Contents

Intel Pentium instruction-set specification

This specification describes the Intel Pentium **[cite intel:pentium]**. At the instruction-set level, this specification could almost be used to describe the 486; the Pentium supports just a few instructions not found on the 486. This specification has *not* been used in an application, which means that it is probably full of bugs. **[**We have plans for a test harness that should generate instructions at random, making sure that we get the same object code no matter whether we emit binary or assembly language, but as of July 1994 that harness is not in place.**]**

Specifying the *x86* is very different from specifying a RISC machine. In particular, it is difficult to know how to factor this architecture, or indeed whether to try to factor it at all. We've ended by trying to factor the opcodes into tables, but not to factor the instructions. Handling the *x86* opcode tables is painful, but we prefer it to specifying the opcodes individually, because we believe the tables reduce the likelihood of error. Factoring the instructions turned out to be a hopeless exercise, with one exception: there is a family of 8 groups of arithmetic instructions that factor nicely.

The *x86* is not quite sure whether it is an 8-bit, a 16-bit, or a 32-bit machine. We don't know the exact history, but we believe that the machine started life as an 8-bit machine, and that new opcodes were added when the architecture was extended to 16 bits. New opcodes were *not* added when it was extended again to 32 bits; instead, the presence or absence of a prefix is used to distinguish 16- from 32-bit instructions. Unfortunately, the encoding isn't completely specified at assembly time; the meaning of the prefix depends on the setting of a bit in the ``executable-segment descriptor.'' We have chosen to specify encodings in which instructions without prefixes operate on 32-bit quantities and the prefix selects 16-bit operation, but the encoding can be changed by changing just two lines in Section [->]. An application that wanted to be able to switch between encodings dynamically would have to use the toolkit to generate two sets of encoding procedures, one each for the 16- and 32-bit defaults.

One thing we do is provide a way to generate just the 32-bit subset:

<pentium32.spec>= **(U-> U->)**
keep *<32-bit constructors>* *<floating-point constructors>*

<pentium-int.spec>=
discard *<floating-point constructors>*

---

height 1pt
[5.8em] --- 9**One-Byte Opcode Map (page 0)**
0 8 0|6ADD|PUSH!POP62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv|ES!ES |PUSH!POP62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv|SS!SS | SEG!DAA62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv| =ES! | SEG!AAA62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv| =SS! 4|8INC general register8
|eAX?eCX?eDX?eBX?eSP?eBP?eSI?eDI 8
|eAX?eCX?eDX?eBX?eSP?eBP?eSI?eDI 6|PUSHA! POPA! BOUND!ARPL!
SEG!SEG!Operand!Address|PUSHAD!POPAD!Gv,Ma!Ew,Gw!=FS!=GS!Size! Size |JO?JNO?JB/JNAE/J?JNB/JAE/J?JZ?JNZ?JBE?JNBE

8|2Immediata Grp1!MOVB^*!Grp1!2TEST!2XCHG224
|Eb,Ib?Ev,Iv!AL,immed!Ev,Ib!Eb,Gb?Ev,Gv!Eb,Gb?Ev,Gv 9|NOP!7XCHG word to double-word register with eAX17
|!eCX?eDX?eBX?eSP?eBP?eSI?eDI A|4MOV|MOVSB!MOVsw!CMPSB!CMPSW44
|AL,Ob?eAX,Ov?Ob,AL?Ov,eAX|Xb,Yb!Xv,Yv!Xb,Yb!Xv,Yv B|8MOV immediate byte into register8
|AL?CL?DL?BL?AH?CH?DH?BH C|2Shift Grp2a!2RET near|LES!LDS!2MOV422
|Eb,Ib?Ev,Ib!Iw? |Gv,Mp?Gv,Mp!Eb,Ib?Ev,Iv |AAM!AAD!*!XLAT44
|Eb,1?Ev,1?Eb,CL?Ev,CL|!!! E|LOOPNE!LOOPE!LOOP!JCXZ/JEC!2IN!2OUT44
|Jb!Jb!Jb!Jb!AL,Ib?AX,Ib!Ib,AL?Ib,eAX 62
|!!!REPE!!!Eb?Ev **Pentium opcodes (page 0)[*]**

---

height 1pt
[5.8em] --- 9**One-Byte Opcode Map (page 1)**
816 0|6OR|PUSH!2-byte62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv|CS!escape |PUSH!POP62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv|DS!DS | SEG!DAS62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv| =CS! | SEG!AAS62
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev?AL,Ib?eAX,Iv| =DS! 4|8DEC general register8
|eAX?eCX?eDX?eBX?eSP?eBP?eSI?eDI 8
|eAX?eCX?eDX?eBX?eSP?eBP?eSI?eDI 6|PUSH!IMUL! PUSH!IMUL! INSB! INSW/D!OUTSB!OUTSW/D|Iv! Gv,Ev,Iv!Ib!
Gv,Ev,Ib!Yb,DX!Yv,DX! DX,Xb!DX,Xv |JS?JNS?JP?JNP?JL?JNL?JLE?JNLE 8|4MOV|MOV!LEA!MOV!POP44
|Eb,Gb?Ev,Gv?Gb,Eb?Gv,Ev|Ew,Sw!Gv,M!Sw,Ew!Ev 9|CBW!CWD/CDQ!CALL!WAIT|PUSHF!POPF!SAHF!LAHF|!!aP!|Fv!Fv!!
!STOSB!STOSW/D|LODSB!LODSW/D!SCASB!SCASW/D26
|AL,Ib?eAX,Iv!Yb,AL!Yv,eAX|AL,Xb!eAX,Xv!AL,Yb!eAX,Yv B|8MOV immediate word or double into word or double
register|eAX!eCX!eDX!eBX!eSP!eBP!eSI!eDI C|ENTER!LEAVE!RET far!RET far!INT!INT!INTO!IRET|Iw,Ib!!Iw!!3!Ib!! D|8ESC
(Escape to coprocessor instruction set)|8 E|CALL!3JMP!2IN!2OUT17
|Jv!Jv?Ap?Jb!AL,DX?eAX,DX!DX,AL?DX,eAX **Pentium opcodes (page 1)[*]**

Intel uses some naming conventions to try to tame the confusion surrounding operands. ``Operand specifiers'' describe the locations and sizes of operands. The specifiers are (mostly) composed from the following pieces:

Operand		Width	
E	effective address (memory or register)	b	8-bit bytes
		w	16-bit words
G	general-purpose register	d	32-bit doublewords
I	immediate	v	variable (w or d)

For example, the specifier ``Eb,Gb'' describes an 8-bit memory-to-register instruction. ``Ev,Gv'' describes a similar instruction that operates on 16 or 32 bits, depending on the presence or absence of a prefix. Most of the *x86* instructions are overloaded, but the opcode tables often use operand specifiers as suffixes to distinguish them. In some cases, where the machine specification doesn't give distinguishing suffixes, we have invented some.

Many of the instructions support all three sizes, and we specify them in two variants: a b variant with no prefix, and v variants with and without prefixes. When the two v variants differ only in the presence or absence of a prefix, we can specify them simultaneously using the ov pattern, which we define in Section [->] to mean ``optional prefix.'' Sometimes, however, we have to specify all three variants explicitly, as when there is an immediate operand---in that case, we have to give three different output patterns because the token holding the immediate operand may be 8, 16, or 32 bits wide.

Here is the overall structure of the Intel specification:

<pentium-core.spec>=
<field specs>
patterns *<patterns for integer opcodes>*
<pattern specs for other patterns>
<prefix assignments>
<placeholders>
relocatable reloc
<constructors for displacements>
<constructors for effective addresses>
<arithmetic constructors>

```
constructors
<alphabetical constructors>
```

Opcodes

For other machines, we were able to specify entire opcode tables in single declarations. The Intel tables are less tractable, because they don't just contain opcodes; they contain a mix of opcodes and suffixes. We've broken most of the tables into pieces, as shown below. There's an argument for abandoning opcode tables entirely, using only constructors to describe the encodings, but we've decided to keep opcodes. This style of specification lets us use a little factoring, and it gives us a little protection against errors in the distributed opcodes, like the `groupx` opcodes.

One-byte opcodes

We want to take advantage of factoring when possible, but it works well only for the ``arithemtic group,'' shown in the upper left corners of Figures [<-] and [<-], which represent the ``one-byte opcode map'' from pages A-5 and A-6 of the Intel manual **[cite intel:pentium]**. This corner of the opcode table clearly represents an outer product of 8 opcodes with 6 suffixes, and we treat it as such.

Most of the rest of the opcode table we treat in purely geometric fashion, using row, column, and page numbers as Cartesian coordinates to determine opcodes.

```
<field specs>= (<-U U->) [D->]
fields of opcodet (8) row 4:7 col 0:2 page 3:3
                    <more fields of opcode>

<placeholders>= (<-U U->) [D->]
placeholder for opcodet is HLT
```

Because the map is so chaotic, we break it into rows, for the most part specifying one or two rows at a time, but sometimes breaking rows into pieces. The first rows are among the most interesting; rows 0--3 contain the outer product of arithmetic operators with operand specifiers:

```
<patterns for integer opcodes>= (<-U U->) [D->]
arith is any of [ ADD OR
                  ADC SBB
                  AND SUB
                  XOR CMP ], which is row = {0 to 3} & page = [0 1]
[ Eb.Gb Ev.Gv Gb.Eb Gv.Ev AL.Ib eAX.Iv ] is col = {0 to 5}
```

The other columns in rows 0--3 follow a less discernible pattern.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ PUSH.ES POP.ES    PUSH.CS esc2
  PUSH.SS POP.SS    PUSH.DS POP.DS
  SEG.ES  DAA       SEG.CS  DAS
  SEG.SS  AAA       SEG.DS  AAS   ] is row = {0 to 3} & page = [0 1] & col = [6 7]
```

Rows 4 and 5 are the general-register opcodes, formed by an outer product of operation and register specifier.**[*]** Although the register specifier is actually part of the opcode, we treat it as an operand below. We create the field `r32` as an alias for `col`, so we can use special register names for the values.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
regops is any of [ INC  DEC
                   PUSH POP ], which is row = [4 5] & page = [0 1]

<more fields of opcode>= (<-U) [D->]
r32 0:2

<field specs>+= (<-U U->) [<-D->]
fieldinfo r32 is [names [ eAX eCX eDX eBX eSP eBP eSI eDI ]]

<more fields of opcode>+= (<-U) [<-D->]
sr16 0:2

<field specs>+= (<-U U->) [<-D->]
fieldinfo sr16 is [sparse [ cs=1, ss=2, ds=3, es=4, fs=5, gs=6 ] ]
```

```
<more fields of opcode>+= (<-U) [<-D->]
r16 0:2

<field specs>+= (<-U U->) [<-D->]
fieldinfo r16 is [names [ AX CX DX BX SP BP SI DI ]]
```

