New Jersey Machine-Code Toolkit Architecture Specifications

Toolkit version 0.5a

Norman Ramsey Mary Fernandez

February 20, 2001

Preface

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, debuggers, and more. The toolkit lets programmers encode and decode machine instructions symbolically. It transforms symbolic manipulations into bit manipulations, guided by a specification that defines mappings between symbolic and binary representations of instructions. This document presents our annotated specifications for the MIPS R3000, SPARC, Pentium, and Alpha instruction sets. Ramsey and Fernández (1995) present an overview of the toolkit; read it first. Ramsey and Fernández (1997) presents the toolkit's specification lanugage in depth, and the reference manual for the toolkit (Ramsey and Fernández 1994) gives a complete description.

This document does not just describe the specifications; it contains them. The `noweb` system (Ramsey 1994) for "literate programming" (Knuth 1992) makes it possible to write a single source containing both document and specifications. This source contains prose interleaved with definitions of named code "chunks." Code chunks contain specification code and references to other code chunks. The code chunks appear in an order suited to explanation, not necessarily in the order required by the toolkit language. Chunk names appear italicized within angle brackets, e.g., ⟨*pattern and constructor specifications* 18a⟩. The "18a" is the number of the chunk's first definition. Chunk definitions are numbered according to the pages on which they appear; When two or more chunks appear on the same page, a letter is appended to the page number, e.g. 18a and 18b.

We present the specifications mostly in order of increasing complexity: the MIPS, followed by the SPARC, the Alpha, and then the Pentium. The annotations in the MIPS specification describe basic use of the toolkit language. More sophisticated techniques appear in the SPARC, Alpha, and Pentium specifications. Annotations include descriptions of how each architecture's instruction set is factored in its specification and highlight useful specification techniques, e.g., those that protect against errors. References to the associated architecture manual also appear in the annotations. We encourage the reader to get the manuals so he can compare the information presented in the manuals to the equivalent information in the specifications.

Contents

1 MIPS instruction-set specification

This section is a complete specification of the MIPS instruction set. The MIPS is a good machine to start with because it has a small, regular instruction set. Its specification requires only the toolkit's basic features, and the regularity of the instruction set lends itself to factoring, making the specification small. As in the MIPS manual (Kane 1988), the specification is divided into two parts: one describing the instructions of the main CPU, the other describing those of the floating-point coprocessor.

Tokens and fields

The MIPS is a RISC machine; all instructions occupy exactly one 32-bit token. Although the instructions of the floating-point coprocessor are listed separately in the manual, they use some of the same fields as the instructions of the main CPU. This property makes it possible to define all the fields in a single token class. We use `noweb` to distribute the field specifications, putting most in the CPU section and the rest in the coprocessor section.

`fields` specifications give only the positions of fields. `fieldinfo` specifications give more information; for example, we use special names to refer to the values of fields that denote registers. `fieldinfo` specifications are never required.

The core MIPS specification describes the fields first, then patterns and constructors. We tell `noweb` to put it in a file called `mips-core.spec`.

4a ⟨*mips-core.spec* 4a⟩≡

```
fields of instruction (32) ⟨field specifications 4b⟩
⟨fieldinfo specifications 5b⟩
⟨pattern and constructor specifications 5c⟩
```

This code is written to file `mips-core.spec`.

1.1 The MIPS CPU

A field is specified by giving a range of bit positions. Some of the MIPS fields are listed on page A-3 of the MIPS architecture manual; others, like `breakcode`, apply to only a couple of instructions. The MIPS manual sometimes uses more than one name for the same field, for example, `offset` and `base` are used instead of `immed` and `rs` to describe load and store instructions. The numbers in the specification are the starting and ending bit positions, where 0 is the least and 31 the most significant bit.

4b ⟨*field specifications* 4b⟩≡ (4a) 12c▷

```
word 0:31 op 26:31 rs 21:25 rt 16:20 immed 0:15 offset 0:15 base 21:25
target 0:25 rd 11:15 shamt 6:10 funct 0:5 cond 16:20 breakcode 6:25
```

Special names are associated with the integer registers. These names can be used in constructor applications, as shown in the definitions of synthetic instructions in Section 1.1.5. These names can also be used to generate symbolic disassemblers, and "encoding" procedures that emit assembly language. On the MIPS, we use names of the form `r`*n*, except for the stack pointer (`sp`).

5a ⟨*properties of integer-register fields* 5a⟩≡ (5b)

```
names [ r0  r1  r2  r3  r4  r5  r6  r7  r8  r9  r10 r11 r12 r13 r14 r15
        r16 r17 r18 r19 r20 r21 r22 r23 r24 r25 r26 r27 r28 sp  r30 r31 ]
```

These names are just one of the properties that could be associated with the integer registers, which are `rs`, `rd`, `rt`, and `base`.

5b ⟨`fieldinfo` *specifications* 5b⟩≡ (4a) 12d▹

```
fieldinfo [ rs rt rd base ] is [ ⟨properties of integer-register fields 5a⟩ ]
```

1.1.1 Opcodes

An opcode is a pattern that constrains the values of one or more fields. Defining opcodes individually would be tedious, and the result would be hard to compare with the architecture manual, which uses opcode tables. The `patterns` declaration binds a table of names when a *generating expression* appears on the right-hand side, letting us write declarations that resemble the tables in the MIPS manual.

The numeric codes for the MIPS opcodes are described in three tables on page A-87 of the MIPS architecture manual. These tables are reproduced in Figure 1. Normal opcodes are six bits, and they appear in the `op` field of the instruction. This statement names patterns for each of the opcodes in the table at the top of Figure 1.

5c ⟨*pattern and constructor specifications* 5c⟩≡ (4a) 7a▹

```
patterns
 [ special bcond  j      jal    beq    bne    blez   bgtz
   addi    addiu  slti   sltiu  andi   ori    xori   lui
   cop0    cop1   cop2   cop3   _      _      _      _
   _       _      _      _      _      _      _      _
   lb      lh     lwl    lw     lbu    lhu    lwr    _
   sb      sh     swl    sw     _      _      swr    _
   lwc0    lwc1   lwc2   lwc3   _      _      _      _
   swc0    swc1   swc2   swc3   _      _      _      _ ]
 is op = {0 to 63}
```

This statement creates names for patterns constraining the `op` field. The expression "`op = {0 to 63}`" generates a list of 64 patterns, ranging from "`op = 0`" to "`op = 63`". Each pattern is associated with the corresponding name in the list to the left-hand side of `is`. For example, an instruction matches the pattern `beq` if its `op` field is 4. Patterns associated with the name "`_`" are discarded.

Opcode

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	†	†	†	†
3	†	†	†	†	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	†
5	SB	SH	SWL	SW	†	†	SWR	†
6	LWC0	LWC1	LWC2	LWC3	†	†	†	†
7	SWC0	SWC1	SWC2	SWC3	†	†	†	†

SPECIAL

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSCALL	BREAK	†	†
2	MFHI	MTHI	MFLO	MTLO	†	†	†	†
3	MULT	MULTU	DIV	DIVU	†	†	†	†
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	†	†	†	†
6	†	†	†	†	†	†	†	†
7	†	†	†	†	†	†	†	†

BCOND

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						
3								

† Operation codes marked with a dagger cause reserved instruction exceptions and are reserved for future versions of the architecture.

Figure 1: Opcode tables from MIPS architecture manual.

Two opcodes, `special` and `bcond`, are used for several instructions. These instructions are decoded by checking the bit-pattern in the `funct` and `cond` fields of the instructions, respectively. The following statement names patterns for each of the `special` opcodes in the table in the center of Figure 1:

7a ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁5c 7b▷

```
patterns
 [ sll     _       srl     sra     sllv    _       srlv    srav
   jr      jalr    _       _       syscall break   _       _
   mfhi    mthi    mflo    mtlo    _       _       _       _
   mult    multu   div     divu    _       _       _       _
   add     addu    sub     subu    and     or      xor     nor
   _       _       slt     sltu    _       _       _       _ ]
 is  special & funct = {0 to 47}
```

The `bcond` table at the bottom of Figure 1 is sparse. Instead of writing a table with many empty entries, we use another form of generating expression on the right-hand side; the list of integers in square brackets generates each integer in turn.

7b ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁7a 8▷

```
patterns
 [ bltz bgez bltzal bgezal ] is bcond & cond = [ 0 1 16 17 ]
```

These statements create new patterns by adding further constraints to the existing patterns `special` and `bcond`.

A slightly different way to look at this style of specification is that "`special`" and "`bcond`" are the names of tables, not opcodes. As we'll see, that view is a better one for the SPARC.

1.1.2 Grouping instructions

The rest of the patterns organize instructions into groups, much as is done in Chapter 3 of the MIPS manual. Most instructions are grouped by assembly-language syntax. Immediate-mode instructions are grouped into signed and unsigned variants (`immedS` and `immedU`), depending on whether the immediate operand is sign-extended. The toolkit uses disjunction to define patterns that match any of a group of related instructions.

8 ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁7b 9▷

```
patterns
  load    is lb | lbu | lh | lhu | lw | lwl | lwr | sb | sh | sw | swl | swr
  immedS  is addi | addiu | slti | sltiu
  immedU  is andi | ori | xori
  arith3  is add | addu | sub | subu | slt | sltu | and | or | xor | nor
  shift   is sll | srl | sra
  vshift  is sllv | srlv | srav
  arith2  is mult | multu | div | divu
  mfhilo  is mfhi | mflo
  mthilo  is mthi | mtlo
  jump    is j  | jal
  jumpr   is jr | jalr
  branch1 is blez | bgtz | bltz | bgez | bltzal | bgezal
  branch2 is beq | bne
  copls   is lwc0 | lwc1 | lwc2 | lwc3 | swc0 | swc1 | swc2 | swc3
```

1.1.3 Constructors for simple instructions

The patterns that group instructions can now be used to define constructors. A constructor specification contains an opcode, operands, possibly a set of equations, and an output pattern. The operands correspond to the procedure's arguments, and when the procedure is called, it emits the sequence of tokens defined by the output pattern. The toolkit generates an encoding procedure for each constructor. When the constructor name on the left-hand side is the name of a pattern, each disjunct of the pattern is used to generate a constructor. For example, the first declaration below generates a constructor for each disjunct bound to the pattern `load`, i.e., `lb`, `lbu`, `lh`, etc.

Most output patterns for the MIPS constructors are conjunctions of the opcode and all the operands. For example, the load instructions could be described by

`load rt, offset!(base) is load & rt & offset & base`.

Because this idiom is so common, not only on the MIPS but on other machines as well, the toolkit provides a special abbreviation for it: if the output pattern is omitted, the conjunction is inferred. We can use this trick to specify almost all of the integer instructions; only the one- and two-operand branches (`branch1` and `branch2`) and the jumps need to be treated specially. Given the groupings we defined above, we specify 56 instructions in just 15 lines:[1]

9 ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁8 10a▷

```
constructors
  load   rt, offset!(base)
  immedS rt, rs, offset!
  immedU rt, rs, offset
  lui    rt, offset
  arith3 rd, rs, rt
  shift  rd, rt, shamt
  vshift rd, rt, rs
  arith2 rs, rt
  mfhilo rd
  mthilo rs
  syscall
  break  breakcode
  copls  ft, offset!(base)
  jr     rs
  jalr   rd, rs
```

Operands with a trailing `!` (e.g., `offset!`) are treated as signed parameters in encoding procedures; the toolkit implicitly narrows their values to produce unsigned fields.

[1] This is the one place in specifications where newlines can be significant, preventing the opcode of a following pattern be taken as an operand of the current pattern. The reference manual explains the lexical details.

These constructor specifications include the assembly-language syntax associated with the instructions being specified. For example, most operands are separated by commas, but the syntax `offset(base)` is used to suggest the address computation in the load and store instructions. Using assembly syntax in the constructor specification makes the specification look more like the architecture manual. We use the assembly syntax to produce symbolic disassemblers and "encoding" procedures that emit assembly language.

1.1.4 Branches and relocatable addresses

Because it's a RISC machine, the MIPS can't use target addresses directly as field values in branch instructions; a 32-bit address wouldn't leave room for an opcode, or anything else. Instead, the branches are relative to the program counter; the field corresponding to the target address records the difference between that address and the PC. We use equations to specify the relationships between the operands and fields.

10a ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁9 11a▷

```
⟨placeholder definition 10b⟩
relocatable reloc
constructors
  branch1 rs, reloc
     { reloc = L + 4 * offset! } is branch1 & rs & offset; L: epsilon
  branch2 rs, rt, reloc
     { reloc = L + 4 * offset! } is branch2 & rs & rt & offset; L: epsilon
```

The branch equations capture the semantics of the MIPS branch instructions; the offset is sign-extended, shifted left 2 bits, and added to the address of the delay slot, i.e., `L`.

Constructor operands designated `relocatable` are *relocatable addresses*, like the branch targets named `reloc` above. Pattern labels, like `L` above, are also relocatable addresses. When a constructor that uses relocatable addresses is applied, it checks to see if those addresses are known (i.e., they have been assigned absolute adresses). If so, it treats them as ordinary integers and emits the instruction. Otherwise, it emits *placeholder* tokens and creates a relocation closure. The placeholders will be overwritten later, when the address becomes known and the application applies the closure.

We have to specify a placeholder pattern for tokens of each class. On the MIPS, we have only one class, so we need one placeholder. We've picked a `break` instruction with code 99, so that we'll get a trap if by some mischance the placeholder sneaks into a real program and we try to execute it.

10b ⟨*placeholder definition* 10b⟩≡ (10a)

```
placeholder for instruction is break(99)
```

The syntax "`break(99)`" isn't a special thing we use for placeholders; it's just the pattern we get by applying the `break` constructor to the value 99.

The PC-relative branches can't jump anywhere in the MIPS address space, only within 2^{16} words of the PC. The MIPS provides a `jump` instruction that lets a program jump to any word in the same quarter of the address space as the PC. It's best specified using a lot of bit operations; the toolkit provides a notation for taking *bit slices* of values. For example, `reloc@[28:31]` denotes the four most significant bits of the 32-bit, relocatable address `reloc`. The description of `j` on page A-31 in the MIPS manual corresponds to the equations below.

