New Jersey Machine-Code Toolkit
Architecture Specifications

Norman Ramsey Mary Fernandez

October 18, 1994

Preface

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, and debuggers. 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, and Intel 486 instruction sets. Ramsey and Fernandez (1995) present an overview of the toolkit; read it first. The reference manual for the toolkit (Ramsey and Fernandez 1994) gives a complete description of the specification language used here.

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* 17a⟩. The "17a" 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 in order of increasing complexity: the MIPS, followed by the SPARC, and then the 486. The annotations in the MIPS specification describe basic use of the toolkit language. More sophisticated techniques appear in the SPARC and 486 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 **3**
1.1 The MIPS CPU . . . 3
1.1.1 Opcodes . . . 4
1.1.2 Grouping instructions . . . 7
1.1.3 Constructors for simple instructions . . . 8
1.1.4 Branches and relocatable addresses . . . 9
1.1.5 Synthetic instructions and conditional assembly . . . 10
1.2 The floating-point coprocessor . . . 11
1.3 Application-specific specifications for the `mld` linker . . . 13

2 SPARC instruction-set specification **16**
2.1 Fields . . . 16
2.2 Opcodes . . . 17
2.3 Complex operands and typed constructors . . . 22
2.4 Load and store, read and write . . . 23
2.5 Shift, logic, and arithmetic . . . 26
2.6 Branches and call . . . 26
2.7 Floating point . . . 27
2.8 Miscellany . . . 27
2.9 Synthetic instructions . . . 28
2.10 Application-specific specifications for the `mld` linker . . . 29

3 Intel Pentium instruction-set specification **31**
3.1 Opcodes . . . 35
3.2 One-byte opcodes . . . 35
3.3 Two-byte opcodes . . . 38
3.4 Operands and effective addresses . . . 39
3.5 Mod R/M opcodes . . . 41
3.6 Operand-size and address-size prefixes . . . 43
3.7 Floating-point opcodes . . . 44
3.8 Arithmetic instructions . . . 45
3.9 Other instructions (in alphabetical order) . . . 45

4 Synthetic instructions **56**

References **57**

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`.

3a ⟨*mips-core.spec* 3a⟩≡
```
fields of instruction (32) ⟨field specifications 3b⟩
⟨fieldinfo specifications 4b⟩
⟨pattern and constructor specifications 4c⟩
```
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.

3b ⟨*field specifications* 3b⟩≡
```
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
```
This definition is continued in chunks 11b and 12b.
This code is used in chunk 3a.

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 might also one day be used to generate symbolic assemblers and disassemblers from a toolkit specification. On the MIPS, we use names of the form `r`*n*, except for the stack pointer (`sp`).

4a ⟨*properties of integer-register fields* 4a⟩≡

```
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 ]
```

This code is used in chunk 4b.

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

4b ⟨`fieldinfo` *specifications* 4b⟩≡

```
fieldinfo [ rs rt rd base ] is [ ⟨properties of integer-register fields 4a⟩ ]
```

This definition is continued in chunk 11.
This code is used in chunk 3a.

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.

4c ⟨*pattern and constructor specifications* 4c⟩≡

```
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 definition is continued in chunks 6–10, 12, and 13.
This code is used in chunk 3a.

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.

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.

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:

6a ⟨*pattern and constructor specifications* 4c⟩+≡

```
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.

6b ⟨*pattern and constructor specifications* 4c⟩+≡

```
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.

7 ⟨*pattern and constructor specifications* 4c⟩+≡

```
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
  mfthilo is mfhi | mflo | 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]

8 ⟨*pattern and constructor specifications* 4c⟩+≡

```
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
  mfthilo rd
  syscall
  break  breakcode
  copls  rt, offset!(base)
  jr     rs
  jalr   rs, rd
```

[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.

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.

These constructor specifications include the assembly-language syntax associated with the instructions being specified. For example, most operand 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 hope one day to use the assembly syntax to produce an assembler and disassembler from this specification.

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.

9 ⟨*pattern and constructor specifications* 4c⟩+≡
⟨*placeholder definition* 10a⟩
```
constructors
  branch1 rs, reloc
     { reloc = $pc + 4 + 4 * offset! } is branch1 & rs & offset
  branch2 rs, rt, reloc
     { reloc = $pc + 4 + 4 * offset! } is branch2 & rs & rt & offset
```

`$pc` denotes the location at which the constructor's tokens begin. The branch equations capture the semantics of the MIPS branch instructions; the offset is sign-extended, shifted left 2 bits, and added to the address of the delay slot, i.e., `$pc + 4`.