Row 6:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ PUSHA   POPA    BOUND   ARPL    SEG.FS  SEF.GS   OpPrefix AddrPrefix
  PUSH.Iv IMUL.Iv PUSH.Ib IMUL.Ib INSB    INSv     OUTSB    OUTSv
] is page = [0 1] & row = 6 & col = {0 to 7}
```

Row 7 is factored so the jump codes can be re-used in the two-byte opcode map. Again, the row contains an outer product, but this time the columns determine jump conditions, not operand specifiers.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
Jb is row = 7
cond is any of [ .O .NO .B .NB .Z .NZ .BE .NBE .S .NS .P .NP .L .NL .LE .NLE ],
      which is page = [0 1] & col = {0 to 7}
```

Row 8 is a bit hard to follow because it contains a seemingly random mix of opcodes and operand specifiers. On page 0, we've left the ``immediate Group1'' implicit, giving only the operand specifiers. On page 1, MOV shares the operand specifiers we gave with the arithmetic instructions in rows 0--3.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ Eb.Ib Ev.Iv MOVB Ev.Ib TEST.Eb.Gb TEST.Ev.Gv  XCHG.Eb.Gb XCHG.Ev.Gv ] is
               row = 8 & page = 0 & col = {0 to 7}
MOV is row = 8 & page = 1
[ MOV.Ew.Sw LEA MOV.Sw.Ew POP.Ev ] is row = 8 & page = 1 & col = {4 to 7}
```

On page 0, row 9 is XCHG (or NOP). Again, the register operand is actually part of the opcode. Page 1 has several opcodes.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
XCHG is row = 9 & page = 0
NOP is XCHG & col = 0
[ CBW CWDQ CALL.aP WAIT PUSHF POPF SAHF LAHF ] is row = 9 & page = 1 & col = {0 to 7}
```

Although there is an outer product or two lurking in row 10 (A), we don't try to specify it, because the operand specifiers used there aren't widely useful.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ MOV.AL.Ob  MOV.eAX.Ov  MOV.Ob.AL  MOV.Ov.eAX MOVSB    MOVSv     CMPSB   CMPSv
  TEST.AL.Ib TEST.eAX.Iv STOSB      STOSv      LODSB    LODSv     SCASB   SCASv
] is row = 10 & page = [0 1] & col = {0 to 7}
```

Row 11 (B) is another row in which the register operand is implicit in the opcode, but we need to define a new field to denote 8-bit registers.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
MOVib is row = 11 & page = 0
MOViv is row = 11 & page = 1

<more fields of opcode>+= (<-U) [<-D]
r8 0:2

<field specs>+= (<-U U->) [<-D->]
fieldinfo r8 is [names [ AL CL DL BL AH CH DH BH ]]
```

Rows 12 and 13 contain the bit operators; the others are easy.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ B.Eb.Ib B.Ev.Ib RET.Iw  RET     LES LDS MOV.Eb.Ib MOV.Ev.Iv
  B.Eb.1  B.Ev.1  B.Eb.CL B.Ev.CL AAM AAD _          XLAT
]   is row = [12 13] & page = 0 & col = {0 to 7}
[ ENTER LEAVE RET.far.Iw RET.far INT3 INT.Ib INTO IRET ]
    is row = 12      & page = 1 & col = {0 to 7}
ESC is row = 13      & page = 1
```

Neither of the last two rows can use factoring.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ LOOPNE  LOOPE   LOOP    JCXZ    IN.AL.Ib  IN.eAX.Ib  OUT.Ib.AL OUT.Ib.eAX
  LOCK    _       REPNE   REP     HLT       CMC        grp3.Eb   grp3.Ev

  CALL.Jv JMP.Jv  JMP.Ap  JMP.Jb  IN.AL.DX  IN.eAX.DX  OUT.DX.AL OUT.DX.eAX
  CLC     STC     CLI     STI     CLD       STD        grp4      grp5
] is page = [0 1] & row = [14 15] & col = {0 to 7}
```

Two-byte opcodes

The two-byte tables are fairly sparse, so we haven't bothered to reproduce them in a table for this report. We describe them one page at a time beginning with page 0. The first token of each two-token pattern contains the esc2 opcode.

The first two rows are:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ grp6       grp7       LAR         LSL
  MOV.Eb.Gb  MOV.Gv.Ev  MOV.Gb.Eb  MOV.Ev.Gv ]
is esc2; page = 0 & row = [0 1] & col = {0 to 3}
CLTS is esc2; page = 0 & row = 0 & col = 6
```

Row 3 is a block of MOV instructions.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ MOV.Rd.Cd  MOV.Rd.Dd  MOV.Cd.Rd  MOV.Dd.Rd  MOV.Rd.Td _  MOV.Td.Rd ]
is esc2; page = 0 & row = 3 & col = {0 to 6}
```

Row 4:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ WRMSR RDTSC RDMSR ] is esc2; page = 0 & row = 4 & col = {0 to 2}
```

The jumps in row 8 and sets in row 9 span two pages. They are outer products of opcodes and the conditions defined in row 7 of the one-byte opcode map.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
Jv   is esc2; row = 8
SETb is esc2; row = 9
```

Rows 10 and 11 are more madness:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ PUSH.FS       POP.FS        CPUID  BT    SHLD.Ib SHLD.CL _
  CMPXCHG.Eb.Gb CMPXCHG.Ev.Gv LSS    BTR   LFS     LGS     MOVZX.Gv.Eb  MOVZX.Gv.Ew ]
is esc2; page = 0 & row = [10 11] & col = {0 to 7}
```

Row 12 is the only remaining non-empty row on page 0.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ XADD.Eb.Gb XADD.Ev.Gv grp9 ] is esc2; page = 0 & row = 12 & col = [0 1 7]
```

There are even fewer opcodes on page 1.

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[INVD WBINVD] is esc2; row = 0 & page = 1 & col = [0 1]
```

Rows 1--7 are empty, and 8 and 9 were covered on the previous page. Row 10 is:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ PUSH.GS POP.GS RSM  BTS   SHRD.Ib SHRD.CL _ IMUL.Gv.Ev]
is esc2; row = 10 & page = 1 & col = {0 to 7}
```

Row 11:

```
<patterns for integer opcodes>+= (<-U U->) [<-D->]
[ grp8 BTC BSF BSR MOVSX.Gv.Eb MOVSX.Gv.Ew]
is esc2; page = 1 & row = 11 & col = {2 to 7}
```

And the single opcode on row 12:

```
<patterns for integer opcodes>+= (<-U U->) [<-D]
BSWAP is esc2; row = 12 & page = 1
```

Operands and effective addresses

Intel operands and addresses are described in Section 25.2.1 and Figure 25-2 of the Pentium manual. Effective addresses use a ``Mod R/M'' byte, the `mod` field of which determines the addressing mode. The Mod R/M byte also holds some bits that denote either a register operand or some extra parts of the opcode (as with the `group`*x* instructions). Indexed addressing modes use an additional byte, called ``SIB,'' which holds a scale factor and index and base registers.

```
<field specs>+= (<-U U->) [<-D->]
fields of modrm (8) mod 6:7 reg_opcode 3:5 r_m 0:2
fields of sib   (8) ss 6:7 index 3:5 base 0:2
```

```
<field specs>+= (<-U U->) [<-D->]
fieldinfo [ base index ] is
          [ names [ eAX eCX eDX eBX eSP eBP eSI eDI ] ]
fieldinfo ss is [ sparse [ "1" = 0, "2" = 1, "4" = 2, "8" = 3 ] ]
```

```
<placeholders>+= (<-U U->) [<-D->]
placeholder for modrm is HLT
placeholder for sib is HLT
```

We're faced with a specification problem because some Intel instructions accept only effective addresses that refer to operands in memory; register modes are not permitted. Most instructions, however, accept any kind of effective address. A good way to express this restriction would be with subtyping, but the toolkit doesn't implement subtyping, so instead we define two different constructor types: `Mem` to refer to effective addresses of operands in memory, and `Eaddr` to refer to any effective address. This separation requires the use of an identity constructor `E` to map `Mem`s into `Eaddr`s. Such a thing is bad enough in a specification, but these `E`'s have to be used in application programs, too. The ugliness is justified because it confers protection against inadvertantly using a register operand with an instruction that doesn't permit one.

```
<constructors for effective addresses>= (<-U U->)
relocatable d a
constructors
  Indir    [reg] : Mem { reg != 4, reg != 5 } is mod = 0 & r_m = reg
  Disp8   d[reg] : Mem { reg != 4 }           is mod = 1 & r_m = reg; i8  = d
  Disp32  d[reg] : Mem { reg != 4 }           is mod = 2 & r_m = reg; i32 = d
  Abs32   a      : Eaddr                      is mod = 0 & r_m = 5;   i32 = a
  Reg     reg    : Eaddr                      is mod = 3 & r_m = reg
  Index    [base][index * ss] : Mem { index != 4, base != 5 } is
                       mod = 0 & r_m = 4; index & base     & ss
  Index8  d[base][index * ss] : Mem { index != 4 } is
                       mod = 1 & r_m = 4; index & base     & ss; i8  = d
  Index32 d[base][index * ss] : Mem { index != 4 } is
                       mod = 2 & r_m = 4; index & base     & ss; i32 = d
  ShortIndex    d[index * ss] : Mem { index != 4 } is
                       mod = 0 & r_m = 4; index & base = 5 & ss; i32 = d
  E Mem : Eaddr is Mem
```

We'll eventually want to be able to keep only 32-bit constructors, to cut down on the time needed to generate encoding procedures.

```
<32-bit constructors>= (<-U) [D->]
Indir Disp32 Reg Index Index32 E
```

Now, this is good as far as it goes, but there are a couple of problems: no support for conditional assembly, and lots of constructors, which increases generation time. So let's get a bit clever:

```
<proposed constructors for effective addresses>=
constructors
  Indir0    [reg] { reg != 4, reg != 5 } is mod = 0 & r_m = reg
  Indir8   d[reg] { reg != 4 }           is mod = 1 & r_m = reg; i8  = d
  Indir32  d[reg] { reg != 4 }           is mod = 2 & r_m = reg; i32 = d
```

```
  Index0   [base][index*ss]  { index != 4, base != 5 } is
                        mod = 0 & r_m = 4; index & base     & ss
  Index8  d[base][index*ss]  { index != 4 } is
                        mod = 1 & r_m = 4; index & base     & ss; i8  = d
  Index32 d[base][index*ss]  { index != 4 } is
                        mod = 2 & r_m = 4; index & base     & ss; i32 = d

relocatable d
constructors
  Indir d[reg]            : Mem  when { d = 0 } is Indir0 (   reg)
                                 when {       } is Indir8 (d, reg)
                                 otherwise      is Indir32(d, reg)
  Index d[base][index*ss] : Mem  when { d = 0 } is Index0 (   base, index, ss)
                                 when {       } is Index8 (d, base, index, ss)
                                 otherwise      is Index32(d, base, index, ss)
  ShortIndex  d[index*ss] : Mem { index != 4 } is
                        mod = 0 & r_m = 4; index & base = 5 & ss; i32 = d
  E   Mem : Eaddr  is  Mem
  Reg reg : Eaddr  is  mod = 3 & r_m = reg

discard Indir0 Indir8 Indir32 Index0 Index8 Index32
```

The only problem now is that it's no longer possible to split out the 32-bit constructors.