11a ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁10a 13b▷

```
constructors
  jump reloc { reloc@[28:31] = L@[28:31],
               reloc@[2:27]  = target,
               reloc@[0:1]   = 0 }
          is L: jump & target
```

The high bits of the target address must match the current address, the middle bits give the `target` field, and the low bits must be zero.

1.1.5 Synthetic instructions and conditional assembly

We have completed our description of the MIPS integer instructions, but MIPS applications often use "synthetic" instructions, i.e., instructions that are in the assembly language but not in the hardware instruction set. We specify constructors for synthetic instructions by defining them in terms of existing constructors. We put synthetic instructions in a separate specification, because not all applications need synthetics.

11b ⟨*mips-synth.spec* 11b⟩≡ 11c▷

```
constructors
  nop           is  sll (r0, r0, 0)
  mov rd, rs    is  addu(rd, rs, r0)
  b reloc       is  beq (r0, r0, reloc)
```

This code is written to file `mips-synth.spec`.

The earlier `fieldinfo` specification permits us to use names, not numbers, to refer to register values.

There is a large collection of synthetic branch instructions; they combine comparisons and branches. The result of comparisons is left in register 1, which MIPS conventions reserve for the assembler.

11c ⟨*mips-synth.spec* 11b⟩+≡ ◁11b 12a▷

```
constructors
  bge  rs, rt, reloc  is  slt (r1, rs, rt);  beq(r1, r0, reloc)
  bgeu rs, rt, reloc  is  sltu(r1, rs, rt);  beq(r1, r0, reloc)
  blt  rs, rt, reloc  is  slt (r1, rs, rt);  bne(r1, r0, reloc)
  bltu rs, rt, reloc  is  sltu(r1, rs, rt);  bne(r1, r0, reloc)
  bleu rs, rt, reloc  is  sltu(r1, rt, rs);  beq(r1, r0, reloc)
  ble  rs, rt, reloc  is  slt (r1, rt, rs);  beq(r1, r0, reloc)
  bgt  rs, rt, reloc  is  slt (r1, rt, rs);  bne(r1, r0, reloc)
  bgtu rs, rt, reloc  is  sltu(r1, rt, rs);  bne(r1, r0, reloc)
```

A full multiplication requires fetching the result from a special register.

12a ⟨*mips-synth.spec* 11b⟩+≡ ◁11c 12b▷
```
constructors
  mul rd, rs, rt  is  multu(rs, rt); mflo(rd)
```

Sometimes the best expansion for a synthetic instruction depends on the values of operands. We can choose one of several expansions by putting alternatives on the right-hand side of a constructor specification, each with its own set of equations. Each application of the constructor uses the first alternative for which the equations can be solved. For example, the `li` (load immeditate) synthetic instruction has three ways to load a signed value `imm` into register `rt`. When `imm` fits in 16 bits, use an immediate mode `addiu` instruction where the second operand is register 0, which is always 0. When the low-order 16 bits of `imm` are zero, use `lui` to assign the high-order bits to `rt`; `lui` automatically zero's the low-order bits. In the general case, cut `imm` into slices and use two instructions: `lui` to assign the high-order bits and `addiu` to add in the low-order bits.

12b ⟨*mips-synth.spec* 11b⟩+≡ ◁12a 15a▷
```
constructors
  li rt, imm
    when { imm@[16:31]! = imm@[15]! } is addiu(rt, r0, imm@[0:15]!)
    when { imm@[0:15] = 0 }           is lui  (rt, imm@[16:31])
    otherwise is lui(rt, imm@[16:31] + imm@[15]); addiu(rt, rt, imm@[0:15]!)
```

For the third branch, the toolkit generates an unnecessary test to see that `imm@[16:31] + imm@[15]` doesn't overflow 16 bits. The test isn't needed, because the only way it can happen is when everything in sight is ones, and in that case we take the first branch. I'd love to have a symbolic-algebra system smart enough to figure this out.

1.2 The floating-point coprocessor

Pages B-5 through B-7 of the MIPS manual introduce a few more field names for the convenience of specifying the floating-point instructions. The integer-register fields `rd`, `rs`, and `rt` could be re-used to refer to fields of floating-point instructions, but we introduce new fields `fd`, `fs`, and `ft` in order to have different names for the registers.

12c ⟨*field specifications* 4b⟩+≡ (4a) ◁4b 13c▷
```
ft 16:20 fs 11:15 fd 6:10 format 21:24 bit25 25:25
```

12d ⟨`fieldinfo` *specifications* 5b⟩+≡ (4a) ◁5b 13a▷
```
fieldinfo [ fs ft fd ] is [ ⟨properties of floating-point-register fields 12e⟩]
```

12e ⟨*properties of floating-point-register fields* 12e⟩≡ (12d)
```
names [ f0  f1  f2  f3  f4  f5  f6  f7  f8  f9  f10 f11 f12 f13 f14 f15
        f16 f17 f18 f19 f20 f21 f22 f23 f24 f25 f26 f27 f28 f29 f30 f31 ]
```

The `format` field has only three legitimate values: 0, 1, and 4. The `sparse` field attribute associates the names `s`, `d`, and `w` (signifying single, double, and word, respectively) with these values.

13a ⟨`fieldinfo` *specifications* 5b⟩+≡ (4a) ◁12d

```
fieldinfo format is [ sparse [ s = 0, d = 1, w = 4 ] ]
```

The instruction codes for Coprocessor 1 (floating point) are given on page B-28 of the MIPS manual. We use names like "`add.`" instead of "`add.fmt`" because they make it possible to form the full name of the instruction by concatenating the name of the opcode pattern and the name of the format, e.g., `add.s`, `add.d`, etc. They also distinguish the floating-point opcodes from analogous integer opcodes.

13b ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁11a 13d▷

```
patterns
 [ add.   sub.   mul.   div.   _      abs.   mov.   neg.
   _      _      _      _      _      _      _      _
   _      _      _      _      _      _      _      _
   _      _      _      _      _      _      _      _
   cvt.s  cvt.d  _      _      cvt.w  _      _      _
   _      _      _      _      _      _      _      _
   c.f    c.un   c.eq   c.ueq  c.olt  c.ult  c.ole  c.ule
   c.sf   c.ngle c.seq  c.ngl  c.lt   c.nge  c.le   c.ngt ]
 is cop1 & funct = {0 to 63} & bit25 = 1
```

Specifying the branch and move instructions is more complicated, because the relevant codes span several entries in the table, and the pattern language is designed to bind one pattern to one entry. We introduce two new fields to simplify the specification.

13c ⟨*field specifications* 4b⟩+≡ (4a) ◁12c

```
cop1code 22:25 copbcode 16:16
```

13d ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁13b 14a▷

```
patterns
  [ mfc1 cfc1 mtc1 ctc1 ] is cop1 & cop1code = {0 to 3} & funct = 0
  bc1x is cop1 & (cop1code = 4 | cop1code = 6)
  bc1f is bc1x & copbcode = 0
  bc1t is bc1x & copbcode = 1
```

The pattern `bc1x` is under-constrained. The architecture manual indicates that the `cop1code` field for the `bc1f` and `bc1t` instructions can have the value 4 or 6. When generating encoding code for the `bc1f` and `bc1t` constructors, the toolkit warns that their output patterns are under-constrained and chooses the `cop1code = 4`, as described in the first disjunct for `bc1x`. Toolkit-generated decoding code recognizes either variant.

Like the integer opcodes, the floating-point opcodes are grouped by assembly-language syntax.

14a ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁13d 14b▷

```
patterns
  arith3. is add. | div. | mul. | sub.
  arith2. is abs. | mov. | neg.
  movec1  is mfc1 | mtc1 | cfc1 | ctc1
  c.cond  is c.f  | c.un   | c.eq  | c.ueq | c.olt | c.ult | c.ole | c.ule |
             c.sf | c.ngle | c.seq | c.ngl | c.lt  | c.nge | c.le  | c.ngt
  lsc1    is lwc1 | swc1
  convert is cvt.s | cvt.d | cvt.w
```

The following constructor specifications introduce the compound-opcode concatenation operator (`^`). Each floating-point opcode has three variants, according to whether the value of the `format` field is `s`, `d`, or `w`. We form the constructors by concatenating opcode names with `format`. One constructor is created for each member of the cross product of the patterns named, (e.g., `add.s`, `add.d`, `add.w`, `div.s`, `div.d`, etc). Constructor names can also include string literals delimited by double quotes, e.g. `convert^"."^format`. As with the integer instructions, most output patterns are implicit.

The floating-point arithmetic, conditional, and type-conversion instructions all require floating-point operands that are even-odd register pairs. The equations for `arith3.^format`, etc., below specify that the register operands are all even values. These equations have been commented out because we believe people won't want to pay the overheads of checking when generating encoding procedures, but we want them when we validate the spec, so I have arranged for the command `sed 's/# {/{/'` to uncomment them.

14b ⟨*pattern and constructor specifications* 5c⟩+≡ (4a) ◁14a

```
constructors
  arith3.^format fd, fs, ft # { fd = 2 * _, fs = 2 * _, ft = 2 * _ }
  arith2.^format fd, fs     # { fd = 2 * _, fs = 2 * _ }
  movec1         rt, rd
  c.cond^"."^format  fs, ft # { fs = 2 * _, ft = 2 * _ }
  convert^"."^format fd, fs # { fd = 2 * _, fs = 2 * _ }
  "bc1f" reloc { reloc = L + 4 * offset! } is bc1f & offset; L: epsilon
  "bc1t" reloc { reloc = L + 4 * offset! } is bc1t & offset; L: epsilon
```

We have to quote `bc1f` and `bc1t` instead of using them directly as opcodes, because these patterns have multiple disjuncts. The constructor-specification process "explodes" opcodes, so it would pick just one disjunct, and we would be unable to recognize the other when decoding an application of the constructor.

There are several synthetic instructions that operate on pairs of floating point registers. `l.d` and `s.d` load and store double words, respectively. `mtc.d` and `mfc.d` move pairs of words in registers to and from the floating point co-processor.

15a ⟨*mips-synth.spec* 11b⟩+≡ ◁12b

```
constructors
  l.d ft,offset!(base) # { ft = 2 * _ }
      is  lwc1(ft, offset!, base); lwc1(ft+1, offset!+4, base)
  s.d ft,offset!(base) # { ft = 2 * _ }
      is  swc1(ft, offset!, base); swc1(ft+1, offset!+4, base)
  mtc1.d rt, fs  # { rt = 2 * _, fs = 2 * _ }
      is  mtc1(rt, fs); mtc1(rt+1, fs+1)
  mfc1.d rt, fs  # { rt = 2 * _, fs = 2 * _ }
      is  mfc1(rt, fs); mfc1(rt+1, fs+1)
```

Because `offset` is a field, it is unsigned, and so we must use `offset!` to get the signed offset that `lwc1` and similar constructors expect.

1.3 Application-specific specifications for the `mld` linker

An application writer has the option of extending a machine's specification to include application-specific information. These extensions can simplify implementation of the application or can be used by the toolkit to generate more efficient encoding code. `mld` is a retargetable, optimizing linker that generates code for the MIPS or the SPARC. `mld` uses the toolkit to emit executable binary directly, instead of going through the assembler. This means it needs substitutes for the synthetic `div`, `divu`, `rem`, and `remu` instructions provided by the MIPS assembler.

The MIPS hardware does no checking for divide by zero or divisions that overflow; these checks have to be done in software. A `break 7` instruction is used to indicate one of these errors; so `mld` has to generate code that does these tests. The first test just checks to see if the divisor is zero, branching around the break instruction if it isn't. If, for some reason, we can tell at compile time that the divisor is zero, we just emit a break, because that's what the MIPS assembler does.

15b ⟨*mld-mips.spec* 15b⟩≡ 16a▷

```
constructors
  break7ifzero rt
    when { rt = 0 } is break(7)
    otherwise       is bne(rt, r0, L); nop(); break(7); L: epsilon
```

This code is written to file `mld-mips.spec`.

The other test, for overflow, is more complicated. On a two's-complement machine, integer division can overflow if it is signed, if the divisor is −1, and if the dividend is the most negative integer—there aren't enough bits to represent the result.

16a ⟨*mld-mips.spec* 15b⟩+≡ ◁15b 16b▷

```
constructors
  break7ifoverflow rs, rt is
        addiu(r1, r0, -1);
        bne(rt, r1, L);
        lui(r1, 0x8000);
        bne(rs, r1, L);
        nop();
        break(7);
    L : epsilon
```

Given these two tests, we can define safe versions of signed and unsigned division and remainder.

16b ⟨*mld-mips.spec* 15b⟩+≡ ◁16a 17a▷

```
constructors
  tested_divu rd, rs, rt
    is divu(rs, rt); nop(); break7ifzero(rt); mflo(rd); nop()
  tested_div rd, rs, rt
    is div(rs, rt);  nop(); break7ifzero(rt); break7ifoverflow(rs, rt); mflo(rd); nop()
  tested_remu rd, rs, rt
    is divu(rs, rt); nop(); break7ifzero(rt); mfhi(rd); nop()
  tested_rem rd, rs, rt
    is div(rs, rt);  nop(); break7ifzero(rt); break7ifoverflow(rs, rt); mfhi(rd); nop()
```

Applications can also play games to trade safety for efficiency. Toolkit-generated encoding procedures prevent the application from putting a value in a field that is too small for it, so for example they won't let an application put 255 in a 5-bit field. If an application writer wants to promise that the values of a field will always fit, it can call the field "unchecked," and the toolkit will omit the check, but it will still mask out the high bits. If the application can guarantee the high bits are zero, it can call a field "guaranteed," and the toolkit won't even do the masking. Many applications, including `mld`, include register allocators that always assign proper values to the fields denoting registers:

16c ⟨*mips-regs.spec* 16c⟩≡

```
fieldinfo [ rs rt rd base fs ft fd ] is [ guaranteed ]
```

This code is written to file `mips-regs.spec`.

