(define_insn "sync_add<mode>"
  [(set (match_operand:IMODE 0 "memory_operand" "=m")
(unspec_volatile:IMODE
```

```
  [(plus:IMODE (match_dup 0)
     (match_operand:IMODE 1 "nonmemory_operand" "r<immconstraint>"))]
  UNSPECV_LOCK))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "lock\;add{<modesuffix>}\t{%1, %0|%0, %1}")

(define_insn "sync_sub<mode>"
  [(set (match_operand:IMODE 0 "memory_operand" "=m")
(unspec_volatile:IMODE
  [(minus:IMODE (match_dup 0)
     (match_operand:IMODE 1 "nonmemory_operand" "r<immconstraint>"))]
  UNSPECV_LOCK))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "lock\;sub{<modesuffix>}\t{%1, %0|%0, %1}")

(define_insn "sync_ior<mode>"
  [(set (match_operand:IMODE 0 "memory_operand" "=m")
(unspec_volatile:IMODE
  [(ior:IMODE (match_dup 0)
     (match_operand:IMODE 1 "nonmemory_operand" "r<immconstraint>"))]
  UNSPECV_LOCK))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "lock\;or{<modesuffix>}\t{%1, %0|%0, %1}")

(define_insn "sync_and<mode>"
  [(set (match_operand:IMODE 0 "memory_operand" "=m")
(unspec_volatile:IMODE
  [(and:IMODE (match_dup 0)
     (match_operand:IMODE 1 "nonmemory_operand" "r<immconstraint>"))]
  UNSPECV_LOCK))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "lock\;and{<modesuffix>}\t{%1, %0|%0, %1}")

(define_insn "sync_xor<mode>"
  [(set (match_operand:IMODE 0 "memory_operand" "=m")
(unspec_volatile:IMODE
  [(xor:IMODE (match_dup 0)
     (match_operand:IMODE 1 "nonmemory_operand" "r<immconstraint>"))]
  UNSPECV_LOCK))
   (clobber (reg:CC FLAGS_REG))]
  ""
  "lock\;xor{<modesuffix>}\t{%1, %0|%0, %1}")
```

2 GPL License

```
;; Copyright (C) 1988, 1994, 1995, 1996, 1997, 1998, 1999, 2000,
;; 2001, 2002, 2003, 2004, 2005
;; Free Software Foundation, Inc.
;; Mostly by William Schelter.
;; x86_64 support added by Jan Hubicka
;;
;; This file is part of GCC.
;;
;; GCC is free software; you can redistribute it and/or modify
;; it under the terms of the GNU General Public License as published by
;; the Free Software Foundation; either version 2, or (at your option)
;; any later version.
;;
;; GCC is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;; GNU General Public License for more details.
;;
;; You should have received a copy of the GNU General Public License
;; along with GCC; see the file COPYING.  If not, write to
;; the Free Software Foundation, 51 Franklin Street, Fifth Floor,
;; Boston, MA 02110-1301, USA.  */
```

References

[1] Appel, Andrew, **JLex: A Lexical Analyzer Generator for Java**, http://www.cs.princeton.edu/ appel/modern/java/JLex

[2] Hudson, Scott, **LALR Parser Generator for Java**, http://www2.cs.tum.edu/projects/cup

[3] IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z Order Number 253667-016 June 2005, p182

[4] IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z Order Number 253667-016 June 2005, Appendix A pA-8