Immediate operands occupy whole tokens and have their own classes.

```
<field specs>+= (<-U U->) [<-D]
fields of I8   (8) i8  0:7
fields of I16 (16) i16 0:15
fields of I32 (32) i32 0:31

<placeholders>+= (<-U U->) [<-D]
placeholder for I8  is HLT
placeholder for I16 is HLT; HLT
placeholder for I32 is HLT; HLT; HLT; HLT
```

Mod R/M opcodes

The Intel architecture offers the spectacle of putting some of the opcode bits in with the operands. The eight values of the `reg_opcode` field of the Mod R/M byte specify different opcodes, depending on the value of the opcode preceding the effective address. Most of these opcodes are notated by ``Group*x*'' in the manual. We use different sets of names for the values depending on what opcode precedes the effective-address specification. To make sure we don't mistakenly use a name like `INC.Eb` for `reg_opcode = 0` when the actual denotation is `INC.Ev`, we include in the specifications the opcode that must precede the Mod R/M byte---in this case, either `grp4` or `grp5`.

This kind of specification is conjoined below with specifications of opcodes and effective addresses. If

```
INC.Eb is grp4; reg_opcode = 0
```

then we might conjoin it with a `grp4` opcode followed by an effective address, writing:

```
INC.Eb Eaddr is (grp4; Eaddr) & INC.Eb
```

The conjunction of the explicit `grp4` with the `grp4` in the definition of `INC.Eb` ensures that `INC.Eb` is used correctly, since *`grp4 & grp4 ==grp4`*. If we incorrectly used `grp5` when defining the `INC.Eb` constructor, `grp4 & grp5` would evaluate to a pattern that never matches anything, and the toolkit would complain.

We've glossed over an important detail in the definition of this constructor. Because conjunction distributes over concatentation, the right-hand side of the `INC.Eb` constructor winds up being equivalent to

```
grp4; (Eaddr & reg_opcode = 0).
```

This conjunction, unfortunately, breaks the rules of conjunction, which requires both patterns conjoined to have the same *shape*, or sequence of token classes. The conjunction is legal when `Eaddr` is an instance of `Indir` or `Reg`, because those effective addresses consist solely of one Mod R/M token, but the other modes contain more tokens, and their shapes don't match `reg_opcode = 0`. The solution is to relax the shape constraint by using the ellipsis operator. ``p ...'' creates a pattern that is equivalent to `p`, except it is permissible to write `p ... & q` whenever `p`'s shape is a prefix of `q`'s shape. **[** The ellipsis may also be used as a prefix operator on patterns, in which case `... p & q` is permissible

whenever p's shape is a suffix of q's shape. We haven't had occasion to use such patterns in machine descriptions, because most hardware decodes complex instructions from left to right.**]** In the case at hand, every `Eaddr` begins with a Mod R/M token, so we can always write

```
Eaddr & reg_opcode = 0 ...
```

We've now covered enough detail to specify the Mod R/M or ``Group*x*'' opcodes. Just to add some extra complexity, groups 1-3 include opcodes that denote different operand specifiers. For example, `ADDi` can denote an integer add of bytes (`Eb.Ib`), 16-byte or 32-byte words (`Ev.Iv`), or words and bytes (`Ev.Ib`), depending on the suffix added to the opcode. For each constructor created from the `ADDi` pattern, only one operand specifier is conjoined into the output pattern; the resulting pattern has just one non-contradictory disjunct, so the bits to be emitted are uniquely determined. The constructors for these patterns are defined in Section [->].

```
<pattern specs for other patterns>= (<-U U->) [D->]
patterns
  arithI    is any of [ ADDi ORi ADCi SBBi ANDi SUBi XORi CMPi ],       # group 1
                        which is (Eb.Ib | Ev.Iv | Ev.Ib); reg_opcode = {0 to 7} ...
  bshifts   is B.Eb.1  | B.Eb.CL # D0 D2
  vshifts   is B.Ev.1  | B.Ev.CL # D1 D3
  immshifts is B.Eb.Ib | B.Ev.Ib # C0 C1
  rot       is any of [ ROL ROR RCL RCR SHLSAL SHR _ SAR],
                        which is (bshifts | vshifts | immshifts);
                                                    reg_opcode = {0 to 7} ...
  grp3ops   is any of
      [ TEST.Ib.Iv _ NOT NEG MUL.AL.eAX IMUL.AL.eAX DIV.AL.eAX IDIV.AL.eAX ],
                        which is (grp3.Eb | grp3.Ev); reg_opcode = {0 to 7} ...
  grp4ops   is any of [ INC.Eb DEC.Eb ],
                        which is grp4; reg_opcode = [0 1] ...
  grp5ops   is any of [ INC.Ev DEC.Ev CALL.Ev CALL.Ep JMP.Ev JMP.Ep PUSH.Ev _ ],
                        which is grp5; reg_opcode = {0 to 7} ...
  grp6ops   is any of [ SLDT STR LLDT LTR VERR VERW _ _ ],
                        which is grp6; reg_opcode = {0 to 7} ...
  grp7ops   is any of [ SGDT SIDT LGDT LIDT SMSW _ LMSW INVLPG ],
                        which is grp7; reg_opcode = {0 to 7} ...
  bittestI  is any of [ BTi BTSi BTRi BTCi ],
                        which is grp8; reg_opcode = {4 to 7} ...
  CMPXCHG8B is                    grp9; reg_opcode = 1 ...
```

Operand-size and address-size prefixes

[*] The Intel uses prefixes to distinguish 16- from 32-bit operands. The meaning of a prefix on the mode and on the setting of the *D* bit in the executable-segment descriptor (see pages 25-1ff of the Pentium manual). We assume here that *D=1*, making the default size 32 bits, but that assumption could be changed by reversing the definitions of `ow` and `od` given here. An application that wanted to be able to use both encodings would have to generate two sets of encoding procedures, perhaps using function pointers to switch back and forth.

In specifications, instructions with `b` suffixes (e.g., ``Eb,Gb'') use no prefix. Instructions with `v` suffixes (e.g., ``Ev,Gv'') begin with `ov`, which is followed by the rest of the instruction. When `ov` is used to build an opcode, this technique automatically creates two variants: `od`, a 32-bit variant with no prefix (`epsilon`) and `ow`, a 16-bit variant with prefix `OpPrefix`.

```
<prefix assignments>= (<-U U->) [D->]
patterns ow is OpPrefix
         od is epsilon
         ov is ow | od
```

The address prefix is similar, but we haven't figured out how it's to be used.

```
<prefix assignments>+= (<-U U->) [<-D]
patterns aw is AddrPrefix
         ad is epsilon
         av is aw | ad
```

Floating-point opcodes

The specifications of floating-point opcodes consume many more tables, but it's not necessary to say much about them; the specification techniques needed are those we use above. We have defined patterns `D9` though `DF`, which express

opcode values in hex; these are used in the specifications of the opcodes, so it seemed expedient to make them patterns, rather than continually writing something like `opcode = 0xd8`.

```
<pattern specs for other patterns>+= (<-U U->) [<-D->]
patterns
  [ D8 D9 DA DB DC DD DE DF ] is ESC & col = {0 to 7}
  [ FADD FMUL FCOM FCOMP FSUB FSUBR FDIV FDIVR ] is reg_opcode = {0 to 7}
  [ FLD _ FST FSTP FLDENV FLDCW FSTENV FSTCW ]   is reg_opcode = {0 to 7} ...
  [ FNOP ]                          is D9; mod = 3 & reg_opcode = 2 & r_m = [0]
  [ FCHS FABS _ _ FTST FXAM _ _  ] is D9; mod = 3 & reg_opcode = 4 & r_m = {0 to 7}
  [ F2XM1 FYL2X FPTAN FPATAN FXTRACT FPREM1 FDECSTP FINCSTP ]
                                    is D9; mod = 3 & reg_opcode = 6 & r_m = {0 to 7}
  FXCH                              is D9; mod = 3 & reg_opcode = 1
  Fconstants is any of [ FLD1 FLDL2T FLDL2E FLDPI FLDLG2 FLDLN2 FLDZ _ ], which
                                    is D9; mod = 3 & reg_opcode = 5 & r_m = {0 to 7}
  [ FPREM FYL2XP1 FSQRT FSINCOS FRNDINT FSCALE FSIN FCOS ]
                                    is D9; mod = 3 & reg_opcode = 7 & r_m = {0 to 7}
  [ FIADD FIMUL FICOM FICOMP FISUB FISUBR FIDIV FIDIVR ] is reg_opcode = {0 to 7} ...
  FUCOMPP                           is DA; mod = 3 & reg_opcode = 5 & r_m = 1
  [ FILD _ FIST FISTP FBLD FLD.ext FBSTP FSTP.ext ] is reg_opcode = {0 to 7} ...
  [ FCLEX FINIT ]                   is DB; mod = 3 & reg_opcode = 4 & r_m = [2 3]
  [ FRSTOR _ FSAVE FSTSW ]           is reg_opcode = {4 to 7} ...
  [ FFREE _ FST.st FSTP.st FUCOM FUCOMP  _ _ ]  is mod = 3 & reg_opcode = {0 to 7}
  [ FADDP _ FUBSRP FDIVRP FMULP _ FSUBP FDIVP ] is mod = 3 & reg_opcode = {0 to 7}
  FCOMPP    is DE; mod = 3 & reg_opcode = 3 & r_m = 1
  FSTSW.AX  is DF; mod = 3 & reg_opcode = 4 & r_m = 0
```

This next group of floating-point patterns define suffixes that we use on other opcodes, not actual opcodes.

```
<pattern specs for other patterns>+= (<-U U->) [<-D->]
patterns
  .STi      is DD; mod = 3
  Fstack is any of [ .ST.STi .STi.St P.STi.ST ], which is [ D8 DC DE ]; mod = 3
  Fint   is any of [.I32 .I16], which is [DA DE]
  Fmem   is any of [.R32 .R64], which is [D8 DC]
  FlsI   is any of [.lsI16 .lsI32], which is [DF DB]
  FlsR   is any of [.lsR32 .lsR64], which is [D9 DD]
```