`mld` emits relocatable addresses into the data segments. There is an argument for putting an address-emitting procedure into the toolkit library, but then that procedure would have to deal with all the details of generating a closure and so on when the value of the address wasn't yet known. Since the toolkit's generator already takes care of those details, it is much easier simply to specify a constructor that emits an address. We create a new token so we can distinguish a 32-bit data token from a 32-bit instruction token, and we choose a placeholder that is known to be an illegal address.

17a ⟨*mld-mips.spec* 15b⟩+≡ ◁16b

```
fields of addrtoken (32) addr32 0:31
placeholder for addrtoken is addr32 = 7
constructors
  emit_raddr reloc is addr32 = reloc
```

1.4 Specifications to support validation

We have validated this specification against the DEC-Ultrix MIPS assembler. To do that, we had to alter a little bit of syntax, and we had to discard some constructors that this assembler didn't know about.

17b ⟨*mips-names.spec* 17b⟩≡

```
assembly opcode
  jr                 is j
  jalr               is jal
  mov                is move
  {sll,srl,sra}v     is $1
  tested_{*}         is $1
assembly operand
  [ rs rt rd base ]  is "$%d"
  [ fs ft fd ]       is "$%s"
```

This code is written to file `mips-names.spec`.

The machine instructions not recognized by the MIPS assembler are discarded.

17c ⟨*mips-check.spec* 17c⟩≡

```
discard
  break lwc0 swc0 arith3.^"w" arith2.^"w" c.cond^"."^"w" cvt.s.s
  cvt.w.w cvt.d.d swc2 lwc2 swc3 lwc3
```

This code is written to file `mips-check.spec`.

On the MIPS, the checker's assembly output must include the `noreorder` directive.

17d ⟨*mips-checker.s* 17d⟩≡

```
.set noreorder
```

This code is written to file `mips-checker.s`.

2 SPARC instruction-set specification

This section specifies the SPARC instruction set. It uses two specification techniques not needed in the simpler MIPS specification. Operands of some SPARC instructions contain more than one field; we use *typed constructors* to specify them. We also use the extended pattern declaration

p `is any of [` $p_0\ p_1 \ldots p_n$ `], which is` *generating expression*,

which not only defines patterns p_0 through p_n, but also defines p to be their disjunction.

We can factor the SPARC instruction in much the same way as we did the MIPS; instructions with identical assembly syntax are grouped using disjunction and defined in one constructor declaration.

The SPARC assembly language contains overloaded instructions, the encodings of which depend on the number and types of the operands. Overloading is easy to handle in a textual assembler because a parser can use information about operands to resolve overloaded names. The toolkit, however, generates encoding procedures written in C, which does not permit users to overload procedures. For that reason, the toolkit does not permit overloading of constructor names; the specification writer must choose a distinct name for each variant of an overloaded instruction. It's also possible to use constructor types to play games with overloaded instructions; we show an example in Section 2.11.

2.1 Fields

The SPARC is a RISC architecture and thus uses a single token class for all of its fields.

18a ⟨*sparc-core.spec* 18a⟩≡

```
fields of instruction (32) ⟨field specifications 18b⟩
⟨fieldinfo specifications 27e⟩
⟨pattern and constructor specifications 19a⟩
```

This code is written to file `sparc-core.spec`.

Information about instruction formats and fields is taken from Chapter 5 of the SPARC architecture manual.

18b ⟨*field specifications* 18b⟩≡ (18a) 26b▷

```
inst 0:31 op 30:31 disp30 0:29 rd 25:29 op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 op3 19:24 rs1 14:18 i 13:13 asi 5:12 rs2 0:4 simm13 0:12 opf 5:13
```

2.2 Opcodes

The following opcode tables are derived from the tables in Appendix F of the SPARC manual (SPARC 1992).

Where an entry in a table refers to another table, we define a pattern with the name of that table (e.g., `TABLE_F2`). That pattern is not useful by itself, but is used to define more opcodes in a pattern-binding statement that resembles the eponymous table. These strange "opcodes" actually play the same role as the "`special`" and "`bcond`" opcodes on the MIPS architecture, but the SPARC people have a different way of naming such things.

The SPARC opcode tables are organized hierarchically; the first table in Appendix F is at the top of the hierarchy, and it has four entries corresponding to the four possible values of the (two-bit) `op` field. Only one of these entries (`CALL`) is an opcode; the others refer the reader to subsequent tables.

19a ⟨*pattern and constructor specifications* 19a⟩≡ (18a) 19b ▹

```
patterns
 [ TABLE_F2 CALL TABLE_F3 TABLE_F4 ] is op  = {0 to 3}
```

Table F-2 is short, but it presents an oddity: an opcode with two names. `SETHI` means `NOP` when `rd` and `imm22` are zero. On the MIPS, no-ops were treated as synthetic instructions, but here we define `NOP` as a separate opcode, reflecting the presentation in the manual.

19b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◃ 19a 20a ▹

```
patterns
 [ UNIMP Bicc SETHI FBfcc CBccc ] is TABLE_F2 & op2 = [0 2 4 6 7]
 NOP                              is SETHI & rd = 0 & imm22 = 0
```

Table F-3 includes most of the arithmetic and logical opcodes. It must be specified slightly differently from the MIPS tables because the values of the `op3` field are "counted down" instead of "counted across." That is, the codes corresponding to `op = 0, 1, 2, ...` are `ADD, AND, OR, ...`, not `ADD, ADDcc, TADDcc, ....` To get the proper numbering, we use the generating expression `{0 to 63 columns 4}`, which generates the integers from 0 to 63 in the order $(0, 16, 32, 48, 1, \ldots, 63)$.

20a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁19b 20b▷

```
patterns
 [ ADD  ADDcc  TADDcc   WRxxx
   AND  ANDcc  TSUBcc   WRPSR
   OR   ORcc   TADDccTV WRWIM
   XOR  XORcc  TSUBccTV WRTBR
   SUB  SUBcc  MULScc   FPop1
   ANDN ANDNcc SLL      FPop2
   ORN  ORNcc  SRL      CPop1
   XNOR XNORcc SRA      CPop2
   ADDX ADDXcc RDxxx    JMPL
   _    _      RDPSR    RETT
   UMUL UMULcc RDWIM    Ticc
   SMUL SMULcc RDTBR    FLUSH
   SUBX SUBXcc _        SAVE
   _    _      _        RESTORE
   UDIV UDIVcc _        _
   SDIV SDIVcc _        _       ] is TABLE_F3 & op3 = {0 to 63 columns 4}
```

We have used `WRxxx` and `RDxxx` to stand for the groups of opcodes that appear in the corresponding positions in Table F-3. These opcodes define variants of the `wr` and `rd` instructions, which are overloaded. The overloading is resolved by looking at the values of operands, as shown by the footnotes to Table F-3. It probably would have been simpler to specify these purely as synthetic instructions, but we've chosen to be slaves to the SPARC manual.

20b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁20a 21▷

```
patterns
  WRASR         is WRxxx & rd != 0   # should be rdi != 0
  WRY           is WRxxx & rd = 0
  RDASR         is RDxxx & rs1 != 0  # should be rs1i != 0
  RDY           is RDxxx & rs1 = 0
  STBAR         is RDxxx & rs1 = 15 & rd = 0
```

These patterns show a use of the inequality operator to specify constraints on a field value. The inequality constraint is a crock that just happens to work in these examples. The toolkit can't represent an inequality constraint directly, so `f != n` is syntactic sugar for:

$$\mathtt{0} \leq f < n \mid \mathtt{n} < f \leq \mathtt{f}_{max},$$

where $\mathtt{f}_{max}$ is the largest value the field can hold. Since a disjunction almost never makes sense as an opcode, this is asking for trouble. Luckily, in these cases `n` is either 0 or $\mathtt{f}_{max}$, so at most one of the disjuncts ever matches anything, and what is left is equivalent to a real inequality constraint. We hope one day to remove this ugly wart from the toolkit.

Table F-4 includes the load and store opcodes.

21 ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁20b 22▷

```
patterns
 [ LD     LDA     LDF   LDC
   LDUB   LDUBA   LDFSR LDCSR
   LDUH   LDUHA   _ _
   LDD    LDDA    LDDF  LDDC
   ST     STA     STF   STC
   STB    STBA    STFSR STCSR
   STH    STHA    STDFQ STDCQ
   STD    STDA    STDF  STDC
   _      _       _     _
   LDSB   LDSBA   _     _
   LDSH   LDSHA   _     _
   _      _       _     _
   _      _       _     _
   LDSTUB LDSTUBA _     _
   _      _       _     _
   SWAP.  SWAPA   _     _  ]  is TABLE_F4 & op3 = {0 to 63 columns 4}
```

Table F-5 includes the floating-point arithmetic and conversion opcodes. Following Table F-5, we list all the opcodes with their corresponding names, instead of using a complete table. We've also chosen to divide the opcodes into two groups: two-operand and three-operand instructions.

We further divide the opcodes based on their operand types. This division makes it easy to use the names `float2`, `float3s`, etc. to refer to the groups, without having to re-enumerate the members of each group; these names are used later in the specification to specify complete instructions.

22 ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁21 23a▷

```
patterns
  float2 is any of [ FMOVs FNEGs FABSs FSQRTs FSQRTd FSQRTq
                     FiTOs FdTOs FqTOs FiTOd  FsTOd  FqTOd
                     FiTOq FsTOq FdTOq FsTOi  FdTOi  FqTOi ],
  which is FPop1 & opf =
                   [ 0x1   0x5   0x9   0x29   0x2a   0x2b
                     0xc4  0xc6  0xc7  0xc8   0xc9   0xcb
                     0xcc  0xcd  0xce  0xd1   0xd2   0xd3 ]
  float2s is FMOVs | FNEGs | FABSs | FSQRTs
  FTOs    is FiTOs | FsTOi
  FTOd    is FiTOd | FsTOd
  FTOq    is FiTOq | FsTOq
  FdTO    is FdTOi | FdTOs
  FqTO    is FqTOs | FqTOi

  float3 is any of [ FADDs FADDd FADDq FSUBs FSUBd FSUBq  FMULs
                     FMULd FMULq FDIVs FDIVd FDIVq FsMULd FdMULq ],
    which is FPop1 & opf =
                   [ 0x41  0x42  0x43  0x45  0x46  0x47   0x49
                     0x4a  0x4b  0x4d  0x4e  0x4f  0x69   0x6e ]
  float3s is  FADDs | FSUBs | FMULs | FDIVs
  float3d is  FADDd | FSUBd | FMULd | FDIVd
  float3q is  FADDq | FSUBq | FMULq | FDIVq
```

This syntactic shorthand, which defines patterns and disjoins them at one blow, reduces the possibility of errors due to miscopying. We didn't need this idiom to describe the MIPS because it uses just a few tables, and the grouping of MIPS instructions doesn't correspond to the organization of the tables. We'll see it again, however, in specifying the rest of the SPARC, and also on the Pentium and the Alpha.

Table F-6 includes the floating-point comparison opcodes. Again, we write down the codes rather than use a table.

23a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁22 23b▷

```
patterns
  fcompares is any of      [ FCMPs FCMPEs ],
    which is FPop2 & opf = [ 0x51  0x55 ]
  fcompared is any of      [ FCMPd FCMPEd ],
    which is FPop2 & opf = [ 0x52  0x56 ]
  fcompareq is any of      [ FCMPq FCMPEq ],
    which is FPop2 & opf = [ 0x53  0x57 ]
```

Table F-7 includes the branch and trap opcodes. It is strange for a different reason; it is actually four tables in one. The rows of each table vary with the values of the `cond` field, but the columns are described by four different patterns. We can't create such a single table with the toolkit, nor would we want to—we just split it into the four underlying tables, which correspond to three kinds of conditional branches and one of conditional traps.

23b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁23a 24b▷

```
patterns
  ibranch is any of [ BN BE  BLE BL  BLEU BCS BNEG BVS
                      BA BNE BG  BGE BGU  BCC BPOS BVC ],
    which is Bicc & cond = {0 to 15}

  fbranch is any of [ FBN FBNE FBLG FBUL FBL   FBUG FBG   FBU
                      FBA FBE  FBUE FBGE FBUGE FBLE FBULE FBO ],
    which is FBfcc & cond = {0 to 15}

  cbranch is any of [ CBN CB123 CB12 CB13 CB1   CB23 CB2   CB3
                      CBA CB0   CB03 CB02 CB023 CB01 CB013 CB012 ],
    which is CBccc & cond = {0 to 15}

  trap is any of    [ TN TE  TLE TL  TLEU TCS TNEG TVS
                      TA TNE TG  TGE TGU  TCC TPOS TVC ],
    which is Ticc & cond = {0 to 15}

  branch is ibranch | fbranch | cbranch
```

We group the three kinds of branches into the pattern `branch`.

People can never agree on the names of conditional branch instructions. We include a few of the most common synonyms, which are used in the definition of synthetic instructions.

24a ⟨*sparc-synth.spec* 24a⟩≡ 32▷

```
patterns
  B    is BA
  BGEU is BCC
  BLU  is BCS
  BNZ  is BNE
  branch.synonyms is B | BGEU | BLU | BNZ
```

This code is written to file `sparc-synth.spec`.

2.3 Structured operands and typed constructors

Unlike the MIPS, the SPARC has instructions whose operands are not simple integers or fields. For example, the integer-arithmetic instructions take an operand that may be a register or an immediate value, and the load and store instructions take an operand that computes an address. The formats for these operands appear on page 84 in Appendix A of the SPARC manual.

We specify such operands by creating a constructor type for them, giving a constructor for each format. We use the "operand syntax" name in the SPARC manual as the name of the type; for example, the constructors for a "register or immediate" operand are:

24b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁23b 25a▷

```
constructors
  imode simm13! : reg_or_imm  is  i = 1 & simm13
  rmode rs2     : reg_or_imm  is  i = 0 & rs2
```

The type `reg_or_imm` is *defined* by these declarations. Used elsewhere, an operand of type `reg_or_imm` always denotes a pattern created by the application of one of these two constructors. Unlike the encoding procedures for ordinary untyped constructors, the encoding procedures generated for these typed constructors have no side effects; they simply return values that can be passed to other constructors requiring an operand of type `reg_or_imm`. Such constructors, notably the arithmetic constructors, appear below.