Constructor operands whose name begin with `reloc` are *relocatable addresses*, like the branch targets above. `$pc` is also a relocatable address. 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.

10a ⟨*placeholder definition* 10a⟩≡
```
placeholder for instruction is break(99)
```
This code is used in chunk 9.

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.

10b ⟨*pattern and constructor specifications* 4c⟩+≡
```
constructors
  jump reloc { reloc[28:31] = $pc[28:31],
               reloc[2:27]  = target,
               reloc[0:1]   = 0 }
         is jump & target
```
The high bits of the target address must match the PC, 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.

10c ⟨*mips-synth.spec* 10c⟩≡
```
constructors
  nop         is  sll (r0, r0, 0)
  mov rd, rs  is  addu(rd, rs, r0)
  b reloc     is  beq (r0, r0, reloc)
```
This definition is continued in chunk 11a.
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.

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.

11a ⟨*mips-synth.spec* 10c⟩+≡

```
constructors
  li rt, imm  when { imm[16:31]! = imm[15]!, offset = imm[0:15]! }
                   is addiu(rt, r0, offset)
              when {  u = imm[16:31], imm[0:15] = 0 }
                   is lui(rt, u)
              when {  u = imm[16:31] + imm[15], l = imm[0:15]! }
                   is lui(rt, u); addiu(rt, rt, l)
```

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.

11b ⟨*field specifications* 3b⟩+≡

```
ft 16:20 fs 11:15 fd 6:10 format 21:24 bit25 25:25
```

11c ⟨`fieldinfo` *specifications* 4b⟩+≡

```
fieldinfo [ fs ft fd ] is [ ⟨properties of floating-point-register fields 11d⟩]
```

11d ⟨*properties of floating-point-register fields* 11d⟩≡

```
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 ]
```

This code is used in chunk 11c.

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.

11e ⟨`fieldinfo` *specifications* 4b⟩+≡

```
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.

12a ⟨*pattern and constructor specifications* 4c⟩+≡

```
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.

12b ⟨*field specifications* 3b⟩+≡

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

12c ⟨*pattern and constructor specifications* 4c⟩+≡

```
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 arbitrarily chooses a value for `cop1code`; toolkit-generated decoding code recognizes either variant.

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

13a ⟨*pattern and constructor specifications* 4c⟩+≡

```
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
  bc1ft   is bc1f | bc1t
  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.

13b ⟨*pattern and constructor specifications* 4c⟩+≡

```
constructors
  arith3.^format fd, fs, ft
  arith2.^format fd, fs
  movec1         rt, fs
  c.cond^"."^format  fs, ft
  convert^"."^format fd, fs
  bc1ft reloc { reloc = $pc + 4 + 4 * offset! } is bc1ft & offset
```

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.

14a ⟨*mld-mips.spec* 14a⟩≡

```
constructors
  break7ifzero rt
    when { rt = 0 }       is break(7)
    when { L = $pc + 12 } is bne(rt, r0, L); nop(); break(7)
```

This definition is continued in chunks 14 and 15.
This code is written to file `mld-mips.spec`.

This example illustrates a minor embarrassment: there's no way in the toolkit's specification language to bind a relocatable address to a location in the token stream. We really would prefer to write something like:

```
break7ifzero rt is bne(rt, r0, L); nop(); break(7); L: not legal
```

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.

14b ⟨*mld-mips.spec* 14a⟩+≡

```
constructors
  break7ifoverflow rs, rt
       { L = $pc + 24 } is
    addiu(r1, r0, -1);      # $pc
    bne(rt, r1, L);         # $pc+4
    lui(r1, 0x8000);        # $pc+8
    bne(rs, r1, L);         # $pc+12
    nop();                  # $pc+16
    break(7)                # $pc+20
```

The comments illustrate the tedium of counting offsets.

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

15a ⟨*mld-mips.spec* 14a⟩+≡

```
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:

15b ⟨*mips-regs.spec* 15b⟩≡

```
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's 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:

15c ⟨*mld-mips.spec* 14a⟩+≡

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

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.

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 use the extended pattern declaration

p `is any of [` p_0 p_1 ... 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 2.10.

2.1 Fields

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

16a ⟨*sparc-core.spec* 16a⟩≡
```
fields of instruction (32) ⟨field specifications 16b⟩
⟨fieldinfo specifications 24d⟩
⟨pattern and constructor specifications 17a⟩
```
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.

16b ⟨*field specifications* 16b⟩≡
```
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
```
This definition is continued in chunks 23c and 25c.
This code is used in chunk 16a.

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.

17a ⟨*pattern and constructor specifications* 17a⟩≡

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

This definition is continued in chunks 17–27.
This code is used in chunk 16a.