Arithmetic instructions

[*] There are eight arithmetic instructions which have many different modes and which are all treated alike. The regular modes are shown in the upper left corners of Figures [<-] and [<-]; the immediate modes are the ``group 1'' instructions (denoted here by `arithI`). This is the only part of the Intel specification we were able to factor very well, but it does give us 112 constructors in just a dozen lines, so it is worth doing.

```
<arithmetic constructors>= (<-U U->)
constructors
  arith^"iAL"    i8!          is      arith & AL.Ib ; i8
  arith^"iAX"    i16!         is  ow; arith & eAX.Iv; i16
  arith^"iEAX"   i32!         is  od; arith & eAX.Iv; i32
  arithI^"b"     Eaddr, i8!   is      (Eb.Ib; Eaddr) & arithI; i8
  arithI^"w"     Eaddr, i16!  is  ow; (Ev.Iv; Eaddr) & arithI; i16
  arithI^"d"     Eaddr, i32!  is  od; (Ev.Iv; Eaddr) & arithI; i32
  arithI^ov^"b"  Eaddr, i8!   is  ov; (Ev.Ib; Eaddr) & arithI; i8
  arith^"mrb"    Eaddr, reg8  is      arith & Eb.Gb; Eaddr & reg_opcode = reg8 ...
  arith^"mr"^ov  Eaddr, reg   is  ov; arith & Ev.Gv; Eaddr & reg_opcode = reg ...
  arith^"rmb"    reg8, Eaddr  is      arith & Gb.Eb; Eaddr & reg_opcode = reg8 ...
  arith^"rm"^ov  reg, Eaddr   is  ov; arith & Gv.Ev; Eaddr & reg_opcode = reg ...

<32-bit constructors>+= (<-U) [<-D->]
arith^"iEAX" arithI^"d" arith^"mr"^od arith^"rm"^od
```

Other instructions (in alphabetical order)

Trying to factor the non-arithmetic instructions proved a thankless task, so we've given almost all instructions merely in alphabetical order as they appear in Chapter 25 of the Pentium manual. There's a little bit of local factoring, as with

some bit operations.

It's not really appropriate to try to explain this part of the specification; the best way to read this section by comparing with the alphabetical section of the Pentium architecture manual **[cite intel:pentium]**. To generate a checker by the end of the millenium, we divide the spec into four parts and check each individually.

```
<alphabetical constructors>= (<-U)
<alphabetical constructors A-F>
<alphabetical constructors G-K>
<alphabetical constructors L-Q>
<alphabetical constructors R>
<alphabetical constructors S-Z>
```

```
<alphabetical constructors A-F>= (<-U U->) [D->]
AAA
AAD is AAD; i8 = 10
AAM is AAM; i8 = 10
AAS
# ADC, ADD, AND are in arith group
ARPL     Eaddr, reg16  is  ARPL;      Eaddr & reg_opcode = reg16 ...
```

```
<32-bit constructors>+= (<-U) [<-D->]
AAA AAD AAM AAS
```

```
<names of pentium constructors>= (U->) [D->]
AAA AAD AAM AAS ARPL
```

Note that ARPL requires a 16-bit register, i.e. `%ax`, `%bx`, for its second operand.

The ``short'' variant of BOUND (boundw) requires a 16-bit register for its first operand.

```
<alphabetical constructors A-F>+= (<-U U->) [<-D->]
constructors
BOUND^ov reg, Mem    is  ov; BOUND; Mem   & reg_opcode = reg ...
BSF^ov   reg, Eaddr  is  ov; BSF;   Eaddr & reg_opcode = reg ...
BSR^ov   reg, Eaddr  is  ov; BSR;   Eaddr & reg_opcode = reg ...
BSWAP    r32         is  BSWAP & ... r32
BT^ov    Eaddr, reg  is  ov; BT;    Eaddr  & reg_opcode = reg ...
BTi^ov   Eaddr, i8!  is  ov; (grp8; Eaddr) & BTi;  i8
BTC^ov   Eaddr, reg  is  ov; BTC;   Eaddr  & reg_opcode = reg ...
BTCi^ov  Eaddr, i8!  is  ov; (grp8; Eaddr) & BTCi; i8
BTR^ov   Eaddr, reg  is  ov; BTR;   Eaddr  & reg_opcode = reg ...
BTRi^ov  Eaddr, i8!  is  ov; (grp8; Eaddr) & BTRi; i8
BTS^ov   Eaddr, reg  is  ov; BTS;   Eaddr  & reg_opcode = reg ...
BTSi^ov  Eaddr, i8!  is  ov; (grp8; Eaddr) & BTSi; i8
```

```
<names of pentium constructors>+= (U->) [<-D->]
BOUND^ov BSF^ov BSR^ov BSWAP BT^ov BTi^ov BTC^ov BTCi^ov BTR^ov BTRi^ov BTS^ov BTSi^ov
```

```
<32-bit constructors>+= (<-U) [<-D->]
BOUND^od BSF^od BSR^od BSWAP BT^od BTi^od BTC^od BTCi^od BTR^od BTRi^od BTS^od BTSi^od
```

To deal with relative displacements, we set up constructors to compute them. The displacements are relative to the *end* of the word.

```
<constructors for displacements>= (<-U U->)
constructors
  rel8  reloc : Rel8  { reloc = L + i8!  } is i8;  L: epsilon
  rel16 reloc : Rel16 { reloc = L + i16! } is i16; L: epsilon
  rel32 reloc : Rel32 { reloc = L + i32! } is i32; L: epsilon
```

```
<32-bit constructors>+= (<-U) [<-D->]
rel32
```

```
<alphabetical constructors A-F>+= (<-U U->) [<-D->]
CALL.Jv^ow reloc    is  ow; CALL.Jv; rel16(reloc)
CALL.Jv^od reloc    is  od; CALL.Jv; rel32(reloc)
CALL.Ep^ov Mem      is  ov; (grp5; Mem) & CALL.Ep
```

```
CALL.aP^ow CS":" IP is  ow; CALL.aP; i16 = CS; i16 = IP
CALL.aP^od CS":" IP is  od; CALL.aP; i16 = CS; i32 = IP
CALL.Ev^ov Mem      is  ov; (grp5; Mem) & CALL.Ev
CBW  is ow; CBW
CWDE is od; CBW
CLC
CLD
CLI
CLTS
CMC
```

The Linux assembler doesn't support multiple segments, so the CALL opcodes that take a code segment and offset are discarded when generating assembly code for a Linux assembler.

<pentium-linux.spec>= **(U-> U->) [D->]**

```
discard CALL.aP^ow CALL.aP^od
```

<alphabetical constructors A-F>+= **(<-U U->) [<-D->]**

```
# CMP  is in the arith group
CMPSB^av    is av; CMPSB
CMPSv^ov^av is (av; ov | ov; av); CMPSv
CMPXCHG.Eb.Gb    Eaddr, reg is     CMPXCHG.Eb.Gb; Eaddr & reg_opcode = reg ...
CMPXCHG.Ev.Gv^ov Eaddr, reg is ov; CMPXCHG.Ev.Gv; Eaddr & reg_opcode = reg ...
CMPXCHG8B        Mem        is     (grp9; Mem) & CMPXCHG8B
CPUID
CWD  is ow; CWDQ
CDQ  is od; CWDQ
```

<32-bit constructors>+= **(<-U) [<-D->]**

```
CALL.Jv^od CALL.Ep^od CALL.Ev^od CALL.aP^od CBW CWDE CLC CLD CLI CLTS CMC
CMPSv^od^av CMPXCHG.Ev.Gv^od CWD CDQ
```

CMPXCHG8B and CPUID are Pentium instructions so we can't check them on a 486.

<names of pentium constructors>+= **(U->) [<-D->]**

```
CMPXCHG8B CPUID CWD
```

<alphabetical constructors A-F>+= **(<-U U->) [<-D->]**

```
DAA
DAS
DEC.Eb     Eaddr  is      (grp4; Eaddr) & DEC.Eb
DEC.Ev^ov  Eaddr  is  ov; (grp5; Eaddr) & DEC.Ev
DEC^ov     r32    is  ov; DEC & r32
DIV^"AL"   Eaddr  is      (grp3.Eb; Eaddr) & DIV.AL.eAX
DIV^"AX"   Eaddr  is  ow; (grp3.Ev; Eaddr) & DIV.AL.eAX
DIV^"eAX"  Eaddr  is  od; (grp3.Ev; Eaddr) & DIV.AL.eAX
```

<32-bit constructors>+= **(<-U) [<-D->]**

```
DAA DAS DEC.Ev^od DEC^od DIV^"eAX"
```

<names of pentium constructors>+= **(U->) [<-D->]**

```
DAA DAS DEC.Eb
```

<alphabetical constructors A-F>+= **(<-U U->) [<-D->]**

```
ENTER   i16, i8!  is  ENTER; i16; i8
F2XM1
```

<32-bit constructors>+= **(<-U) [<-D->]**

```
ENTER F2XM1
```

<names of pentium constructors>+= **(U->) [<-D->]**

```
ENTER F2XM1
```

The first use of the left-hand side ellipsis is used in the definition of FADD^Fstack. The left ellipsis denotes that the pattern (FADD & r_m = idx) must be a legal suffix of the pattern Fstack.

<alphabetical constructors A-F>+= **(<-U U->) [<-D->]**

```
FABS
```

```
FADD^Fmem   Mem  is  Fmem; Mem & FADD ...
FADD^Fstack idx  is  Fstack & ... (FADD & r_m = idx)

<floating-point constructors>= (<-U <-U U->) [D->]
FABS FADD^Fmem FADD^Fstack

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FIADD^Fint  Mem  is  Fint; Mem & FIADD ...
FBLD        Mem  is  DF; Mem & FBLD
FBSTP       Mem  is  DF; Mem & FBSTP

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FCHS

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FCHS

<synthetics with WAIT>= (U->) [D->]
FCLEX            is  WAIT; FCLEX

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FNCLEX           is  FCLEX

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns FCOMs is FCOM | FCOMP

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
   constructors
FCOMs^Fmem     Mem  is  Fmem; Mem & FCOMs ...           # includes FICOM, FICOMP
FCOMs^.ST.STi  idx  is  .ST.STi & ... (FCOMs & r_m = idx)
FCOMPP
FCOS

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FCOMs^Fmem FCOMs^.ST.STi FCOMPP FCOS

<pentium-linux.spec>+= (U-> U->) [<-D->]
discard FCOMs^.ST.STi

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FDECSTP
FDIV^Fmem     Mem  is  Fmem; Mem & FDIV ...
FDIV^Fstack   idx  is  Fstack & ... (FDIV & r_m = idx)
FDIVR^Fmem    Mem  is  Fmem; Mem & FDIVR ...
FDIVR^Fstack  idx  is  Fstack & ... (FDIVR & r_m = idx)

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FDECSTP FDIV^Fmem FDIV^Fstack FDIVR^Fmem FDIVR^Fstack

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FFREE         idx  is  DD; FFREE & r_m = idx

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FICOM^Fint   Mem is Fint; Mem & FICOM
FICOMP^Fint  Mem is Fint; Mem & FICOMP
FILD^FlsI Mem is FlsI; Mem & FILD
FILD64    Mem is DF;  Mem & FLD.ext ...
FINIT

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns FISTs is FIST | FISTP

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
   constructors
FISTs^FlsI  Mem  is  FlsI; Mem & FISTs
FISTP64     Mem  is  DF;   Mem & FSTP.ext

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FICOM^Fint FICOMP^Fint FICOM16 FICOM32 FICOMP16 FICOMP32
```

```
FILD^FlsI FILD64 FINIT FISTs^FlsI FISTP64
```