Specifying addresses is a bit problematic because it's not clear what convention to follow. The underlying general mechanism is that a register is added to a value of type `reg_or_imm`, but there are many useful abbreviations:[2]

25a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁24b 25b▷

```
constructors
  generalA  rs1 + reg_or_imm : address  is  rs1 & reg_or_imm
  dispA     rs1 + simm13!    : address  is  generalA(rs1, imode(simm13!))
  absoluteA simm13!          : address  is  generalA(0,   imode(simm13!))
  indexA    rs1 + rs2        : address  is  generalA(rs1, rmode(rs2))
  indirectA rs1              : address  is  generalA(rs1, rmode(0))
```

Unfortunately We can't call the type `address` because `address` is reserved for the toolkit to describe the treatment of addresses in decoding specifications. Another ugly wart.

2.4 Load and store, read and write

We group the load and store opcodes by their assembly syntax. These groupings reflect the information provided on pages 90 and 92 for the load instructions and pages 95 and 97 for the stores. Each group uses a different set of names for their register operands. Those names are defined below.

25b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁25a 26a▷

```
patterns
  loadg  is LDSB  | LDSH  | LDUB  | LDUH  | LD  | LDSTUB  | SWAP.
  loada  is LDSBA | LDSHA | LDUBA | LDUHA | LDA | LDSTUBA | SWAPA
  storeg is STB   | STH   | ST
  storea is STBA  | STHA  | STA
```

[2]As far as `lcc` goes, we could revert to the original mechanism

The constructors for the load and store instructions illustrate the use of the typed constructor `address` as an operand. Addresses are bracketed, as in SPARC assembly language. Because the constructors' output patterns are the conjunctions of the opcodes and operands, they can be omitted.

Loads and stores of double words require even-odd register pairs for the destination and source registers; their equations specify that constraint. For performance reasons, these equations don't appear in an ordinary spec, but we want them when we validate the spec, so I have arranged for the command `sed 's/# {/{/'` to uncomment them

26a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁25b 27a▷

```
constructors
  loadg  [address], rd
  LDD    [address], rd    # { rd = 2 * _ }
  LDF    [address], fd
  LDDF   [address], fd    # { fd = 2 * _ }
  LDC    [address], cd
  LDDC   [address], cd    # { cd = 2 * _ }

  storeg rd, [address]
  STD    rd, [address]    # { rd = 2 * _ }
  STF    fd, [address]
  STDF   fd, [address]    # { fd = 2 * _ }
  STC    cd, [address]
  STDC   cd, [address]    # { cd = 2 * _ }
```

We use the synonyms `fd` and `cd` for the destination register `rd` when the affected register is a floating-point or coprocessor register; same trick applies to `fs1` and `fs2`. This trick enables us to use different names for different registers, which is helpful in generating a disassembler from these specifications.

26b ⟨*field specifications* 18b⟩+≡ (18a) ◁18b 28d▷

```
fd 25:29 cd 25:29 fs1 14:18 fs2 0:4
```

The `loada` and `storea` only accept address operands constructed from registers. As in the SPARC manual, we introduce the constructor type `regaddr` to describe this operand.[3]

27a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁26a 28a▷

```
constructors
  indexR    rs1 + rs2      : regaddr  is  i = 0 & rs1 & rs2
  indirectR rs1            : regaddr  is  i = 0 & rs2 = 0 & rs1

  loada  [regaddr]asi, rd
  LDDA   [regaddr]asi, rd # { rd = 2 * _ }
  storea rd, [regaddr]asi
  STDA   rd, [regaddr]asi # { rd = 2 * _ }
```

Here are the names associated with the different registers. They have to be quoted because the toolkit doesn't think the `%` character can be part of an identifier.

27b ⟨*properties of integer-register fields* 27b⟩≡ (27e)

```
names [ "%g0"  "%g1"  "%g2"  "%g3"  "%g4"  "%g5"  "%g6"  "%g7"
        "%o0"  "%o1"  "%o2"  "%o3"  "%o4"  "%o5"  "%sp"  "%o7"
        "%l0"  "%l1"  "%l2"  "%l3"  "%l4"  "%l5"  "%l6"  "%l7"
        "%i0"  "%i1"  "%i2"  "%i3"  "%i4"  "%i5"  "%fp"  "%i7" ]
```

27c ⟨*properties of floating-point registers* 27c⟩≡ (27e)

```
names [ "%f0"  "%f1"  "%f2"  "%f3"  "%f4"  "%f5"  "%f6"  "%f7"
        "%f8"  "%f9"  "%f10" "%f11" "%f12" "%f13" "%f14" "%f15"
        "%f16" "%f17" "%f18" "%f19" "%f20" "%f21" "%f22" "%f23"
        "%f24" "%f25" "%f26" "%f27" "%f28" "%f29" "%f30" "%f31" ]
```

27d ⟨*properties of coprocessor destination register* 27d⟩≡ (27e)

```
names [ "%c0"  "%c1"  "%c2"  "%c3"  "%c4"  "%c5"  "%c6"  "%c7"
        "%c8"  "%c9"  "%c10" "%c11" "%c12" "%c13" "%c14" "%c15"
        "%c16" "%c17" "%c18" "%c19" "%c20" "%c21" "%c22" "%c23"
        "%c24" "%c25" "%c26" "%c27" "%c28" "%c29" "%c30" "%c31" ]
```

27e ⟨`fieldinfo` *specifications* 27e⟩≡ (18a) 29c▷

```
fieldinfo
[ rd rs1 rs2 ] is [ ⟨properties of integer-register fields 27b⟩ ]
[ fd fs1 fs2 ] is [ ⟨properties of floating-point registers 27c⟩ ]
            cd is [ ⟨properties of coprocessor destination register 27d⟩ ]
```

[3] In fact, `address` is a subtype of `regaddr`. The constructors `indexA` and `indirectA` are equivalent to `indexR` and `indirectR`. We could specify these addresses by adding a constructor to create an `address` from a `regaddr`, but that pollutes the name space. It would be better to have subtypes.

The SPARC also has several specialized load and store instructions. The registers loaded and stored are implicit in the opcodes, but we put them into the specifications as "assembly-language syntax." This technique helps us generate a disassembler, but more importantly, it makes the specification easier to read and understand.

28a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁27a 28b▷

```
constructors
  LDFSR  [address], "%fsr"
  LDCSR  [address], "%csr"
  STFSR  "%fsr", [address]
  STCSR  "%csr", [address]
  STDFQ  "%fq",  [address]
  STDCQ  "%cq",  [address]
```

A similar group of instructions modify a different set of registers. Perhaps because they move information between special-purpose registers and general-purpose registers, instead of special-purpose registers and memory, they are called "read" and "write" instructions instead of "load" and "store" instructions.

28b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁28a 28c▷

```
constructors
  RDY    "%y",   rd
  RDPSR  "%psr", rd
  RDWIM  "%wim", rd
  RDTBR  "%tbr", rd
  WRY    rs1, reg_or_imm, "%y"
  WRPSR  rs1, reg_or_imm, "%psr"
  WRWIM  rs1, reg_or_imm, "%wim"
  WRTBR  rs1, reg_or_imm, "%tbr"
```

Finally, the instructions that read and write the ancillary state register (`asr`) have their own special syntax.

28c ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁28b 29a▷

```
constructors
  RDASR   "%asr"rs1i, rd
  WRASR   rs1, reg_or_imm, "%asr"rdi
  STBAR
```

The registers `rs1i` and `rdi` are versions of `rs1` and `rd` that print as integers.

28d ⟨*field specifications* 18b⟩+≡ (18a) ◁26b

```
rs1i 14:18 rdi 25:29
```

2.5 Shift, logic, and arithmetic

The logical, shift, and arithmetic instructions share identical assembly language syntax, so we can get away with only a single constructor declaration. Including the shift instructions with the logical and arithmetic instructions is a cheat, but it's a cheat that works because the immediate field, if used, must be zero in the high-order 8 bits.

29a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁28c 29b▷

```
patterns
  logical is AND | ANDcc | ANDN | ANDNcc | OR | ORcc | ORN | ORNcc |
             XOR | XORcc | XNOR | XNORcc
  shift   is SLL | SRL   | SRA
  arith   is ADD | ADDcc | ADDX | ADDXcc | TADDcc | TADDccTV |
             SUB | SUBcc | SUBX | SUBXcc | TSUBcc | TSUBccTV |
             MULScc | UMUL | SMUL | UMULcc | SMULcc |
             UDIV | SDIV | UDIVcc | SDIVcc |
             SAVE | RESTORE
  alu     is logical | shift | arith

constructors
  alu rs1, reg_or_imm, rd
```

2.6 Branches and call

As on the MIPS, a placeholder pattern must be specified before the declaration of any constructor that refers to relocatable addresses; such a constructor may emit a placeholder in lieu of the relocated instruction. We choose the `UNIMP` instruction because it causes an illegal-instruction trap if executed.

29b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁29a 29d▷

```
placeholder for instruction is UNIMP & imm22 = 0xbad
```

Branch instructions come in two variants depending on the setting of the annul (`a`) bit. In the assembly language, the `a` bit is notated with the suffix `,a` when set, and with no suffix when clear. We can work this notation into the toolkit specification by making these suffixes the names of the values of the `a` bit:

29c ⟨`fieldinfo` *specifications* 27e⟩+≡ (18a) ◁27e

```
fieldinfo a is [ names [ "" ",a" ] ]
```

We then use `a` as a suffix to the branch instruction:

29d ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁29b 30a▷

```
relocatable target raddr
constructors
  branch^a target  { target = L + 4 * disp22! } is L: branch & a & disp22
```

This declaration creates 96 constructors, two for each disjunct of the `branch` pattern.

As on the MIPS, all of the branch instructions take a relocatable address and emit a PC-relative displacement in the output pattern. To relate them, we have simply used the equation that appears on page 120 of the SPARC manual.

The `call` instruction is like the branches, except it uses a 30-bit displacement, and there is no annul bit.

30a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁29d 30b▷

```
constructors
  call  target   { target = L + 4 * disp30! } is L: CALL & disp30
```

We write the constructor name with two trailing underscores because we play overloading games below.

2.7 Floating point

Floating-point arithmetic instructions include two-operand and three-operand variants. Their output patterns are the implicit conjunctions of their opcodes and operands so the output pattern is omitted. Equations enforce that the register specified for a double-word operand is even-numbered (i.e., specifies the first member of an even-odd pair) and that the register for a quad-word operand is a multiple of four (i.e., specifies the first in a four-register set). As usual, those equations are commented out for speed.

30b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁30a 31a▷

```
constructors
  float2s fs2, fd
  FSQRTd  fs2, fd # { fs2 = 2 * _, fd = 2 * _ }
  FSQRTq  fs2, fd # { fs2 = 4 * _, fd = 4 * _ }

  FTOs fs2, fd
  FTOd fs2, fd  # { fd = 2 * _ }
  FTOq fs2, fd  # { fd = 4 * _ }
  FdTO fs2, fd  # { fs2 = 2 * _ }
  FqTO fs2, fd  # { fs2 = 4 * _ }
  FqTOd fs2, fd  # { fs2 = 4 * _, fd = 2 * _ }
  FdTOq fs2, fd  # { fs2 = 2 * _, fd = 4 * _ }

  float3s  fs1, fs2, fd
  float3d  fs1, fs2, fd # { fs1 = 2 * _, fs2 = 2 * _, fd = 2 * _ }
  float3q  fs1, fs2, fd # { fs1 = 4 * _, fs2 = 4 * _, fd = 4 * _ }
  FsMULd   fs1, fs2, fd # { fd = 4 * _ }
  FdMULq   fs1, fs2, fd # { fs1 = 2 * _, fs2 = 2 * _, fd = 4 * _ }

  fcompares fs1, fs2
  fcompared fs1, fs2 # { fs1 = 2 * _, fs2 = 2 * _ }
  fcompareq fs1, fs2 # { fs1 = 4 * _, fs2 = 4 * _ }
```

2.8 Miscellany

The remaining instructions don't belong to any particular grouping.

31a ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁30b 31b▷

```
constructors
  NOP
  FLUSH address
  JMPL  address, rd
  RETT  address
  trap  address
  UNIMP imm22
```

The SPARC architecture manual defines `sethi` such that it destroys the least significant ten bits on encoding. Therefore, no single bi-directional defintion of `sethi` can be written without loss of information. Our solution is to provide two constructors: `sethi` and `decode_sethi`. `sethi` encodes a `SETHI` instruction by ignoring the 10 low-order bits of `val`; `sethi` should be used for encoding. The toolkit requires that all bits in a pattern be constrained; `decode_sethi` constrains the 10 low-order bits of `val` and should be used for decoding.

31b ⟨*pattern and constructor specifications* 19a⟩+≡ (18a) ◁31a

```
constructors
  sethi        "%hi("n")", rd                  is SETHI & rd & imm22 = n@[10:31]
  decode_sethi "%hi("n")", rd { n@[0:9] = 0 } is SETHI & rd & imm22 = n@[10:31]
```

2.9 Synthetic instructions

The synthetic instructions are defined on pages 84–85 in the SPARC manual; their definitions appear below. When dealing with overloaded instructions like `call`, `mov`, and `clr`, we've reserved the standard name (e.g., `call`) for the most common variant, using either semi-mnemonic names (e.g., `movr` fopr move-register or `clrw` for clear-word, `calla` for call-address) or names with trailing underscores (e.g., `???_`) for other variants.