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.

17b ⟨*pattern and constructor specifications* 17a⟩+≡

```
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 $(0, 16, 32, 48, 1, \ldots, 63)$.

18a ⟨*pattern and constructor specifications* 17a⟩+≡

```
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.

18b ⟨*pattern and constructor specifications* 17a⟩+≡

```
patterns
  WRASR          is WRxxx & rd != 0
  WRY            is WRxxx & rd = 0
  RDASR          is RDxxx & rs1 != 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:

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

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.

19 ⟨*pattern and constructor specifications* 17a⟩+≡

```
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. This division makes it easy to use the names `float2` and `float3` to refer to the groups, without having to re-enumerate the members of each group.

20a ⟨*pattern and constructor specifications* 17a⟩+≡

```
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 ]

  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 ]
```

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 486.

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

20b ⟨*pattern and constructor specifications* 17a⟩+≡

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

21a ⟨*pattern and constructor specifications* 17a⟩+≡

```
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.

21b ⟨*sparc-synth.spec* 21b⟩≡

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

This definition is continued in chunks 28 and 29a.
This code is written to file `sparc-synth.spec`.

2.3 Complex operands and typed constructors

Unlike the MIPS, the SPARC has instructions whose operands are not simple integers or fields. For example, the load and store instructions take an operand that computes an address, and the integer-arithmetic instructions take one that may be a register or an immediate value. 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 an address operand are

22a ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  dispA     rs1 + simm13! : address_  is  i = 1 & rs1 & simm13
  absoluteA simm13!       : address_  is  i = 1 & rs1 = 0 & simm13
  indexA    rs1 + rs2     : address_  is  i = 0 & rs1 & rs2
  indirectA rs1           : address_  is  i = 0 & rs2 = 0 & rs1
```

The type `address_`[2] is *defined* by these declarations. Used elsewhere, an operand of type `address_` always denotes a pattern created by the application of one of these four 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 `address_`. Such constructors, notably the load and store constructors, appear below.

The type of "register or immediate" operand is specified similarly; such an operand is created by an immediate-mode or a register-mode constructor:

22b ⟨*pattern and constructor specifications* 17a⟩+≡

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

[2]We can't call it `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.

23a ⟨*pattern and constructor specifications* 17a⟩+≡

```
patterns
  loadg  is LDSB  | LDSH  | LDUB  | LDUH  | LD  | LDD  | LDSTUB  | SWAP_
  loada  is LDSBA | LDSHA | LDUBA | LDUHA | LDA | LDDA | LDSTUBA | SWAPA
  loadf  is LDF   | LDDF
  loadc  is LDC   | LDDC
  storeg is STB   | STH  | ST  | STD
  storea is STBA  | STHA | STA | STDA
  storef is STF   | STDF
  storec is STC   | STDC
```

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.

23b ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  loadg  [address_], rd
  loada  [address_]asi, rd
  loadf  [address_], fd
  loadc  [address_], cd
  storeg rd, [address_]
  storea rd, [address_]asi
  storef fd, [address_]
  storec cd, [address_]
```

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

23c ⟨*field specifications* 16b⟩+≡

```
fd 25:29 cd 25:29
```

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.

24a ⟨*properties of integer-register fields* 24a⟩≡
```
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" ]
```
This code is used in chunk 24d.

24b ⟨*properties of floating-point destination register* 24b⟩≡
```
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" ]
```
This code is used in chunk 24d.

24c ⟨*properties of coprocessor destination register* 24c⟩≡
```
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" ]
```
This code is used in chunk 24d.

24d ⟨`fieldinfo` *specifications* 24d⟩≡
```
fieldinfo
[ rd rs1 rs2 ] is [ ⟨properties of integer-register fields 24a⟩ ]
            fd is [ ⟨properties of floating-point destination register 24b⟩ ]
            cd is [ ⟨properties of coprocessor destination register 24c⟩ ]
```
This definition is continued in chunk 26c.
This code is used in chunk 16a.

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 may one day help us generate a disassembler, but more importantly, it makes the specification easier to read and understand.

24e ⟨*pattern and constructor specifications* 17a⟩+≡
```
constructors
  LDFSR  [address_], "%fsr"
  LDCSR  [address_], "%csr"
  STFSR  "%fsr", [address_]
  STCSR  "%csr", [address_]
  STDFQ  "%fq",  [address_]
  STDCQ  "%csr", [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.

25a ⟨*pattern and constructor specifications* 17a⟩+≡

```
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.

25b ⟨*pattern and constructor specifications* 17a⟩+≡

```
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.