EDIT ME? There is no obvious way to reference indirectly the stack pointer, i.e., (%esp), because the encoding of this addressing mode is an escape that indicates an sib byte follows the mod-rm byte. The only way to construct this address is to build a sib byte that ignores its index field. Page 26-7 specifies how to do this. Luckily, we only need (%esp) for a few floating-point instructions that use the address on the top of the stack and then pops it off. Although this addressing mode is technically an Eaddr, we specify it individually because virutally no other instruction uses it.

```
<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FLD^FlsR    Mem  is  FlsR; Mem & FLD
FLD80       Mem  is  DB;   Mem & FLD.ext ...
FLD.STi     idx  is  D9;    mod = 3 & FLD & r_m = idx
Fconstants
FLDCW       Mem  is  D9;   Mem & FLDCW
FLDENV      Mem  is  D9;   Mem & FLDENV

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FLD^FlsR FLD80 FLD.STi Fconstants FLDCW FLDENV

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FMUL^Fmem   Mem  is  Fmem; Mem & FMUL ...
FMUL^Fstack idx  is  Fstack & ... (FMUL & r_m = idx)

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FMUL^Fmem FMUL^Fstack

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FNOP

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FNOP

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FPATAN
FPREM
FPREM1
FPTAN

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FPATAN FPREM FPREM1 FPTAN

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FRNDINT
FRSTOR Mem  is DD; Mem & FRSTOR

<floating-point constructors>+= (<-U <-U U->) [<-D->]

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FNSAVE Mem  is DD; Mem & FSAVE

<floating-point constructors>+= (<-U <-U U->) [<-D->]

<synthetics with WAIT>+= (U->) [<-D->]
FSAVE  Mem  is WAIT; FNSAVE(Mem)

<floating-point constructors>+= (<-U <-U U->) [<-D->]

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FSCALE
FSIN
FSINCOS
FSQRT

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns
FSTs is FST | FSTP
FSTs.st is FST.st | FSTP.st

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
```

```
   constructors
FSTs^FlsR     Mem  is  FlsR; Mem & FSTs
FSTP80        Mem  is  DB;   Mem & FSTP.ext
FSTs.st^.STi  idx  is  .STi & ... (FSTs.st & r_m = idx)
FSTCW         Mem  is  D9;   Mem & FSTCW

<synthetics with WAIT>+= (U->) [<-D->]
FNSTCW        Mem  is  WAIT; FSTCW(Mem)

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FSCALE FSIN FSINCOS FSQRT FSTs^FlsR FSTP80 FSTs.st^.STi FSTCW FNSTCW

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FSTENV        Mem  is  D9;   Mem & FSTENV

<synthetics with WAIT>+= (U->) [<-D->]
FNSTENV       Mem  is  WAIT; FSTENV(Mem)

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FSTENV FNSTENV

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FSTSW         Mem  is  DD;   Mem & FSTSW
FSTSW.AX

<synthetics with WAIT>+= (U->) [<-D->]
FNSTSW        Mem  is  WAIT; FSTSW(Mem)
FNSTSW.AX          is  WAIT; FSTSW.AX()

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FSTSW FSTSW.AX FNSTSW FNSTSW.AX

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FSUB^Fmem     Mem  is  Fmem; Mem & FSUB ...
FSUB^Fstack   idx  is  Fstack & ... (FSUB & r_m = idx)
FSUBR^Fmem    Mem  is  Fmem; Mem & FSUBR ...
FSUBR^Fstack  idx  is  Fstack & ... (FSUBR & r_m = idx)

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FSUB^Fmem FSUB^Fstack FSUBR^Fmem FSUBR^Fstack

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FTST

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FTST

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns FUCOMs is FUCOM | FUCOMP

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
   constructors
FUCOMs  idx   is  DD; FUCOMs & r_m = idx
FUCOMPP

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FUCOMs FUCOMPP

<synthetics with WAIT>+= (U->) [<-D]
FWAIT is WAIT

<floating-point constructors>+= (<-U <-U U->) [<-D->]
FWAIT

<alphabetical constructors A-F>+= (<-U U->) [<-D->]
FXAM
FXCH idx  is  FXCH & ... r_m = idx
FXTRACT
```

```
<floating-point constructors>+= (<-U <-U U->) [<-D->]
FXAM FXCH FXTRACT

<alphabetical constructors A-F>+= (<-U U->) [<-D]
FYL2X
FYL2XP1

<floating-point constructors>+= (<-U <-U U->) [<-D]
FYL2X FYL2XP1

<names of pentium constructors>+= (U->) [<-D]
FIADD^Fint FBLD FBSTP FNCLEX FCOS FDECSTP FDIVR^Fmem FIDIV^Fint FIDIVR^Fint
FICOM^Fint FICOMP^Fint FILD64 FINCSTP FINIT FISTP64 FLD80 Fconstants
FLDCW FLDENV FIMUL^Fint FNOP FPATAN FPREM FPREM1 FPTAN FRNDINT
FRSTOR FSCALE FSIN FSINCOS FSQRT FSUBR^Fmem FISUB^Fint FISUBR^Fint
FTST FUCOMs FUCOMPP FXAM FXCH FXTRACT FYL2X FYL2XP1

<alphabetical constructors G-K>= (<-U U->) [D->]
HLT

<32-bit constructors>+= (<-U) [<-D->]
HLT

<alphabetical constructors G-K>+= (<-U U->) [<-D->]
IDIV        Eaddr  is      (grp3.Eb; Eaddr) & IDIV.AL.eAX
IDIV^"AX"   Eaddr  is  ow; (grp3.Ev; Eaddr) & IDIV.AL.eAX
IDIV^"eAX"  Eaddr  is  od; (grp3.Ev; Eaddr) & IDIV.AL.eAX
IMULb        Eaddr            is      (grp3.Eb;    Eaddr) & IMUL.AL.eAX
IMUL^ov      Eaddr            is  ov; (grp3.Ev;    Eaddr) & IMUL.AL.eAX
IMULrm^ov    reg, Eaddr       is  ov; IMUL.Gv.Ev; Eaddr & reg_opcode = reg ...
IMUL.Ib^ov   reg, Eaddr, i8!  is  ov; IMUL.Ib;    Eaddr & reg_opcode = reg ... ; i8
IMUL.Iv^"w"  reg, Eaddr, i16! is  ow; IMUL.Iv;    Eaddr & reg_opcode = reg ... ; i16
IMUL.Iv^"d"  reg, Eaddr, i32! is  od; IMUL.Iv;    Eaddr & reg_opcode = reg ... ; i32
IN.AL.Ib       i8! is      IN.AL.Ib;  i8
IN.eAX.Ib^ov   i8! is  ov; IN.eAX.Ib; i8
IN.AL.DX
IN.eAX.DX^ov       is  ov; IN.eAX.DX
INC.Eb     Eaddr  is      (grp4; Eaddr) & INC.Eb
INC.Ev^ov  Eaddr  is  ov; (grp5; Eaddr) & INC.Ev
INC^ov     r32    is  ov; INC & r32
INSB
INSv^ov   is  ov; INSv
INT3
INT.Ib i8! is  INT.Ib; i8
INTO
INVD
INVLPG  Mem  is  (grp7; Mem) & INVLPG
IRET

<32-bit constructors>+= (<-U) [<-D->]
IDIV^"eAX" IMUL^od IMULrm^od IMUL.Iv^"d" IN.eAX.DX^od INC.Ev^od INC^od
INSv^od INT3 INT.Ib INTO INVD INVLPG IRET

<alphabetical constructors G-K>+= (<-U U->) [<-D->]
Jb^cond    reloc   is      Jb & cond;     rel8(reloc)
Jv^cond^ow reloc   is  ow; Jv & ... cond; rel16(reloc)
Jv^cond^od reloc   is  od; Jv & ... cond; rel32(reloc)
JMP.Jb     reloc   is      JMP.Jb; rel8(reloc)
JMP.Jv^ow  reloc   is  ow; JMP.Jv; rel16(reloc)
JMP.Jv^od  reloc   is  od; JMP.Jv; rel32(reloc)
JMP.Ap^ow  CS, IP  is  ow; JMP.Ap; i16 = CS; i16 = IP
JMP.Ap^od  CS, IP  is  od; JMP.Ap; i16 = CS; i32 = IP
JMP.Ev^ov Eaddr    is  ov; (grp5; Eaddr) & JMP.Ev
JMP.Ep^ov Mem      is  ov; (grp5; Mem  ) & JMP.Ep

<32-bit constructors>+= (<-U) [<-D->]
Jv^cond^od JMP.Jv^od JMP.Ap^od JMP.Ev^od JMP.Ep^od
```

```
<pentium-linux.spec>+= (U-> U->) [<-D->]
discard JMP.Ap^ow  JMP.Ap^ov

<alphabetical constructors L-Q>= (<-U U->) [D->]
LAHF
LAR^ov  reg, Eaddr  is  ov; LAR; Eaddr & reg_opcode = reg ...

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns lfp is LDS | LES | LFS | LGS | LSS

<alphabetical constructors G-K>+= (<-U U->) [<-D]
   constructors
lfp^ov  reg, Mem    is  ov; lfp; Mem & reg_opcode = reg ...
LEA^ov  reg, Mem    is  ov; LEA; Mem & reg_opcode = reg ...
LEAVE
LGDT  Mem   is  (grp7; Mem) & LGDT
LIDT  Mem   is  (grp7; Mem) & LIDT
LLDT  Eaddr is  (grp6; Eaddr) & LLDT
LMSW  Eaddr is  (grp7; Eaddr) & LMSW
LOCK
LODSB
LODSv^ov is ov; LODSv

<pattern specs for other patterns>+= (<-U U->) [<-D->]
    patterns LOOPs is LOOP | LOOPE | LOOPNE

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
    constructors
LOOPs^ov  reloc  is  ov; LOOPs; rel8(reloc)