32 ⟨*sparc-synth.spec* 24a⟩+≡ ◁24a 33a▷

```
constructors
  jmp address          is JMPL (address, "%g0")
  calla address        is JMPL (address, "%o7")
  tst  rs2              is ORcc ("%g0", rmode(rs2), "%g0")
  ret                   is JMPL (dispA("%i7",8), "%g0")
  retl                  is JMPL (dispA("%o7",8), "%g0")
  restore_              is RESTORE ("%g0", rmode("%g0"), "%g0")
  save_                 is SAVE("%g0", rmode("%g0"), "%g0")
  not   rd              is XNOR(rd,    rmode("%g0"), rd)
  not2  rs1, rd         is XNOR(rs1,   rmode("%g0"), rd)
  neg   rd              is SUB ("%g0", rmode(rd),    rd)
  neg2  rs2, rd         is SUB ("%g0", rmode(rs2),   rd)
  inc   val, rd         is ADD (rd, imode(val), rd)
  inccc val, rd         is ADDcc (rd, imode(val), rd)
  dec   val, rd         is SUB (rd, imode(val), rd)
  deccc val, rd         is SUBcc (rd, imode(val), rd)
  cmp rs1, reg_or_imm   is SUBcc(rs1, reg_or_imm, "%g0")   # must follow dec
  btst reg_or_imm, rs1  is ANDcc(rs1, reg_or_imm, "%g0")
  bset reg_or_imm, rd   is OR  (rd, reg_or_imm, rd)
  bclr reg_or_imm, rd   is ANDN(rd, reg_or_imm, rd)
  btog reg_or_imm, rd   is XOR (rd, reg_or_imm, rd)
  mov  reg_or_imm, rd   is OR  ("%g0", reg_or_imm, rd)
  movr rs2, rd          is OR  ("%g0", rmode(rs2),   rd)
  clr  rd               is OR  ("%g0", rmode("%g0"), rd)
  clrw [address]       is ST  ("%g0", address)
  clrb [address]       is STB ("%g0", address)
  clrh [address]        is STH ("%g0", address)
```

Some of the constructors applied in the output patterns use `reg_or_imm` or `address` operands. When such a value is a literal constant (e.g., register `%g0`), we apply either the `imode` or `rmode` constructor to it to get a value of type `reg_or_imm` (e.g., `rmode("%g0")`).

The SPARC has only one conditionally assembled instruction. The `set` instruction has three ways to load an immediate value `val` into the register `rd`. If the low-order 10 bits of `val` are zero, use `sethi` to load the high-order 20 bits. If `val` fits in 13 signed bits, `OR` it with register `%g0`, which is always zero. In the general case, use `sethi` to load the high-order bits into `rd` and `OR` to or in the low-order bits.

33a ⟨*sparc-synth.spec* 24a⟩+≡ ◁32

```
constructors
  set val, rd
    when { val@[0:9] = 0 }  is  sethi(val, rd)
    otherwise               is  OR("%g0", imode(val), rd)
    otherwise               is  sethi(val, rd); OR(rd, imode(val@[0:9]), rd)
```

Again, because `set` loses information, we must specify `decode_set` for use in matching statements. This is really gross. Can anything be done? Having tried to use it in a SPARC disassembler, NR is now convinced that this constructor is always useless, and he has deleted it from the spec.

33b ⟨*useless junk* 33b⟩≡

```
constructors
  decode_set val, rd
    when { val@[0:9] = 0 }     is  decode_sethi(val, rd)
    when { val = val@[0:12]! } is  OR("%g0", imode(val), rd)
    otherwise                  is  decode_sethi(val, rd); OR(rd, imode(val@[0:9]), rd)
```

This code is written to file `useless junk`.

2.10 Application-specific specifications for the `lcc` compiler

`lcc` uses set, load, and store constructors that operate on relocatable addresses instead of SPARC operands of type `address`.

33c ⟨*lcc-sparc.spec* 33c⟩≡ 33d ▷

```
constructors
  setr raddr, rd  is  set(raddr, rd)
#  ldr  raddr, rd  is  sethi(raddr, "%g1"); ld(dispA("%g1", raddr@[0:9]), rd)
#  str  rd, raddr  is  sethi(raddr, "%g1"); st(rd, dispA("%g1", raddr@[0:9]))
```

This code is written to file `lcc-sparc.spec`.

`lcc` also supports a synthetic doubleword load and store operating on base-displacement addresses only.

33d ⟨*lcc-sparc.spec* 33c⟩+≡ ◁33c 34a ▷

```
constructors
  ld2f [rs1+disp!], fd  is  ldf(dispA(rs1, disp), fd); ldf(dispA(rs1, disp+4), fd+1)
  st2f fd, [rs1+disp!]  is  stf(fd, dispA(rs1, disp)); stf(fd+1, dispA(rs1, disp+4))
```

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

34a ⟨*lcc-sparc.spec* 33c⟩+≡ ◁33d

```
fields of addrtoken (32) addr32 0:31
placeholder for addrtoken is addr32 = 7
constructors
  emit_raddr raddr is addr32 = raddr
```

As on the MIPS, `mld` doesn't want to pay the overhead of checking register numbers:

34b ⟨*sparc-regs.spec* 34b⟩≡ 35c▷

```
fieldinfo [ rs1 rs2 rd fd fs1 fs2 cd ] is [ guaranteed ]
```

This code is written to file `sparc-regs.spec`.

2.11 Application-specific specifications for the `mld` linker

`mld`, defines set, load, and store constructors that operate on relocatable addresses instead of SPARC operands of type `address`.

34c ⟨*mld-sparc.spec* 34c⟩≡ 34d▷

```
constructors
  setr raddr, rd  is  set(raddr, rd)
  ldr  raddr, rd  is  sethi(raddr, "%o2"); ld(dispA("%o2", raddr@[0:9]), rd)
  str  rd, raddr  is  sethi(raddr, "%o2"); st(rd, dispA("%o2", raddr@[0:9]))
```

This code is written to file `mld-sparc.spec`.

`mld` uses the synonyms we've defined for the branch instructions.

34d ⟨*mld-sparc.spec* 34c⟩+≡ ◁34c 34e▷

```
constructors
  branch.synonyms^a target  { target = L + 4 * disp22! }  is
        L: branch.synonyms & a & disp22
```

Like the load and store instructions, the SPARC call instruction is overloaded; it accepts either a relocatable address or an SPARC-style `address`. The first style, which we've named `call`, uses the real `CALL` instruction, the second, which we've named `calla`, uses `JMPL`. Because `mld` generates code using a bottom-up tree walk, it decides which kind of call instruction it wants before it's actually ready to emit the call. `mld` wants to make up a cookie identifying the call, but doesn't emit the call until the cookie is eaten. The toolkit supports this trick using typed constructors; the cookie is a value of type `Function`.

34e ⟨*mld-sparc.spec* 34c⟩+≡ ◁34d 35a▷

```
constructors
  relocfun target    : Function  is  call(target)
  addrfun  address : Function  is  calla(address)
```

`relocfun` and `addrfun` create the cookies. It looks like these constructors emit the calls, but actually they don't have any side effects; the just return patterns representing the instructions to be emitted. Actually emitting such an instruction is the job of the `_call` constructor:

35a ⟨*mld-sparc.spec* 34c⟩+≡ ◁34e 35b▷

```
constructors
  _call Function is Function
```

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

35b ⟨*mld-sparc.spec* 34c⟩+≡ ◁35a

```
fields of addrtoken (32) addr32 0:31
placeholder for addrtoken is addr32 = 7
constructors
  emit_raddr raddr is addr32 = raddr
```

As on the MIPS, `mld` doesn't want to pay the overhead of checking register numbers:

35c ⟨*sparc-regs.spec* 34b⟩+≡ ◁34b

```
fieldinfo [ rs1 rs2 rd fd fs1 fs2 cd ] is [ guaranteed ]
```

2.12 Validating against the SPARC assembler

As on the MIPS, the same assembly name is used for integer and floating-point variants of an instruction. The assembler uses the operand names to disambiguate.

35d ⟨*sparc-names.spec* 35d⟩≡ 36a▷

```
assembly component
  decode_{*}              is $1
  {LDF,LDFSR,LDC,LDCSR}   is LD
  {LDDF,LDDC}             is LDD
  {STF,STFSR,STC,STCSR}   is ST
  {STDF,STDFQ,STDC,STDCQ} is STD
  {RDY,RDASR,RDPSR,RDWIM,RDTBR} is RD
  {WRY,WRASR,WRPSR,WRWIM,WRTBR} is WR
  {not,neg}2              is $1
  {set,ld,st,mov}r        is $1
  {ld2,st2}f              is $1
  {call}a                 is $1
  SWAP.                   is SWAP
  {*}{_,__}               is $1
  _{*}                    is $1
```

This code is written to file `sparc-names.spec`.

We need the extra syntax so that unbracketed `address` operands are accepted by the assembler.

36a ⟨*sparc-names.spec* 35d⟩+≡ ◁35d

```
assembly syntax
  absoluteA "%g0 + " simm13!
```

The suggested assembly syntax for several SPARC instructions is not recognized by the SunOS assembler, so those instructions are discarded. `emit_raddr`, `ldr`, and `str` are application-specific and are not in the SPARC assembly language.

36b ⟨*sparc-check.spec* 36b⟩≡

```
discard FLUSH RDASR WRASR STBAR RETT emit_raddr ldr str decode_sethi
```

This code is written to file `sparc-check.spec`.

We don't have to put any special headers in assembly source used by the checker.

36c ⟨*sparc-checker.s* 36c⟩≡

This code is written to file `sparc-checker.s`.

3 Alpha instruction-set specification

The Alpha lies somewhere between the MIPS and SPARC in complexity. It has one structured operand, which can be either an immediate or register value. The Alpha is a 64-bit architecture, but all instructions are 32 bits wide.

Readers shouldn't rely completely on this specification, because validation identified discrepancies in the assembler's treatment of the following instructions:

```
jmp jsr ldf ldg lds
```

The hint seems off in the jump instructions, and the assembler is expanding these floating-point loads into two-instruction sequences. We haven't yet resolved these discrepancies.

3.1 Fields

As usual for RISC, we have a single token, and the core specification has the normal structure.

37 ⟨*alpha-core.spec* 37⟩≡
  `fields of instruction (32)` ⟨*field specifications* 38a⟩
  ⟨`fieldinfo` *specifications* 38c⟩
  ⟨*pattern and constructor specifications* 39c⟩

This code is written to file `alpha-core.spec`.

According to page 1-4 of the manual (Sites and Witek 1995),[4] the Alpha has four basic formats, which we show here in the order opposite to that in the manual:

31 26	25 21	20 16	15 5	4 0	
Opcode	RA	RB	Function	RC	Operate format
Opcode	RA	RB	Memory displacement		Memory format
Opcode	RA	Branch displacement			Branch format
Opcode	Number or PAL function				PALcode format

The operate format comes in both integer and floating-point flavors, and the two flavors use different register sets. We use one token to specify all formats, and as usual, the real story is a bit more complicated—in particular, the area from bits 0–20 can be broken up in a variety of ways. These field specifications should be suggestive:

38a ⟨*field specifications* 38a⟩≡ (37) 38b▷

```
opcode_ 26:31   ra 21:25 rb 16:20 sbz 13:15 opfmt 12:12 func 5:11   rc 0:4
                         imm 13:20
                fa 21:25 fb 16:20 fpfunc 5:15                       fc 0:4
                                  mdisp 0:15
                                  mop 14:15 mhint 0:13
                         bdisp 0:20
                palfunc 0:25
```

Our spec[5] uses a couple of combinations not shown in the table. The "function" code for integers is split into `func` for functions, and `opfmt` to distinguish register from immediate operands. Immediate operands live in `imm`, and when register operands are used `sbz` Should Be Zero. The jump instructions split `mdisp` into an operation code and a hint.

It's easier to specify the integer and floating-point function codes by splitting them into two parts.

38b ⟨*field specifications* 38a⟩+≡ (37) ◁38a

```
flo 5:8 fhi 9:11 fplo 5:10 fphi 11:15
```

The registers have two sets of names; we've chosen the ones that obviously correspond to register numbers.

38c ⟨`fieldinfo` *specifications* 38c⟩≡ (37) 43c▷

```
fieldinfo
  [ ra rb rc ] is [ ⟨properties of integer-register fields 39a⟩ ]
  [ fa fb fc ] is [ ⟨properties of floating-point fields 39b⟩ ]
```

[4]References to the manual are a bit confused because MFF had the second edition and NR had the first edition

[5]Unfortunately `opcode` is a reserved word, so we use "`opcode_`".

39a ⟨*properties of integer-register fields* 39a⟩≡ (38c)

```
  names [ r0  r1  r2  r3  r4  r5  r6  r7
          r8  r9  r10 r11 r12 r13 r14 r15
          r16 r17 r18 r19 r20 r21 r22 r23
          r24 r25 r26 r27 r28 r29 r30 r31 ]
```

39b ⟨*properties of floating-point fields* 39b⟩≡ (38c)

```
  names [ f0  f1  f2  f3  f4  f5  f6  f7
          f8  f9  f10 f11 f12 f13 f14 f15
          f16 f17 f18 f19 f20 f21 f22 f23
          f24 f25 f26 f27 f28 f29 f30 f31 ]
```

3.2 Integer opcodes

The opcodes are specified in in Appendix C (Sites and Witek 1995). Table C-5 includes all the load, store and branch opcodes.

39c ⟨*pattern and constructor specifications* 39c⟩≡ (37) 39d ▹

```
  patterns
    [ call_pal  _      _     _      _    _     _     _
      lda       ldah   _     ldq_u  _    _     _     stq_u
      inta      intl   ints  intm   _    fltv  flti  fltl
      misc      pal19  jsrs  pal1b  _    pal1d pal1e pal1f
      ldf       ldg    lds   ldt    stf  stg   sts   stt
      ldl       ldq    ldl_l ldq_l  stl  stq   stl_c stq_c
      br        fbeq   fblt  fble   bsr  fbne  fbge  fbgt
      blbc      beq    blt   ble    blbs bne   bge   bgt  ]
    is opcode_ = { 0 to 63 }
```

(In NR's edition, the transpose of this table appears as C-9.)

We group by assembly syntax and encoding semantics.