25c ⟨*field specifications* 16b⟩+≡

```
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.

26a ⟨*pattern and constructor specifications* 17a⟩+≡

```
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.

26b ⟨*pattern and constructor specifications* 17a⟩+≡

```
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:

26c ⟨`fieldinfo` *specifications* 24d⟩+≡

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

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

26d ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  branch^a reloc  { reloc = $pc + 4 * disp22! } is 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.

27a ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  call__  reloc   { reloc = $pc + 4 * disp30! } is 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.

27b ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  float2   rs2, rd
  float3   rs1, rs2, rd
  fcompare rs1, rs2
```

2.8 Miscellany

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

27c ⟨*pattern and constructor specifications* 17a⟩+≡

```
constructors
  NOP
  FLUSH address_
  JMPL  address_, rd
  RETT  address_
  trap  address_
  UNIMP imm22
  sethi "%hi("val")", rd  { imm22 = val[10:31] } is SETHI & rd & imm22
```

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 names with trailing underscores (e.g., `call_`) for other variants.

28 ⟨*sparc-synth.spec* 21b⟩+≡

```
constructors
  cmp rs1, reg_or_imm   is SUBcc(rs1, reg_or_imm, "%g0")
  jmp address_          is JMPL (address_, "%g0")
  call_ address_        is JMPL (address_, "%o7")
  tst  rs1              is SUBcc(rs1, rmode("%g0"), "%g0")
  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)
  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)
  clr  rd               is OR  ("%g0", rmode("%g0"), rd)
  clr_ address_         is ST  ("%g0", address_)
  clrb address_         is STB ("%g0", address_)
  clrh address_         is STH ("%g0", address_)
  mov  reg_or_imm, rd   is OR  ("%g0", reg_or_imm, rd)
  mov_ rs2, rd          is OR  ("%g0", rmode(rs2),   rd)
```

Note that some of the constructors applied in the output patterns require `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 16 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.

29a ⟨*sparc-synth.spec* 21b⟩+≡

```
constructors
  set val, rd
    when { val[0:9] = 0 }     is  sethi(val, rd)
    when { val = val[0:15]! } is  OR("%g0", imode(val), rd)
    when { lo = val[0:9] }    is  sethi(val, rd); OR(rd, imode(lo), rd)
```

2.10 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_`.

29b ⟨*mld-sparc.spec* 29b⟩≡

```
constructors
  setr reloc, rd  is  set(reloc, rd)
  ldr reloc, rd
      { lo = reloc[0:9] }
    is sethi(reloc, "%o2"); ld(dispA("%o2", lo), rd)
  str rd, reloc
      { lo = reloc[0:9] }
    is sethi(reloc, "%o2"); st(rd, dispA("%o2", lo))
```

This definition is continued in chunks 29 and 30.
This code is written to file `mld-sparc.spec`.

It's slightly embarrasing that we have to create a set of equations in order to get `lo = reloc[0:9]`, but the toolkit doesn't accept expressions like `reloc[0:9]` as operands. There's no good reason for it; it's just another of the warts we hope to remove some day.

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

29c ⟨*mld-sparc.spec* 29b⟩+≡

```
constructors
  branch.synonyms^a reloc  { reloc = $pc + 4 * disp22! }  is
        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 `call_`, 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`.

30a ⟨*mld-sparc.spec* 29b⟩+≡

```
constructors
  relocfun reloc    : Function  is  call__(reloc)
  addrfun  address_ : Function  is  call_(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:

30b ⟨*mld-sparc.spec* 29b⟩+≡

```
constructors
  _call Function is Function
```

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

30c ⟨*mld-sparc.spec* 29b⟩+≡

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

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

30d ⟨*sparc-regs.spec* 30d⟩≡

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

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

3 Intel Pentium instruction-set specification

Although we have referred elsewhere to the "486," this specification describes the Intel Pentium (Intel 1993). There is little difference at this level; 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.[3]

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 3.6. 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.

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

[3] 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.

One-Byte Opcode Map (page 0)

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES	POP ES
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	ADC						PUSH SS	POP SS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	AND						SEG =ES	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	XOR						SEG =SS	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
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 PUSHAD	POPA POPAD	BOUND Gv,Ma	ARPL Ew,Gw	SEG =FS	SEG =GS	Operand Size	Address 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 Jb	LOOPE Jb	LOOP Jb	JCXZ/JEC Jb	IN		OUT	
					AL,Ib	AX,Ib	Ib,AL	Ib,eAX
F	LOCK	*	REPNE	REP REPE	HLT	CMC	Unary Grp3	
							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)

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 3.6 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:

34 ⟨*i486-core.spec* 34⟩≡
```
⟨field specs 35a⟩
patterns ⟨patterns for integer opcodes 35c⟩
⟨pattern specs for other patterns 42⟩
⟨prefix assignments 43a⟩
⟨placeholders 35b⟩
⟨constructors for displacements 46b⟩
⟨constructors for effective addresses 40a⟩
⟨arithmetic constructors 45a⟩
constructors
⟨alphabetical constructors 45b⟩
```
This code is written to file `i486-core.spec`.