LSL^ov  reg, Eaddr  is  ov; LSL; Eaddr & reg_opcode = reg ...
LTR  Eaddr  is  (grp6; Eaddr) & LTR

<32-bit constructors>+= (<-U) [<-D->]
LAHF LAR^ov lfp^ov LEA^ov LEAVE LGDT LIDT LLDT LMSW LOCK LODSB LODSv^ov

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
MOV^"mrb"    Eaddr, reg    is      MOV & Eb.Gb; Eaddr & reg_opcode = reg ...
MOV^"mr"^ov  Eaddr, reg    is  ov; MOV & Ev.Gv; Eaddr & reg_opcode = reg ...
MOV^"rmb"    reg, Eaddr    is      MOV & Gb.Eb; Eaddr & reg_opcode = reg ...
MOV^"rm"^ov  reg, Eaddr    is  ov; MOV & Gv.Ev; Eaddr & reg_opcode = reg ...
MOV.Ew.Sw    Mem, sr16     is  ow; MOV.Ew.Sw;   Mem   & reg_opcode = sr16 ...
MOV.Sw.Ew    Mem, sr16     is      MOV.Sw.Ew;   Mem   & reg_opcode = sr16 ...
# assume 32-bit address mode
MOV.AL.Ob      offset      is      MOV.AL.Ob;  i32 = offset
MOV.eAX.Ov^ov  offset      is  ov; MOV.eAX.Ov; i32 = offset
MOV.Ob.AL      offset      is      MOV.Ob.AL;  i32 = offset
MOV.Ov.eAX^ov  offset      is  ov; MOV.Ov.eAX; i32 = offset
MOVib          r8,  i8!    is      MOVib & r8; i8
MOViw          r32, i16!   is  ow; MOViv & r32; i16
MOVid          r32, i32!   is  od; MOViv & r32; i32
MOV.Eb.Ib      Eaddr, i8!  is      MOV.Eb.Ib; Eaddr & reg_opcode = 0 ...; i8
MOV.Ev.Iv^ow   Eaddr, i16! is  ow; MOV.Ev.Iv; Eaddr & reg_opcode = 0 ...; i16
MOV.Ev.Iv^od   Eaddr, i32! is  od; MOV.Ev.Iv; Eaddr & reg_opcode = 0 ...; i32
MOV.Cd.Rd      cr, reg     is  MOV.Cd.Rd; mod = 3 & r_m = reg & reg_opcode = cr
MOV.Rd.Cd      reg, cr     is  MOV.Rd.Cd; mod = 3 & r_m = reg & reg_opcode = cr
MOV.Dd.Rd      dr, reg     is  MOV.Dd.Rd; mod = 3 & r_m = reg & reg_opcode = dr
MOV.Rd.Dd      reg, dr     is  MOV.Rd.Dd; mod = 3 & r_m = reg & reg_opcode = dr
MOVSB
MOVSv^ov is ov; MOVSv
MOVSX.Gv.Eb^ov    r32, Eaddr    is ov; MOVSX.Gv.Eb; Eaddr & reg_opcode = r32 ...
MOVSX.Gv.Ew       r16, Eaddr    is     MOVSX.Gv.Ew; Eaddr & reg_opcode = r16 ...
MOVZX.Gv.Eb^ov    r32, Eaddr    is ov; MOVZX.Gv.Eb; Eaddr & reg_opcode = r32 ...
MOVZX.Gv.Ew       r16, Eaddr    is     MOVZX.Gv.Ew; Eaddr & reg_opcode = r16 ...
MUL.AL      Eaddr  is      (grp3.Eb; Eaddr) & MUL.AL.eAX
MUL.AX^ov   Eaddr  is  ov; (grp3.Ev; Eaddr) & MUL.AL.eAX

<32-bit constructors>+= (<-U) [<-D->]
```

```
MOV^"mrb" MOV^"mr"^ov MOV^"rmb" MOV^"rm"^ov
MOV.Ew.Sw MOV.Sw.Ew MOV.AL.Ob MOV.eAX.Ov^ov MOV.Ob.AL
MOV.Ov.eAX^ov MOVib MOViw MOVid MOV.Eb.Ib MOV.Ev.Iv^ow
MOV.Ev.Iv^od MOVSB MOVSv^ov MOVSX.Gv.Eb^od
MOVSX.Gv.Ew MOVZX.Gv.Eb^od MOVZX.Gv.Ew MUL.AL MUL.AX^ov

<pentium-linux.spec>+= (U-> U->) [<-D->]
discard  MOVSX.Gv.Ew MOVZX.Gv.Eb^ov MOVZX.Gv.Ew MOV.Ew.Sw

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
NEGb    Eaddr  is      (grp3.Eb; Eaddr) & NEG
NEG^ov  Eaddr  is  ov; (grp3.Ev; Eaddr) & NEG
NOP
NOTb    Eaddr  is      (grp3.Eb; Eaddr) & NOT
NOT^ov  Eaddr  is  ov; (grp3.Ev; Eaddr) & NOT

<32-bit constructors>+= (<-U) [<-D->]
NEGb NEG^ov NOP NOTb NOT^ov

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
# OR is in the arith group
OUT.Ib.AL       i8! is      OUT.Ib.AL;  i8
OUT.Ib.eAX^ov   i8! is  ov; OUT.Ib.eAX; i8
OUT.DX.AL
OUT.DX.eAX^ov       is  ov; OUT.DX.eAX
OUTSB
OUTSv^ov is ov; OUTSv

<32-bit constructors>+= (<-U) [<-D->]
OUT.Ib.AL OUT.Ib.eAX^ov OUT.DX.AL OUT.DX.eAX^ov OUTSB OUTSv^ov

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
POP.Ev^ov   Mem   is  ov; POP.Ev; Mem & reg_opcode = 0 ...
POP^ov      r32   is  ov; POP & r32

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns POPs is POP.ES | POP.SS | POP.DS | POP.FS | POP.GS
            POPv is POPA | POPF

<alphabetical constructors L-Q>+= (<-U U->) [<-D->]
   constructors
POPs
POPv^ov  is  ov; POPv
PUSH.Ev^ov  Eaddr  is  ov; (grp5; Eaddr) & PUSH.Ev
PUSH^ov     r32    is  ov; PUSH & r32
PUSH.Ib     i8!    is      PUSH.Ib; i8
PUSH.Iv^ow  i16!   is  ow; PUSH.Iv; i16
PUSH.Iv^od  i32!   is  od; PUSH.Iv; i32

<pattern specs for other patterns>+= (<-U U->) [<-D->]
   patterns PUSHs is PUSH.CS | PUSH.SS | PUSH.DS | PUSH.ES | PUSH.FS | PUSH.GS
            PUSHv is PUSHA | PUSHF

<alphabetical constructors L-Q>+= (<-U U->) [<-D]
   constructors
PUSHs
PUSHv^ov  is  ov; PUSHv

<32-bit constructors>+= (<-U) [<-D->]
POPs POPv^ov  PUSH.Ev^ov PUSH^ov PUSH.Ib PUSH.Iv^ow PUSH.Iv^od PUSHs PUSHv^ov

<alphabetical constructors R>= (<-U U->) [D->]
# ROL ROR RCL RCR SHLSAL SHR SAR
rot^bshifts    Eaddr       is     (bshifts; Eaddr) & rot
rot^vshifts^ov Eaddr       is ov; (vshifts; Eaddr) & rot
rot^B.Eb.Ib    Eaddr, i8!  is     (B.Eb.Ib; Eaddr) & rot; i8
rot^B.Ev.Ib^ov Eaddr, i8!  is ov; (B.Ev.Ib; Eaddr) & rot; i8
```

```
<32-bit constructors>+= (<-U) [<-D->]
rot^bshifts rot^vshifts^ov rot^B.Ev.Ib^ov

<alphabetical constructors R>+= (<-U U->) [<-D]
RDMSR
REP
REPNE
RET
RET.far
RET.Iw      i16  is RET.Iw; i16
RET.far.Iw  i16  is RET.far.Iw; i16
RSM

<32-bit constructors>+= (<-U) [<-D->]
RDMSR REP REPNE RET RET.far RET.Iw RET.far.Iw RSM

<pentium-linux.spec>+= (U-> U->) [<-D->]
discard RDMSR

<alphabetical constructors S-Z>= (<-U U->) [D->]
SAHF
# SAL SAR SHL SR above with rot
# SBB is in the arith group
SCASB
SCASv^ov  is  ov; SCASv
## SETb^cond Mem  is  SETb & ... cond; Mem
SETb^cond Eaddr  is SETb & ... cond; Eaddr
SGDT  Mem  is (grp7; Mem) & SGDT
SIDT  Mem  is (grp7; Mem) & SIDT

<32-bit constructors>+= (<-U) [<-D->]
SCASB SCASv^ov SETb^cond SGDT SIDT

<pattern specs for other patterns>+= (<-U U->) [<-D]
  patterns shdIb is SHRD.Ib | SHLD.Ib
           shdCL is SHRD.CL | SHLD.CL

<alphabetical constructors S-Z>+= (<-U U->) [<-D->]
  constructors
shdIb^ov  Eaddr, reg, count  is  ov; shdIb; Eaddr & reg_opcode = reg ... ; i8 = count
shdCL^ov  Eaddr, reg, "CL"   is  ov; shdCL; Eaddr & reg_opcode = reg ...
SLDT Eaddr  is  (grp6; Eaddr) & SLDT
SMSW Eaddr  is  (grp7; Eaddr) & SMSW
STC
STD
STI
STOSB
STOSv^ov is ov; STOSv
STR  Mem  is  (grp6; Mem) & STR
# SUB is in the arith group

<32-bit constructors>+= (<-U) [<-D->]
shdIb^ov shdCL^ov SLDT SMSW STC STD STI STOSB STOSv^ov STR

<alphabetical constructors S-Z>+= (<-U U->) [<-D->]
TEST.AL.Ib      i8          is     TEST.AL.Ib; i8
TEST.eAX.Iv^ow  i16         is ow; TEST.eAX.Iv; i16
TEST.eAX.Iv^od  i32         is od; TEST.eAX.Iv; i32
TEST.Eb.Ib      Eaddr, i8   is     (grp3.Eb; Eaddr) & TEST.Ib.Iv; i8
TEST.Ew.Iw      Eaddr, i16  is ow; (grp3.Ev; Eaddr) & TEST.Ib.Iv; i16
TEST.Ed.Id      Eaddr, i32  is od; (grp3.Ev; Eaddr) & TEST.Ib.Iv; i32
TEST.Eb.Gb      Eaddr, reg  is     TEST.Eb.Gb; Eaddr & reg_opcode = reg ...
TEST.Ev.Gv^ov   Eaddr, reg  is ov; TEST.Ev.Gv; Eaddr & reg_opcode = reg ...

<32-bit constructors>+= (<-U) [<-D->]
TEST.AL.Ib TEST.eAX.Iv^ow TEST.eAX.Iv^od TEST.Eb.Ib TEST.Ew.Iw
TEST.Ed.Id TEST.Eb.Gb TEST.Ev.Gv^ov
```

```
<alphabetical constructors S-Z>+= (<-U U->) [<-D->]
VERR  Eaddr  is (grp6; Eaddr) & VERR
VERW  Eaddr  is (grp6; Eaddr) & VERW

<32-bit constructors>+= (<-U) [<-D->]
VERR VERW

<alphabetical constructors S-Z>+= (<-U U->) [<-D->]
WAIT
WBINVD
WRMSR

<32-bit constructors>+= (<-U) [<-D->]
WAIT WBINVD WRMSR

<pentium-linux.spec>+= (U-> U->) [<-D]
discard WRMSR

<alphabetical constructors S-Z>+= (<-U U->) [<-D]
XADD.Eb.Gb     Eaddr, reg  is     XADD.Eb.Gb; Eaddr & reg_opcode = reg ...
XADD.Ev.Gv^ov  Eaddr, reg  is ov; XADD.Ev.Gv; Eaddr & reg_opcode = reg ...
XCHG^"eAX"^ov  r32         is ov; XCHG & r32
XCHG.Eb.Gb     Eaddr, reg  is     XCHG.Eb.Gb; Eaddr & reg_opcode = reg ...
XCHG.Ev.Gv^ov  Eaddr, reg  is ov; XCHG.Ev.Gv; Eaddr & reg_opcode = reg ...
XLATB is XLAT
# XOR is in the arith group

<32-bit constructors>+= (<-U) [<-D]
XADD.Eb.Gb XADD.Ev.Gv^ov XCHG^"eAX"^ov XCHG.Eb.Gb XCHG.Ev.Gv^ov XLATB
```