39d ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◃39c 40 ▹

```
  patterns
    ldst    is lda | ldah | ldl | ldq | ldq_u | ldl_l | ldq_l
                        | stl | stq | stl_c | stq_c | stq_u
    fldst   is ldf | ldg | lds | ldt | stf | stg | sts | stt
    branch  is br | bsr | blbc | beq | blt | ble | blbs | bne | bge | bgt
    fbranch is fbeq | fblt | fble | fbne | fbge | fbgt
    [ jmp jsr ret jsr_co ] is jsrs & mop = {0 to 3}
    jump_hint    is jmp | jsr
    jump_predict is ret | jsr_co
```

If transcribed directly, the opcode table C-7 (C-5 in NR's edition) is sparse and confusing, so we separate it into sub-tables of related opcodes and use the `flo`/`fhi` trick.

40 ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁39d 41a▷

```
patterns
  arith is any of
    [ addl   s4addl subl   s4subl _      cmpbge
      _      s8addl _      s8subl cmpult _
      addq   s4addq subq   s4subq cmpeq  _
      _      s8addq _      s8subq cmpule _
      addlv  _      sublv  _      cmplt  _
      addqv  _      subqv  _      cmple  _  ],
  which is opcode_ = 0x10 &
           fhi = [ 0 1 2 3 4 6 ] & flo = [ 0x0 0x2 0x9 0xb 0xd 0xf ]

  logical is any of
    [ and _       _       bic
      _   cmovlbs cmovlbc _
      bis cmoveq  cmovne  ornot
      xor cmovlt  cmovge  eqv
      _   cmovle  cmovgt  _ ],
  which is opcode_ = 0x11 &
           fhi = [ 0 1 2 4 6 ] & flo = [ 0x0 0x4 0x6 0x8 ]

  byteops is any of
    [ _   _      mskbl _     extbl  _     _   _     insbl _
      _   _      mskwl _     extwl  _     _   _     inswl _
      _   _      mskll _     extll  _     _   _     insll _
      zap zapnot mskql srl   extql  _     sll _   insql sra
      _   _      mskwh _     _      inswh _   extwh _     _
      _   _      msklh _     _      inslh _   extlh _     _
      _   _      mskqh _     _      insqh _   extqh _     _  ],
  which is opcode_ = 0x12 &
           fhi = [ 0 1 2 3 5 6 7 ] & flo = [ 0 1 2 4 6 7 9 10 11 12 ]

  mulops is any of [ mull mullv mulq mulqv umulh ],
    which is intm & func = [ 0x00 0x40 0x20 0x60 0x30 ]
```

3.3 Structured operands

Integer register-to-register instructions have two operand formats, in which one of the source operands is an integer literal or a register. The encoding is very much like the SPARC, except instead of unused bits the Alpha has an `sbz` field, which should be zero. Since the Alpha doesn't seem to have a special terminology for these operands, we reuse the SPARC terminology.

41a ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁40 41b▷

```
  constructors
    imode imm : reg_or_imm  is  opfmt = 1 & imm
    rmode rb  : reg_or_imm  is  opfmt = 0 & sbz = 0 & rb
```

3.4 Integer instructions

The integer load/store instructions appear first (Sites and Witek 1995, p4-4).

41b ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁41a 41c▷

```
  constructors
    ldst  ra, mdisp!(rb)
```

The control instructions begin on page 4-16 (Sites and Witek 1995). The `branch` instructions are simliar to the MIPS branch instructions. The `jmp` and `jsr` instructions take a 14-bit "hint[6]", which is used to compute the predicted target of the jump as $PC + 4 \times hint$.[7]The `ret` and `jsr_co` instructions ignore the hint and use a prediction implementation stack, if there is one. In that case the hint is reserved for use by software and is required to be 0 or 1 (page 4-21). We could specify these two pairs separately, but we follow the manual, which defines them together.

41c ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁41b 41d▷

```
  relocatable target
  placeholder for instruction is call_pal & palfunc = 0  # halt (privileged)
  constructors
    branch ra, target { target = L + 4 * bdisp! } is branch & ra & bdisp; L: epsilon
    jump_hint ra, (rb), mhint
    proc    "0" : Return is mhint = 0
    nonproc "1" : Return is mhint = 1
    jump_predict ra, (rb), Return
```

We provide alternate versions that take an address and encode the hint:

41d ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁41c 42a▷

```
  constructors
    jump_hint^"*" ra, (rb), target { target = L + 4 * mhint }
          is  L: jump_hint & ra & rb & mhint
```

It's not clear whether these are useful.

[6] We're having a problem with the coding of these hints, as noted above.
[7] The PC is the updated PC.

The integer arithmetic instructions begin on page 4-24 (4-22 in NR's edition) and can be specified in one line. Hurray for orthogonality!

42a ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁41d 42b▷

```
patterns alu is arith | logical | byteops | mulops
constructors
  alu ra, reg_or_imm, rc
```

3.5 Floating-point opcodes

The floating-point opcodes are specified in Table C-7 of NR's edition. Many floating point instructions take suffixes, called "qualifiers", which specify a variety of rounding and exception modes. The suffixes are described beginning on page 4-61. We've aggressively factored the table on page C-4.

First we attack the row labels, then the column labels. For the row labels, we look at column /C, where the column code is zero, and for the row labels, we look at row ADDS, where the row code is zero. For the row labels, we have to slice manually (or shift right 6 bits). In a better world, that operation would be in our specification language.

42b ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁42a 42c▷

```
patterns           [ adds addt cmpteq cmptlt cmptle cmptun
                     cvtqs cvtqt cvtts divs divt muls mult subs subt cvttq ]
  is flti & fplo = [ 0x00 0x20 0x25 0x26 0x27 0x24
                     0x3c 0x3e 0x2c 0x03 0x23 0x02 0x22 0x01 0x21 0x2f ]
  fpqual is any of
  [ none c m d u uc um ud
    su suc sum sud sui suic suim suid ],
  which is flti & fphi = [ 0x02 0x00 0x01 0x03 0x06 0x04 0x05 0x07
                           0x16 0x14 0x15 0x17 0x1e 0x1c 0x1d 0x1f ]
```

Having done this, we now group both qualifiers and opcodes according to the rows of the tables that have the same columns filled in.

42c ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁42b 43a▷

```
patterns
  cmpqual is none | su
  cvtqual is none | c | m | d | sui | suic | suim | suid
  fpop  is adds | addt | divs | divt | muls | mult | subs | subt
  cmpop is cmpteq | cmptlt | cmptle | cmptun
  cvtop is cvtqs | cvtqt
  qqqual is any of [ qq qqc qqm qqd v vc vm vd
                     sv svc svm svd svi svic svim svid ],
  which is flti & fphi = [ 0x02 0x00 0x01 0x03 0x06 0x04 0x05 0x07
                           0x16 0x14 0x15 0x17 0x1e 0x1c 0x1d 0x1f ]
```

The qualifiers for `cvttq` mostly have different names. The `qq` trick enables us to avoid collisions with names of qualifiers used in `fpqual`.

3.6 Floating-point instructions

The floating-point load/store and branch instructions are similar to the integer instructions.

43a ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁42c 43b▷
```
  constructors
    fldst         fa, mdisp(rb)
    fbranch       fa, target { target = L + 4 * bdisp! }
                          is fbranch & fa & bdisp; L: epsilon
```

We use compound opcodes to define instructions that take qualifiers.

43b ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁43a 43d▷
```
  constructors
    fpop^fpqual    fa, fb, fc
    cvtts^fpqual       fb, fc
    cvtop^cvtqual      fb, fc
    cmpop^cmpqual  fa, fb, fc
    cvttq^qqqual       fb, fc
```

3.7 PAL instructions

The Privileged Access Library provides a large set of hardware and firmware control functions that can be used under the OpenVMS, DEC OSF/1, and Windows NT operating systems. We define only those PAL functions that must be recognized by all implementations.

43c ⟨`fieldinfo` *specifications* 38c⟩+≡ (37) ◁38c
```
  fieldinfo palfunc is
    [ sparse
      [ halt = 0, draina = 2, cserve = 0xa, swppal = 0xb,
        bpt = 0x80, bugchk = 0x81, imb = 0x86,
        rdunique = 0x9e, wrunique = 0x9f, gentrap = 0xaa ] ]
```

43d ⟨*pattern and constructor specifications* 39c⟩+≡ (37) ◁43b
```
  constructors
   call_pal palfunc
```

3.8 Synthetics

Section C.7 of the DEC OSF/1 Assembly Language Programmer's Guide lists some of the expansions of synthetic instructions, but not all of them. Being lazy, we define just a handful.

44a ⟨*alpha-synth.spec* 44a⟩≡
```
constructors
  andnot ra, rb, rc          is  bic(ra, rmode(rb), rc)
  clr    rc                  is  bis(r31, rmode(r31), rc)
  mov    reg_or_imm, rc      is  bis(r31, reg_or_imm, rc)
  nop                        is  bis(r31, rmode(r31), r31)
  not    rb, rc              is  ornot(r31, rmode(rb), rc)
  or     ra, reg_or_imm, rc  is  bis(ra, reg_or_imm, rc)
```
This code is written to file `alpha-synth.spec`.

3.9 Span-dependent branches

Dirty-trick central. First, opposite branches:

44b ⟨*alpha-span.spec* 44b⟩≡
```
patterns xbranch is any of
  [ xbeq xbne xble xblt xbge xbgt xblbc xblbs ], which is
  [  bne  beq  bgt  bge  blt  ble  blbs  blbc ]
constructors
  xbranch ra, target { target = L + 4 * bdisp! } is xbranch & ra & bdisp; L: epsilon
constructors
  "span"^branch ra, target
     where { } is branch ra, target
     otherwise is
       "x"^branch ra, L;
       bis(r31, imode(target@[0:7]), 1); # bogus -- must set 64 bits
       jmp(r31, 1, 0);
     L: epsilon
```
This code is written to file `alpha-span.spec`.

3.10 Checking against the Digital OSF/1 Assembler

Apparently, unlike in the manual, qualifiers in OSF/1 assembler are not preceded by a slash, so all we need to do is fix a couple of names.

44c ⟨*alpha-names.spec* 44c⟩≡ 45a▷
```
assembly component
  none*  is ""
  qq{*} is $1
  jsr_co is "jsr_coroutine"
```
This code is written to file `alpha-names.spec`.

The OSF assembler uses the usual DEC conventions about register names:

45a ⟨*alpha-names.spec* 44c⟩+≡ ◁44c

```
assembly operand
  [ ra rb rc ] is "$%d"
  [ fa fb fc ] is "$%s"
```

There's a bug in the toolkit that prevents us from checking `call_pal`.

45b ⟨*alpha-check.spec* 45b⟩≡

```
discard call_pal
discard jump_hint^"*"          # not really in the assembly language
discard mov nop                # OSF/1 assembler uses different synthetics
```

This code is written to file `alpha-check.spec`.

When assembling code for checking, we might use `$at`, we don't want instructions out of order, and we want screaming and shouting when the assembler expands something into a multi-instruction sequence.

45c ⟨*alpha-checker.s* 45c⟩≡

```
.set noat
.set noreorder
.set nomacro
```

This code is written to file `alpha-checker.s`.

The result of the checking is OK except for the discrepancies noted above. The other oddity is that the assembler expands multiplication by constants into a sequence of add and subtract operations.

3.11 Guaranteeing registers

Most applications will want to make the usual assumptions about the values of register operands.

45d ⟨*alpha-regs.spec* 45d⟩≡

```
fieldinfo [ra rb rc fa fb fc] is [guaranteed]
```

This code is written to file `alpha-regs.spec`.

4 Intel Pentium instruction-set specification

This specification describes the Intel Pentium (Intel 1993). 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.[8]

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 4.6. 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:

46a ⟨*pentium32.spec* 46a⟩≡ (86b 87f)
  `keep` ⟨*32-bit constructors* 55b⟩ ⟨*floating-point constructors* 64h⟩

46b ⟨*pentium-int.spec* 46b⟩≡
  `discard` ⟨*floating-point constructors* 64h⟩

This code is written to file `pentium-int.spec`.

One-Byte Opcode Map (page 0)

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES
1	ADC						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS
2	AND						SEG	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=ES	
3	XOR						SEG	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=SS	
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	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
7	Short-displacement jump on condition (Jb)							
	JO	JNO	JB/JNAE/J	JNB/JAE/J	JZ	JNZ	JBE	JNBE
8	Immediata Grp1		MOVB*	Grp1	TEST		XCHG	
	Eb,Ib	Ev,Iv	AL,immed	Ev,Ib	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv
9	NOP	XCHG word to double-word register with eAX						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV				MOVSB	MOVsw	CMPSB	CMPSW
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX	Xb,Yb	Xv,Yv	Xb,Yb	Xv,Yv
B	MOV immediate byte into register							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp2a		RET near		LES	LDS	MOV	
	Eb,Ib	Ev,Ib	Iw		Gv,Mp	Gv,Mp	Eb,Ib	Ev,Iv
D	Shift Grp2				AAM	AAD	*	XLAT
	Eb,1	Ev,1	Eb,CL	Ev,CL				
E	LOOPNE	LOOPE	LOOP	JCXZ/JEC	IN		OUT	
	Jb	Jb	Jb	Jb	AL,Ib	AX,Ib	Ib,AL	Ib,eAX
F	LOCK	*	REPNE	REP	HLT	CMC	Unary Grp3	
				REPE			Eb	Ev

Figure 2: Pentium opcodes (page 0)

One-Byte Opcode Map (page 1)

	8	9	A	B	C	D	E	F
0	OR						PUSH	2-byte
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	CS	escape
1	SBB						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	DS	DS
2	SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
3	CMP						SEG	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=DS	
4	DEC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP into general register							
	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
7	Short-displacement jump on condition (Jb)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	MOV				MOV	LEA	MOV	POP
	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		
A	TEST		STOSB	STOSW/D	LODSB	LODSW/D	SCASB	SCASW/D
	AL,Ib	eAX,Iv	Yb,AL	Yv,eAX	AL,Xb	eAX,Xv	AL,Yb	eAX,Yv
B	MOV 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	ESC (Escape to coprocessor instruction set)							
E	CALL	JMP			IN		OUT	
	Jv	Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC	INC/DEC
							Grp4	Grp5

Figure 3: 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 4.6 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:

49 ⟨*pentium-core.spec* 49⟩≡
  ⟨*field specs* 50a⟩
  `patterns` ⟨*patterns for integer opcodes* 50c⟩
  ⟨*pattern specs for other patterns* 58⟩
  ⟨*prefix assignments* 59a⟩
  ⟨*placeholders* 50b⟩
  `relocatable reloc`
  ⟨*constructors for displacements* 62g⟩
  ⟨*constructors for effective addresses* 55a⟩
  ⟨*arithmetic constructors* 61a⟩
  `constructors`
  ⟨*alphabetical constructors* 61c⟩

This code is written to file `pentium-core.spec`.