Synthetic instructions

The only synthetics we've identified are those that are preceded by a `WAIT` instruction, which we've included in the alphabetical list above, since they're all given together in Chapter 25.

The Gnu-Linux assembler does not recognize instructions prefixed by `WAIT` as synthetics. For example, `wait; fclex` disassembles as `fclex`; `fclex` disassembles as `fnclex`.

```
<pentium-synth.spec>=
constructors
  <synthetics with WAIT>
```

`mld` needs some synthetics to help it deal with overloaded operators. This is a hack but better than writing them all out by hand.

```
<mld-pentium.spec>= [D->]
constructors
  _call_l reloc  : Function is  call_jvod(reloc)
  _call_rm Mem : Function is  call_evod(Mem)
  callfn Function is Function
```

The following constructor is used to emit relocatable addresses. It is the same as that defined for the MIPS and SPARC.

```
<mld-pentium.spec>+= [<-D]
fields of addrtoken (32) addr32 0:31
placeholder for addrtoken is addr32 = 7
constructors
  emit_raddr reloc is addr32 = reloc
```

Assembly Specification for Gnu-Linux Assembler

Some assemblers overload instruction names, using context to determine which instruction is meant. For example, the Pentium add can mean any of five different instructions, depending on the sizes and locations of the operands. The toolkit cannot do this kind of overloading; it must use different names for different instructions (constructors). The reason is that the toolkit must generate a different encoding procedure for each instruction, and in most programming languages, different procedures must have different names. Even in languages that do support overloading, we might not be able to use the same name, because the name-resolution mechanisms used in programming languages are typically type-based and quite different from what an assembler uses (LR parsing).

We solve this problem by requiring each constructor to have a different name. Typically, the specification writer distinguishes variants by adding suffixes to the base name of the constructor. For example, the Pentium add instructions include constructors called `addb`, `addib`, `addiowb`, `addmrb`, and `addrmb`. To get from these names back to assembly language, we have to define an appropriate mapping. These mappings are defined separately from the main specification, because different vendors use different syntaxes for their assembly languages.

An assembly specification includes three parts: a mapping of constructor names to assembly opcodes, the assembly format for each constructor operand, and the assembly syntax for each constructor. We use the specification for the assembly language supported by the Gnu-Linux assembler to illustrate assembly specifications. First, we describe assembly name mappings, then operand format and constructor assembly syntax.

Assembly names for opcodes

A constructor's name may contain multiple parts derived from pattern names and constant strings. Each part may or may not contribute to the assembly name. For example, the constructor name `add^"mrb"` contains two parts, the first derived from the pattern `add` and the second from the string `"mrb"`. The assembly name for this constructor is `addb`, so the pattern `add` contributes its name and the suffix `"mrb"` is mapped to the string `"b"`. This example illustrates how the constructor `addmrb` has a more specific name to disambiguate it from other overloaded variants of `addb`.

`assembly opcode` introduces mappings from complete constructor names (strings) to their assembly names (strings). `assembly component` introduces mappings from parts of constructor names to their assembly names. We provide a component-wise mapping, because it improves factoring of assembly names among constructors that share common suffixes and prefixes. For example, the suffix `B.Eb.Ib` always maps to `b` in every constructor name where it appears.

There is redundancy in the mapping of constructor names, so we use a regular expression syntax to group related names. The regular expression syntax is the same as the syntax for C-shell ``globbing'' expressions **[cite joy:c-shell]**. If a complete name mapping exists for a constructor name, it is applied first. If no complete mapping exists, mappings are applied individually to *each* part of a constructor's name and the resulting strings are concatenated into the complete assembly name. For example, the mappings applied to the parts of the constructor name `add^"mrb"` are:

<example mappings>=
```
assembly component
    add     is  add
    {"mrb","rmb"} is b
```

The first rule maps `add` to itself and the second maps any string that matches `mrb` or `rmb` to `b`. The least general rule that matches a string is applied.

It is often useful to define a default mapping, i.e., for the pattern ``*''. On the Pentium, for example, most constructor names do not map directly to assembly names, so the default maps a name to the empty string. *This is wrong.*

<pentium-linux-default.spec>= **(U->)**
```
assembly component
    {Indir,{Disp*},Abs32,Reg,{*Index*},E,rel{8,16,32}} is ""
    {*}  is     $1
```

On the MIPS and SPARC, however, most constructor names map are assembly names, so the default maps a name to itself, i.e., `assembly component {*} is $1`.

The remaining rules specify all the assembly names for the Pentium constructors and illustrate use of the C-shell globbing expressions. In globbing expressions, ``*'' matches any string; any character and ``.'' matches itself. The concatenation operator is implicit, so adjacent characters are concatenated. Alternatives are comma-separated lists of strings delimited by ``{'' and ``}''.

We provide one extension to the C-shell syntax: expressions in curly braces may be referenced on the right-hand side by $*n*, where *n* is the *n*-th braced expression on the left-hand side. For example, the first rule specifies that the suffixes `ow` and `aw` map to the assembly name `w`, and `od` and `ad` map to `l`. The rest of these rules map all the suffixes used in constructor names to their assembly names.

<pentium-linux-names.spec>= **(U->) [D->]**
```
assembly component
    {iAL,AL}                is b
    {iAX,AX}                is w
    {iEAX,eAX}              is l
    {o,a}d                  is l
    {o,a}w                  is w
    {.I32,.R64,.lsI32,.lsR64} is l
```

```
    {.I16,.R32,.lsR32}      is s
    .lsI16                  is w
    b.*                     is b
    {b,w}                   is $1
    d                       is l
    B.{Eb.1,Eb.CL,Eb.Ib,Ev.Ib}   is b
    B.{Ev.1,Ev.CL}          is w
    {.STi,.ST.STi,.STi.St}  is ""
    P.STi.ST                is P
    .{O,NO,B,NB,Z,NZ,BE,NBE,S,NS,P,NP,L,NL,LE,NLE} is $1
```

Many constructor names contain suffixes that are mneumonics for the opcodes they represent. These suffixes are often eliminated in the assembly name. For example, the assembly names for the immediate opcodes `ADD.i` and `OR.i` are `ADD` and `OR`, respectively. The following rules truncate these suffixes.

```
<pentium-linux-names.spec>+= (U->) [<-D->]
assembly component
    {CALL}.*                is $1
    {CALL}l                 is $1
    CMPXCHG8B               is CMPXCHG
    {CMPXCHG,XADD,XCHG,TEST}.Eb.Gb is $1b
    CMPSv                   is CMPS
    {CMP*}.*                is $1
    {DEC,INC}.*             is $1
    {DIV}.*                 is $1
    {*}.st                  is $1
    {FLD,FSTP}80            is $1t
    {FLD,FSTP}.*            is $1
    {FILD,FISTP}.*          is $1
    {FICOM*}16              is $1s
    {FICOM*}32              is $1l
    {FILD,FISTP*}64         is $1ll
    {IDIV,IMUL}.*           is $1
    {IN,INT,J}.*            is $1
    JMP.Ep                  is lJMP
    {JMP}.*                 is $1
    MOV{.Eb.Ib,.AL.Ob,.Ob.AL} is MOVb
    {MOViv,MOV.Ev.Iv}       is MOV
    MOVSX.Gv.Ew             is MOVSwl
    {MOV.Ew.Sw,MOV.Sw.Ew} is MOVw
    {MOVS,MOVZ}X.Gv.Eb      is $1b
    {MOVSv,MOVSX.*}         is MOVS
    {MOV,MOVS}.*            is $1
    {MOVSX,MOVZX}.*         is $1
    MOVi{b,w}               is MOV$1
    MOVid                   is MOVl
    {*}.AX                  is $1
    {OUT.Ib.AL,OUT.DX.AL}   is OUTb
    {OUT,OUTS}.*            is $1
    {RET.far}*              is lRET
    {POP,PUSH,RET}.*        is $1
    {SCAS,STOS}v            is $1
    {SHRD,SHLD}.*           is $1
    SHRSAL                  is SHR
    TEST{.*.Ib,.Eb.*}       is TESTb
    TEST.*.Iw               is TESTw
    TEST.*.Id               is TESTl
    TEST.*                  is TEST
    {XADD*}.*               is $1
    {XCHG*}.*               is $1
    {*}i                    is $1
```

The remaining rules are for constructors with special assembly names.

```
<pentium-linux-names.spec>+= (U->) [<-D->]
assembly component
```

```
    {"mr","rm"}             is ""
    {"mrb","rmb"}           is b
    {*}64                   is $1
    {IDIV,DIV}"AL"          is $1
    {IDIV,DIV}"AX"          is $1
    {IDIV,DIV}"eAX"         is $1
    {IMULrm}                is IMUL
    INT3                    is INT
    FLD.ext                 is FLDLL
    {Jv,Jb}                 is J
    {INC}.Eb                is INCb
    {INS,LODS}v             is $1
    MUL.AL                  is MULb
    OUTSv                   is OUTS
    SHLSAL                  is SHL
    TEST.Ew.Iw              is TESTw
    TEST.Ed.Id              is TESTl
    SETb                    is SET
```

Component-wise mapping doesn't work for all names.

```
<pentium-linux-names.spec>+= (U->) [<-D->]
assembly opcode
  CALL.{Ev}{od}             is CALL
  CALL.{Jv,Ep}{od,ow}       is lCALL
  CALL.aP{od}               is CALL
  CMPSv{od,ow}ad            is CMPSl
  CMPSv{od,ow}aw            is CMPSw
  JMP.Epod                  is lJMP
  MOVSX.Gv.Ebod             is MOVSbl
  MOVSX.Gv.Ebow             is MOVSbw
  {ROL,ROR,RCL,RCR,SHR,SAR}{B.Ev.*}od is $1l
  {ROL,ROR,RCL,RCR,SHR,SAR}{B.Ev.*}ow is $1w
  SHLSAL{B.Ev.*}od          is SHLl
  SHLSAL{B.Ev.*}ow          is SHLw
  XCHGeAXow                 is XCHGw
  XCHGeAXod                 is XCHGl
```

Assembly formats for operands

`assembly operand` introduces mappings from operands to formatted strings that specify how to print the operands in assembly code. Operands may be fields, integer inputs, relocatable addresses, or constant strings. We use `printf`-style syntax for formatted strings.