[8] 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.

4.1 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 **group***x* opcodes.

4.2 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 2 and 3, which represent the "one-byte opcode map" from pages A-5 and A-6 of the Intel manual (Intel 1993). 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.

50a ⟨*field specs* 50a⟩≡ (49 86d) 51c▷
```
fields of opcodet (8) row 4:7 col 0:2 page 3:3
                                ⟨more fields of opcode 51b⟩
```

50b ⟨*placeholders* 50b⟩≡ (49 86d) 54e▷
```
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:

50c ⟨*patterns for integer opcodes* 50c⟩≡ (49 86d) 50d▷
```
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.

50d ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁50c 51a▷
```
[ 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.

51a ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁50d 51h▷
```
regops is any of [ INC  DEC
                   PUSH POP ], which is row = [4 5] & page = [0 1]
```

51b ⟨*more fields of* `opcode` 51b⟩≡ (50a) 51d▷
```
r32 0:2
```

51c ⟨*field specs* 50a⟩+≡ (49 86d) ◁50a 51e▷
```
fieldinfo r32 is [names [ eAX eCX eDX eBX eSP eBP eSI eDI ]]
```

51d ⟨*more fields of* `opcode` 51b⟩+≡ (50a) ◁51b 51f▷
```
sr16 0:2
```

51e ⟨*field specs* 50a⟩+≡ (49 86d) ◁51c 51g▷
```
fieldinfo sr16 is [sparse [ cs=1, ss=2, ds=3, es=4, fs=5, gs=6 ] ]
```

51f ⟨*more fields of* `opcode` 51b⟩+≡ (50a) ◁51d 52d▷
```
r16 0:2
```

51g ⟨*field specs* 50a⟩+≡ (49 86d) ◁51e 52e▷
```
fieldinfo r16 is [names [ AX CX DX BX SP BP SI DI ]]
```

Row 6:

51h ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁51a 51i▷
```
[ 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.

51i ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁51h 51j▷
```
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.

51j ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁51i 52a▷
```
[ 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.

52a ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◃51j 52b▹

```
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.

52b ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◃52a 52c▹

```
[ 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.

52c ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◃52b 52f▹

```
MOVib is row = 11 & page = 0
MOViv is row = 11 & page = 1
```

52d ⟨*more fields of* opcode 51b⟩+≡ (50a) ◃51f

```
r8 0:2
```

52e ⟨*field specs* 50a⟩+≡ (49 86d) ◃51g 54c▹

```
fieldinfo r8 is [names [ AL CL DL BL AH CH DH BH ]]
```

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

52f ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◃52c 52g▹

```
[ 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.

52g ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◃52f 53a▹

```
[ 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}
```

4.3 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:

53a ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁52g 53b▷
```
[ 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.

53b ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53a 53c▷
```
[ 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:

53c ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53b 53d▷
```
[ 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.

53d ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53c 53e▷
```
Jv   is esc2; row = 8
SETb is esc2; row = 9
```

Rows 10 and 11 are more madness:

53e ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53d 53f▷
```
[ 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.

53f ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53e 53g▷
```
[ 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.

53g ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53f 53h▷
```
[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:

53h ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53g 54a▷
```
[ 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:

54a ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁53h 54b▷
```
[ 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:

54b ⟨*patterns for integer opcodes* 50c⟩+≡ (49 86d) ◁54a
```
BSWAP is esc2; row = 12 & page = 1
```

4.4 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.

54c ⟨*field specs* 50a⟩+≡ (49 86d) ◁52e 54d▷
```
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
```

54d ⟨*field specs* 50a⟩+≡ (49 86d) ◁54c 56b▷
```
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 ] ]
```

54e ⟨*placeholders* 50b⟩+≡ (49 86d) ◁50b 56c▷
```
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.

55a ⟨*constructors for effective addresses* 55a⟩≡ (49 86d)

```
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]   : Mem                        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.

55b ⟨*32-bit constructors* 55b⟩≡ (46a) 61b▹

```
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:

56a ⟨*proposed constructors for effective addresses* 56a⟩≡

```
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
```

This code is written to file `proposed constructors for effective addresses`.

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.

56b ⟨*field specs* 50a⟩+≡ (49 86d) ◁54d

```
fields of I8   (8) i8  0:7
fields of I16 (16) i16 0:15
fields of I32 (32) i32 0:31
```

56c ⟨*placeholders* 50b⟩+≡ (49 86d) ◁54e

```
placeholder for I8  is HLT
placeholder for I16 is HLT; HLT
placeholder for I32 is HLT; HLT; HLT; HLT
```

4.5 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.[9] In the case at hand, every `Eaddr` begins with a Mod R/M token, so we can always write

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

[9] 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.

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 4.8.

58 ⟨*pattern specs for other patterns* 58⟩≡ (49 86d) 60a▷

```
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 ...
```

4.6 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`.

59a ⟨*prefix assignments* 59a⟩≡ (49 86d) 59b ▷

```
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.

59b ⟨*prefix assignments* 59a⟩+≡ (49 86d) ◁ 59a

```
patterns aw is AddrPrefix
         ad is epsilon
         av is aw | ad
```

4.7 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`.

60a ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁58 60b▷

```
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.

60b ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁60a 65c▷

```
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]
```

4.8 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 2 and 3; 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.

61a ⟨*arithmetic constructors* 61a⟩≡ (49 87d)

```
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 ...
```

61b ⟨*32-bit constructors* 55b⟩+≡ (46a) ◃55b 62b▹

```
arith^"iEAX" arithI^"d" arith^"mr"^od arith^"rm"^od
```

4.9 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 (Intel 1993). To generate a checker by the end of the millenium, we divide the spec into four parts and check each individually.

61c ⟨*alphabetical constructors* 61c⟩≡ (49)

⟨*alphabetical constructors A-F* 62a⟩
⟨*alphabetical constructors G-K* 69d⟩
⟨*alphabetical constructors L-Q* 70e⟩
⟨*alphabetical constructors R* 73c⟩
⟨*alphabetical constructors S-Z* 73h⟩

62a ⟨*alphabetical constructors A-F* 62a⟩≡ (61c 86e) 62d▷

```
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 ...
```

62b ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁61b 62f▷

```
AAA AAD AAM AAS
```

62c ⟨*names of pentium constructors* 62c⟩≡ (86b) 62e▷

```
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.

62d ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁62a 63a▷

```
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
```

62e ⟨*names of pentium constructors* 62c⟩+≡ (86b) ◁62c 63e▷

```
BOUND^ov BSF^ov BSR^ov BSWAP BT^ov BTi^ov BTC^ov BTCi^ov BTR^ov BTRi^ov BTS^ov BTSi^ov
```

62f ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁62b 62h▷

```
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.

62g ⟨*constructors for displacements* 62g⟩≡ (49 86d)

```
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
```

62h ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁62f 63d▷

```
rel32
```

63a ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁62d 63c▷
```
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 Eaddr    is  ov; (grp5; Eaddr) & 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.

63b ⟨*pentium-linux.spec* 63b⟩≡ (86b 87f) 65f▷
```
discard CALL.aP^ow CALL.aP^od
```

63c ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁63a 64a▷
```
# 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
```

63d ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁62h 64b▷
```
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.

63e ⟨*names of pentium constructors* 62c⟩+≡ (86b) ◁62e 64c▷
```
CMPXCHG8B CPUID CWD
```

64a ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁63c 64d▷
```
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
```

64b ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁63d 64e▷
```
DAA DAS DEC.Ev^od DEC^od DIV^"eAX"
```

64c ⟨*names of pentium constructors* 62c⟩+≡ (86b) ◁63e 64f▷
```
DAA DAS DEC.Eb
```

64d ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁64a 64g▷
```
ENTER   i16, i8!  is  ENTER; i16; i8
F2XM1
```

64e ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁64b 69e▷
```
ENTER F2XM1
```

64f ⟨*names of pentium constructors* 62c⟩+≡ (86b) ◁64c 69c▷
```
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`.

64g ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁64d 64i▷
```
FABS
FADD^Fmem   Mem  is  Fmem; Mem & FADD ...
FADD^Fstack idx  is  Fstack & ... (FADD & r_m = idx)
```

64h ⟨*floating-point constructors* 64h⟩≡ (46 87e) 64k▷
```
FABS FADD^Fmem FADD^Fstack
```

64i ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁64g 64j▷
```
FIADD^Fint  Mem  is  Fint; Mem & FIADD ...
FBLD        Mem  is  DF; Mem & FBLD
FBSTP       Mem  is  DF; Mem & FBSTP
```

64j ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁64i 65b▷
```
FCHS
```

64k ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁64h 65e▷
```
FCHS
```

65a ⟨*synthetics with* `WAIT` 65a⟩≡ (75e) 67d ▷

```
FCLEX            is  WAIT; FCLEX
```

65b ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 64j 65d ▷

```
FNCLEX           is  FCLEX
```

65c ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁ 60b 65k ▷

```
    patterns FCOMs is FCOM | FCOMP
```

65d ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 65b 65g ▷

```
    constructors
FCOMs^Fmem     Mem  is  Fmem; Mem & FCOMs ...           # includes FICOM, FICOMP
FCOMs^.ST.STi  idx  is  .ST.STi & ... (FCOMs & r_m = idx)
FCOMPP
FCOS
```

65e ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁ 64k 65h ▷

```
FCOMs^Fmem FCOMs^.ST.STi FCOMPP FCOS
```

65f ⟨*pentium-linux.spec* 63b⟩+≡ (86b 87f) ◁ 63b 70d ▷

```
discard FCOMs^.ST.STi
```

65g ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 65d 65i ▷

```
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)
```

65h ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁ 65e 66a ▷

```
FDECSTP FDIV^Fmem FDIV^Fstack FDIVR^Fmem FDIVR^Fstack
```

65i ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 65g 65j ▷

```
FFREE         idx  is  DD; FFREE & r_m = idx
```

65j ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 65i 65l ▷

```
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
```

65k ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁ 65c 67g ▷

```
    patterns FISTs is FIST | FISTP
```

65l ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁ 65j 66b ▷

```
    constructors
FISTs^FlsI  Mem  is  FlsI; Mem & FISTs
FISTP64     Mem  is  DF;   Mem & FSTP.ext
```

66a ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁65h 66c▷
```
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.*

66b ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁65l 66d▷
```
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
```

66c ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁66a 66e▷
```
FLD^FlsR FLD80 FLD.STi Fconstants FLDCW FLDENV
```

66d ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁66b 66f▷
```
FMUL^Fmem   Mem  is  Fmem; Mem & FMUL ...
FMUL^Fstack idx  is  Fstack & ... (FMUL & r_m = idx)
```

66e ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁66c 66g▷
```
FMUL^Fmem FMUL^Fstack
```

66f ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁66d 66h▷
```
FNOP
```

66g ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁66e 66i▷
```
FNOP
```

66h ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁66f 66j▷
```
FPATAN
FPREM
FPREM1
FPTAN
```

66i ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁66g 67a▷
```
FPATAN FPREM FPREM1 FPTAN
```

66j ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁66h 67b▷
```
FRNDINT
FRSTOR Mem  is DD; Mem & FRSTOR
```

67a ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁66i 67c▷

67b ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁66j 67f▷
```
FNSAVE Mem  is DD; Mem & FSAVE
```

67c ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁67a 67e▷

67d ⟨*synthetics with* `WAIT` 65a⟩+≡ (75e) ◁65a 67i▷
```
FSAVE  Mem  is WAIT; FNSAVE(Mem)
```

67e ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁67c 67j▷

67f ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁67b 67h▷
```
FSCALE
FSIN
FSINCOS
FSQRT
```

67g ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁65k 68g▷
```
   patterns
FSTs is FST | FSTP
FSTs.st is FST.st | FSTP.st
```

67h ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁67f 67k▷
```
   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
```

67i ⟨*synthetics with* `WAIT` 65a⟩+≡ (75e) ◁67d 67l▷
```
FNSTCW        Mem  is  WAIT; FSTCW(Mem)
```

67j ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁67e 67m▷
```
FSCALE FSIN FSINCOS FSQRT FSTs^FlsR FSTP80 FSTs.st^.STi FSTCW FNSTCW
```

67k ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁67h 67n▷
```
FSTENV        Mem  is  D9;   Mem & FSTENV
```

67l ⟨*synthetics with* `WAIT` 65a⟩+≡ (75e) ◁67i 68a▷
```
FNSTENV       Mem  is  WAIT; FSTENV(Mem)
```

67m ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁67j 68b▷
```
FSTENV FNSTENV
```

67n ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁67k 68c▷
```
FSTSW         Mem  is  DD;   Mem & FSTSW
FSTSW.AX
```

68a ⟨*synthetics with* `WAIT` 65a⟩+≡ (75e) ◁67l 68j▷
```
FNSTSW       Mem  is  WAIT; FSTSW(Mem)
FNSTSW.AX         is  WAIT; FSTSW.AX()
```

68b ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁67m 68d▷
```
FSTSW FSTSW.AX FNSTSW FNSTSW.AX
```

68c ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁67n 68e▷
```
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)
```

68d ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68b 68f▷
```
FSUB^Fmem FSUB^Fstack FSUBR^Fmem FSUBR^Fstack
```

68e ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁68c 68h▷
```
FTST
```

68f ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68d 68i▷
```
FTST
```

68g ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁67g 70f▷
```
   patterns FUCOMs is FUCOM | FUCOMP
```

68h ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁68e 68l▷
```
   constructors
FUCOMs  idx   is  DD; FUCOMs & r_m = idx
FUCOMPP
```

68i ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68f 68k▷
```
FUCOMs FUCOMPP
```

68j ⟨*synthetics with* `WAIT` 65a⟩+≡ (75e) ◁68a
```
FWAIT is WAIT
```

68k ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68i 68m▷
```
FWAIT
```

68l ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁68h 69a▷
```
FXAM
FXCH idx  is  FXCH & ... r_m = idx
FXTRACT
```

68m ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68k 69b▷
```
FXAM FXCH FXTRACT
```

69a ⟨*alphabetical constructors A-F* 62a⟩+≡ (61c 86e) ◁68l
```
FYL2X
FYL2XP1
```

69b ⟨*floating-point constructors* 64h⟩+≡ (46 87e) ◁68m
```
FYL2X FYL2XP1
```

69c ⟨*names of pentium constructors* 62c⟩+≡ (86b) ◁64f
```
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
```

69d ⟨*alphabetical constructors G-K* 69d⟩≡ (61c 86f) 69f▷
```
HLT
```

69e ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁64e 70a▷
```
HLT
```

69f ⟨*alphabetical constructors G-K* 69d⟩+≡ (61c 86f) ◁69d 70b▷
```
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
```

70a ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁69e 70c▷
```
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
```

70b ⟨*alphabetical constructors G-K* 69d⟩+≡ (61c 86f) ◁69f 70g▷
```
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)
JCXZ       reloc   is      JCXZ         ; rel8(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
```

70c ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁70a 71b▷
```
Jv^cond^od JMP.Jv^od JMP.Ap^od JMP.Ev^od JMP.Ep^od
```

70d ⟨*pentium-linux.spec* 63b⟩+≡ (86b 87f) ◁65f 72a▷
```
discard JMP.Ap^ow  JMP.Ap^ov
```

70e ⟨*alphabetical constructors L-Q* 70e⟩≡ (61c 87a) 71a▷
```
LAHF
LAR^ov  reg, Eaddr  is  ov; LAR; Eaddr & reg_opcode = reg ...
```

70f ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁68g 70h▷
```
   patterns lfp is LDS | LES | LFS | LGS | LSS
```

70g ⟨*alphabetical constructors G-K* 69d⟩+≡ (61c 86f) ◁70b
```
   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
```

70h ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁70f 72g▷
```
    patterns LOOPs is LOOP | LOOPE | LOOPNE
```

71a ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁70e 71c▷

```
    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
```

71b ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁70c 71d▷

```
  LAHF LAR^ov lfp^ov LEA^ov LEAVE LGDT LIDT LLDT LMSW LOCK LODSB LODSv^ov
```

71c ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁71a 72b▷

```
  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          r16, i16!   is  ow; MOViv & r16; 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
```

71d ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁71b 72c▷

```
  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
```

72a ⟨*pentium-linux.spec* 63b⟩+≡ (86b 87f) ◁70d 73g▷
```
discard  MOVSX.Gv.Ew MOVZX.Gv.Eb^ov MOVZX.Gv.Ew MOV.Ew.Sw
```

72b ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁71c 72d▷
```
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
```

72c ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁71d 72e▷
```
NEGb NEG^ov NOP NOTb NOT^ov
```

72d ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁72b 72f▷
```
# 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
```

72e ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁72c 73b▷
```
OUT.Ib.AL OUT.Ib.eAX^ov OUT.DX.AL OUT.DX.eAX^ov OUTSB OUTSv^ov
```

72f ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁72d 72h▷
```
POP.Ev^ov   Mem   is  ov; POP.Ev; Mem & reg_opcode = 0 ...
POP^ov      r32   is  ov; POP & r32
```

72g ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁70h 72i▷
```
   patterns POPs is POP.ES | POP.SS | POP.DS | POP.FS | POP.GS
            POPv is POPA | POPF
```

72h ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁72f 73a▷
```
   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
```

72i ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁72g 74a▷
```
   patterns PUSHs is PUSH.CS | PUSH.SS | PUSH.DS | PUSH.ES | PUSH.FS | PUSH.GS
            PUSHv is PUSHA | PUSHF
```

73a ⟨*alphabetical constructors L-Q* 70e⟩+≡ (61c 87a) ◁72h
```
    constructors
  PUSHs
  PUSHv^ov  is  ov; PUSHv
```

73b ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁72e 73d▷
```
POPs POPv^ov  PUSH.Ev^ov PUSH^ov PUSH.Ib PUSH.Iv^ow PUSH.Iv^od PUSHs PUSHv^ov
```

73c ⟨*alphabetical constructors R* 73c⟩≡ (61c 87b) 73e▷
```
# 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
```

73d ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁73b 73f▷
```
rot^bshifts rot^vshifts^ov rot^B.Ev.Ib^ov
```

73e ⟨*alphabetical constructors R* 73c⟩+≡ (61c 87b) ◁73c
```
RDMSR
REP
REPNE
RET
RET.far
RET.Iw      i16  is RET.Iw; i16
RET.far.Iw  i16  is RET.far.Iw; i16
RSM
```

73f ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁73d 73i▷
```
RDMSR REP REPNE RET RET.far RET.Iw RET.far.Iw RSM
```

73g ⟨*pentium-linux.spec* 63b⟩+≡ (86b 87f) ◁72a 75b▷
```
discard RDMSR
```

73h ⟨*alphabetical constructors S-Z* 73h⟩≡ (61c 87c) 74b▷
```
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
```

73i ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁73f 74c▷
```
SCASB SCASv^ov SETb^cond SGDT SIDT
```

74a ⟨*pattern specs for other patterns* 58⟩+≡ (49 86d) ◁72i

```
  patterns shdIb is SHRD.Ib | SHLD.Ib
           shdCL is SHRD.CL | SHLD.CL
```

74b ⟨*alphabetical constructors S-Z* 73h⟩+≡ (61c 87c) ◁73h 74d▷

```
  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
```

74c ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁73i 74e▷

```
shdIb^ov shdCL^ov SLDT SMSW STC STD STI STOSB STOSv^ov STR
```

74d ⟨*alphabetical constructors S-Z* 73h⟩+≡ (61c 87c) ◁74b 74f▷

```
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 ...
```

74e ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁74c 74g▷

```
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
```

74f ⟨*alphabetical constructors S-Z* 73h⟩+≡ (61c 87c) ◁74d 74h▷

```
VERR  Eaddr  is (grp6; Eaddr) & VERR
VERW  Eaddr  is (grp6; Eaddr) & VERW
```

74g ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁74e 75a▷

```
VERR VERW
```

74h ⟨*alphabetical constructors S-Z* 73h⟩+≡ (61c 87c) ◁74f 75c▷

```
WAIT
WBINVD
WRMSR
```

75a ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁74g 75d▷
```
WAIT WBINVD WRMSR
```

75b ⟨*pentium-linux.spec* 63b⟩+≡ (86b 87f) ◁73g
```
discard WRMSR
```

75c ⟨*alphabetical constructors S-Z* 73h⟩+≡ (61c 87c) ◁74h
```
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
```

75d ⟨*32-bit constructors* 55b⟩+≡ (46a) ◁75a
```
XADD.Eb.Gb XADD.Ev.Gv^ov XCHG^"eAX"^ov XCHG.Eb.Gb XCHG.Ev.Gv^ov XLATB
```

4.10 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`.

75e ⟨*pentium-synth.spec* 75e⟩≡
```
constructors
  ⟨synthetics with WAIT 65a⟩
```
This code is written to file `pentium-synth.spec`.

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

75f ⟨*mld-pentium.spec* 75f⟩≡ 75g▷
```
constructors
  _call_l reloc  : Function is  call_jvod(reloc)
  _call_rm Mem : Function is  call_evod(Mem)
  callfn Function is Function
```
This code is written to file `mld-pentium.spec`.

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

75g ⟨*mld-pentium.spec* 75f⟩+≡ ◁75f
```
fields of addrtoken (32) addr32 0:31
placeholder for addrtoken is addr32 = 7
constructors
  emit_raddr reloc is addr32 = reloc
```

4.11 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.

4.11.1 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 (Joy 1994). 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:

77a ⟨*example mappings* 77a⟩≡

```
assembly component
     add      is  add
     {"mrb","rmb"} is b
```

This code is written to file `example mappings`.

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.*

77b ⟨*pentium-linux-default.spec* 77b⟩≡ (86c)

```
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.

78 ⟨*pentium-linux-names.spec* 78⟩≡ (86c) 79▷

```
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.

79 ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁78 80a▷

```
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.

80a ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁79 80b▷

```
  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.

80b ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁80a 81a▷

```
  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
```

4.11.2 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.

81a ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁80b 81b▷

```
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`.

81b ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁81a 81c▷

```
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.

81c ⟨*pentium-linux-names.spec* 78⟩+≡ (86c) ◁81b

```
assembly operand
    dx    is "%%dx"
    ax    is "%%ax"
```

4.11.3 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.

82 ⟨*pentium-linux-syntax.spec* 82⟩≡ (86c) 83▹

```
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
```

83 ⟨*pentium-linux-syntax.spec* 82⟩+≡ (86c) ◁82 84▷

```
  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
```

84 ⟨*pentium-linux-syntax.spec* 82⟩+≡ (86c) ◁83 85▷

```
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!, r16
  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
```

85 ⟨*pentium-linux-syntax.spec* 82⟩+≡ (86c) ◁84 86a▷

```
    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.

86a ⟨*pentium-linux-syntax.spec* 82⟩+≡ (86c) ◁85
```
assembly syntax
  Indir       (reg)
  Disp32      d(reg)
  Index       (base,index,ss)
  Index32     d(base,index,ss)
  ShortIndex  d(,index,ss)
```

4.12 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.

86b ⟨*i486-linux.spec* 86b⟩≡
  ⟨*pentium32.spec* 46a⟩
  `discard` ⟨*names of pentium constructors* 62c⟩
  ⟨*pentium-linux.spec* 63b⟩

This code is written to file `i486-linux.spec`.

Names for generating assembly emitters are specified in several chunks.

86c ⟨*pentium-names.spec* 86c⟩≡
  ⟨*pentium-linux-names.spec* 78⟩
  ⟨*pentium-linux-default.spec* 77b⟩
  ⟨*pentium-linux-syntax.spec* 82⟩

This code is written to file `pentium-names.spec`.

86d ⟨*header.spec* 86d⟩≡ (86 87)
  ⟨*field specs* 50a⟩
  `patterns` ⟨*patterns for integer opcodes* 50c⟩
  ⟨*pattern specs for other patterns* 58⟩
  ⟨*prefix assignments* 59a⟩
  ⟨*placeholders* 50b⟩
  ⟨*constructors for displacements* 62g⟩
  ⟨*constructors for effective addresses* 55a⟩

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

86e ⟨*pentium-AF.spec* 86e⟩≡
  ⟨*header.spec* 86d⟩
  ⟨*alphabetical constructors A-F* 62a⟩

This code is written to file `pentium-AF.spec`.

86f ⟨*pentium-GK.spec* 86f⟩≡
  ⟨*header.spec* 86d⟩
  ⟨*alphabetical constructors G-K* 69d⟩

This code is written to file `pentium-GK.spec`.

87a ⟨*pentium-LQ.spec* 87a⟩≡
⟨*header.spec* 86d⟩
⟨*alphabetical constructors L-Q* 70e⟩
This code is written to file `pentium-LQ.spec`.

87b ⟨*pentium-R.spec* 87b⟩≡
⟨*header.spec* 86d⟩
⟨*alphabetical constructors R* 73c⟩
This code is written to file `pentium-R.spec`.

87c ⟨*pentium-SZ.spec* 87c⟩≡
⟨*header.spec* 86d⟩
⟨*alphabetical constructors S-Z* 73h⟩
This code is written to file `pentium-SZ.spec`.

87d ⟨*pentium-arith.spec* 87d⟩≡
⟨*header.spec* 86d⟩
⟨*arithmetic constructors* 61a⟩
This code is written to file `pentium-arith.spec`.

87e ⟨*pentium-float.spec* 87e⟩≡
⟨*header.spec* 86d⟩
⟨*floating-point constructors* 64h⟩
This code is written to file `pentium-float.spec`.

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.

87f ⟨*pentium-check.spec* 87f⟩≡
⟨*pentium32.spec* 46a⟩
⟨*pentium-linux.spec* 63b⟩
```
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
```
This code is written to file `pentium-check.spec`.

87g ⟨*pentium-checker.s* 87g⟩≡
```
.align 16
```

This code is written to file `pentium-checker.s`.

References

Intel Corporation. 1993.
Architecture and Programming Manual. Vol. 3 of *Pentium Processor User's Manual.*
Mount Prospect, IL.

Joy, William. 1994.
An Introduction to the C shell, Chapter 4.
Sebastopol, CA: O'Reilly.

Kane, Gerry. 1988.
MIPS RISC Architecture.
Englewood Cliffs, NJ: Prentice Hall.

Knuth, Donald E. 1992.
Literate Programming. CSLI Lecture Notes Number 27.
SUCSLI.
Distributed by the University of Chicago Press.

Ramsey, Norman. 1994 (September).
Literate programming simplified.
IEEE Software, 11(5):97–105.

Ramsey, Norman and Mary F. Fernández. 1994 (October).
New Jersey Machine-Code Toolkit reference manual.
Technical Report TR-471-94, Department of Computer Science, Princeton University.
Revised December, 1996.

———. 1995 (January).
The New Jersey Machine-Code Toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

———. 1997 (May).
Specifying representations of machine instructions.
ACM Transactions on Programming Languages and Systems, 19(3):492–524.

Sites, Richard and Richard Witek. 1995.
Alpha AXP Architecture Reference Manual. Second edition.
Digital Press.

SPARC International. 1992.
The SPARC Architecture Manual, Version 8.
Englewood Cliffs, NJ: Prentice Hall.