The first `assembly operand` rule below specifies that immediate operands are prefixed by ``$'' and are printed as integers. The second rule specifies that the listed field inputs are prefixed by ``%'' and printed as strings, using the names provided in their `fieldinfo` declarations.

```
<pentium-linux-names.spec>+= (U->) [<-D->]
assembly operand
    [count i8 i16 i32]           is "$%d"
    [r32 sr16 r16 r8 base index] is "%%%s"
```

Some inputs are not declared as fields but should be printed in the same format as fields. For example, the input `reg` should be printed using the names associated with the field `base`. The following rule uses the optional `using` *field* clause to specify that `reg`, `reg8`, etc. should be printed using the fieldinfo associated with `base`.

```
<pentium-linux-names.spec>+= (U->) [<-D->]
assembly operand
    [reg reg8 sreg cr dr]       is "%%%s" using field base
```

Some operands are used implicitly in a constructor. For example, the constructor `OUT.DX.AL "dx","al"`, implicitly uses the `dx` and `al` registers as its operands. Like all other operands, their format is assembler dependent so we provide mappings for printing them in the Gnu-Linux format.

```
<pentium-linux-names.spec>+= (U->) [<-D]
assembly operand
```

```
    dx    is "%%dx"
    ax    is "%%ax"
```

Assembly syntax for constructors

The default assembly syntax for a constructor appears in the constructor's specification; an alternate syntax may be specified with `assembly syntax`. Providing assembly syntax in a constructor specification can help a specification writer or user read and identify a constructor, and it is concise when only one assembly syntax is required. An alternate syntax may be needed, however, if more than one assembly language is used on the target.

`assembly syntax` uses the same syntax as the `constructors` directive: a constructor name followed by a list of operands. The assembly-syntax specification must use the same set of operands that the constructor uses, but the operands may appear in any order and with any syntactic sugar. To reduce redundancy, we define new patterns to group constructors that share the same assembly syntax.

The Gnu-Linux assembly language reverses the order of all operands. The following directives reverse the order.

<pentium-linux-syntax.spec>= **(U->) [D->]**

```
assembly syntax
  arith^"iAL"     i8!, "%al"
  arith^"iAX"     i16!, "%ax"
  arith^"iEAX"    i32!, "%eax"
  DIV^"AL"       Eaddr, "%al"
  DIV^"AX"       Eaddr, "%ax"
  DIV^"eAX"      Eaddr, "%eax"

  arithI^"b"     i8!,  Eaddr
  arithI^"w"     i16!, Eaddr
  arithI^"d"     i32!, Eaddr
  arithI^ov^"b"  i8!,  Eaddr
  MOV.Eb.Ib      i8!,  Eaddr
  MOV.Ev.Iv^ow   i16!, Eaddr
  MOV.Ev.Iv^od   i32!, Eaddr
```

<pentium-linux-syntax.spec>+= **(U->) [<-D->]**

```
assembly syntax
  arith^"rmb"    Eaddr, reg8
  arith^"rm"^ov  Eaddr, reg
  IMULrm^ov      Eaddr, reg
  MOV^"rmb"      Eaddr, reg
  MOV^"rm"^ov    Eaddr, reg
  MOVZX.Gv.Ew    Eaddr, r16
  MOVSX.Gv.Ew    Eaddr, r16
  MOVZX.Gv.Eb^ov Eaddr, r32
  MOVSX.Gv.Eb^ov Eaddr, r32
  BSF^ov  Eaddr, reg
  BSR^ov  Eaddr, reg
  LAR^ov  Eaddr, reg

  arith^"mrb"   reg8, Eaddr
  arith^"mr"^ov  reg, Eaddr
  MOV^"mr"^ov    reg, Eaddr
  MOV^"mrb"      reg, Eaddr
  TEST.Ev.Gv^ov  reg, Eaddr
  BT^ov          reg, Eaddr
  BTi^ov         i8!, Eaddr
  BTC^ov         reg, Eaddr
  BTCi^ov        i8!, Eaddr
  BTR^ov         reg, Eaddr
  BTRi^ov        i8!, Eaddr
  BTS^ov         reg, Eaddr
  BTSi^ov        i8!, Eaddr
  CMPXCHG.Eb.Gb  reg, Eaddr
  CMPXCHG.Ev.Gv^ov reg, Eaddr
```

<pentium-linux-syntax.spec>+= **(U->) [<-D->]**

```
patterns
  fstack    is FADD | FDIV | FDIVR | FMUL | FSUB | FSUBR
  fsti      is fstack | FCOMs
  stidx     is FFREE  | FUCOMs | FXCH
  Sstack    is P.STi.ST | .STi.St

assembly syntax
  fstack^Sstack  "%st", "%st"(idx)
  fstack^.ST.STi "%st"(idx), "%st"
  FCOMs^.ST.STi  "%st"(idx), "%st"
  FSTs.st^.STi   "%st"(idx)
  FLD.STi        "%st"(idx)
  stidx          "%st"(idx)

  FNSTSW.AX      "%ax"
  FSTSW.AX       "%ax"

  IDIV^"AX"     Eaddr, "%ax"
  IDIV^"eAX"    Eaddr, "%eax"

  IN.AL.Ib     i8!, "%al", i8!
  IN.eAX.Ib^ov i8!, "%eax", i8!
  IN.AL.DX     "%dx, %al"
  IN.eAX.DX^ov "%dx, %eax"

  IMUL.Iv^"d"    i32!,  Eaddr,  reg
  INT3           "$3"
  LEA^ov         Mem, reg

  MOVib          i8!, r8
  MOViw          i16!, r32
  MOVid          i32!, r32

  MOV.AL.Ob      offset, "%al"
  MOV.eAX.Ov^ov  offset, "%eax"
  MOV.Ob.AL      "%al", offset
  MOV.Ov.eAX^ov  "%eax", offset

  OUT.Ib.AL      "%al", i8!
  OUT.Ib.eAX^ov  "%eax", i8!
  OUT.DX.AL      "%al", "%dx"
  OUT.DX.eAX^ow  "%al", "%dx"
  OUT.DX.eAX^od  "%eax", "%dx"

patterns
  pES is POP.ES | PUSH.ES
  pSS is POP.SS | PUSH.SS
  pDS is POP.DS | PUSH.DS
  pFS is POP.FS | PUSH.FS
  pGS is POP.GS | PUSH.GS
assembly syntax
  pES "%ES"
  pSS "%SS"
  pDS "%DS"
  pFS "%FS"
  pGS "%GS"
  PUSH.CS "%CS"

  rot^B.Eb.1     "$1", Eaddr
  rot^B.Ev.1^ov  "$1", Eaddr

  rot^B.Eb.CL    "%cl", Eaddr
  rot^B.Ev.CL^ov "%cl", Eaddr

  rot^B.Eb.Ib    i8!, Eaddr
  rot^B.Ev.Ib^ov i8!, Eaddr
```

```
  shdIb^ov  count, reg, Eaddr
  shdCL^ov  "%cl", reg, Eaddr
```

<pentium-linux-syntax.spec>+= **(U->) [<-D->]**

```
  TEST.AL.Ib     i8, "%al"
  TEST.eAX.Iv^ow i16, "%ax"
  TEST.eAX.Iv^od i32, "%ax"
  TEST.Eb.Ib     i8,  Eaddr
  TEST.Ew.Iw     i16, Eaddr
  TEST.Ed.Id     i32, Eaddr
  TEST.Eb.Gb     reg, Eaddr
  TEST.Ev.Gv^ov  reg, Eaddr
  XADD.Eb.Gb     reg, Eaddr
  XADD.Ev.Gv^ov  reg, Eaddr
  XCHG.Eb.Gb     reg, Eaddr
  XCHG^"eAX"^ov  "%eax", r32
  XCHG.Ev.Gv^ov  reg, Eaddr
```

Effective addresses also use a different syntax under Gnu-Linux.

<pentium-linux-syntax.spec>+= **(U->) [<-D]**

```
assembly syntax
  Indir       (reg)
  Disp32      d(reg)
  Index       (base,index,ss)
  Index32     d(base,index,ss)
  ShortIndex  d(,index,ss)
```

Miscellaneous

The Intel 486 instruction set is a subset of the Pentium set. When generating instructions for that target, the pentium-only instructions are discarded.

<i486-linux.spec>=
<pentium32.spec>
discard *<names of pentium constructors>*
<pentium-linux.spec>

Names for generating assembly emitters are specified in several chunks.

<pentium-names.spec>=
<pentium-linux-names.spec>
<pentium-linux-default.spec>
<pentium-linux-syntax.spec>

<header.spec>= **(U-> U-> U-> U-> U-> U-> U->)**
<field specs>
patterns *<patterns for integer opcodes>*
<pattern specs for other patterns>
<prefix assignments>
<placeholders>
<constructors for displacements>
<constructors for effective addresses>

Check the arithmetic constructors. Substitute other chunks for this one to check each chunk.

<pentium-AF.spec>=
<header.spec>
<alphabetical constructors A-F>

<pentium-GK.spec>=
<header.spec>
<alphabetical constructors G-K>

<pentium-LQ.spec>=
<header.spec>
<alphabetical constructors L-Q>

<pentium-R.spec>=
<header.spec>
<alphabetical constructors R>

<pentium-SZ.spec>=
<header.spec>
<alphabetical constructors S-Z>

<pentium-arith.spec>=
<header.spec>
<arithmetic constructors>

<pentium-float.spec>=
<header.spec>
<floating-point constructors>

The Pentium checker checks all 32-bit instructions accepted by the Gnu-Linux assembler. We omit some variants of the `rot` instructions, because exhaustively checking each one seemed unnecessary.

<pentium-check.spec>=
<pentium32.spec>
<pentium-linux.spec>
```
discard
rot^B.Eb.1 rot^B.Ev.CL ROL^B.Eb.Ib RCL^B.Eb.Ib
SHLSAL^B.Eb.Ib SAR^B.Eb.Ib ROR^B.Ev.Ib^ov
RCR^B.Ev.Ib^ov SHR^B.Ev.Ib^ov
```

<pentium-checker.s>=
```
.align 16
```