3.1 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.

3.2 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.

35a ⟨*field specs* 35a⟩≡

```
fields of opcode (8) row 4:7 col 0:2 page 3:3
                        ⟨more fields of opcode 36c⟩
```

This definition is continued in chunks 36d, 37e, 39c, and 40b.
This code is used in chunk 34.

35b ⟨*placeholders* 35b⟩≡

```
placeholder for opcode is HLT
```

This definition is continued in chunks 39d and 40c.
This code is used in chunk 34.

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:

35c ⟨*patterns for integer opcodes* 35c⟩≡

```
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}
```

This definition is continued in chunks 36–39.
This code is used in chunk 34.

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

36a ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

36b ⟨*patterns for integer opcodes* 35c⟩+≡
```
regops is any of [ INC  DEC
                   PUSH POP ], which is row = [4 5] & page = [0 1]
```

36c ⟨*more fields of* `opcode` 36c⟩≡
```
r32 0:2
```
This definition is continued in chunk 37d.
This code is used in chunk 35a.

36d ⟨*field specs* 35a⟩+≡
```
fieldinfo r32 is [names [ eAX eCX eDX eBX eSP eBP eSI eDI ]]
```

Row 6:

36e ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

36f ⟨*patterns for integer opcodes* 35c⟩+≡
```
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.

36g ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

37a ⟨*patterns for integer opcodes* 35c⟩+≡
```
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.

37b ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

37c ⟨*patterns for integer opcodes* 35c⟩+≡
```
MOVib is row = 11 & page = 0
MOViv is row = 11 & page = 1
```

37d ⟨*more fields of* opcode 36c⟩+≡
```
r8 0:2
```

37e ⟨*field specs* 35a⟩+≡
```
fieldinfo r8 is [names [ AL CL DL BL AH CH DH BH ]]
```

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

37f ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

37g ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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}
```

3.3 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:

38a ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

38b ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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:

38c ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

38d ⟨*patterns for integer opcodes* 35c⟩+≡
```
Jv   is esc2; row = 8
SETb is esc2; row = 9
```

Rows 10 and 11 are more madness:

38e ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

38f ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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.

38g ⟨*patterns for integer opcodes* 35c⟩+≡
```
[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:

38h ⟨*patterns for integer opcodes* 35c⟩+≡
```
[ 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:

39a ⟨*patterns for integer opcodes* 35c⟩+≡

```
[ 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:

39b ⟨*patterns for integer opcodes* 35c⟩+≡

```
BSWAP is esc2; row = 12 & page = 1
```

3.4 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.

39c ⟨*field specs* 35a⟩+≡

```
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
```

39d ⟨*placeholders* 35b⟩+≡

```
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.

40a ⟨*constructors for effective addresses* 40a⟩≡

```
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
  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
```

This code is used in chunk 34.

Immediate operands occupy whole tokens and have their own classes.

40b ⟨*field specs* 35a⟩+≡

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

40c ⟨*placeholders* 35b⟩+≡

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

3.5 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.[4] In the case at hand, every `Eaddr` begins with a Mod R/M token, so we can always write

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

[4]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 3.8.

42 ⟨*pattern specs for other patterns* 42⟩≡

```
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
  rot       is any of [ ROL ROR RCL RCR SHLSAL SHR _ SAR],
                        which is (bshifts | vshifts); 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 ...
```

This definition is continued in chunk 44.
This code is used in chunk 34.

3.6 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`.

43a ⟨*prefix assignments* 43a⟩≡

```
patterns ow is OpPrefix
         od is epsilon
         ov is ow | od
```

This definition is continued in chunk 43b.
This code is used in chunk 34.

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

43b ⟨*prefix assignments* 43a⟩+≡

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

3.7 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`.

44a ⟨*pattern specs for other patterns* 42⟩+≡

```
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 = 1 & r_m = [0]
  [ FCHS FABS _ _ FTST FXAM _ _  ] is D9; mod = 3 & reg_opcode = 2 & r_m = {0 to 7}
  [ F2XM1 FYL2X FPTAN FPATAN FXTRACT FPREM1 FDECSTP FINCSTP ]
                                    is D9; mod = 3 & reg_opcode = 3 & r_m = {0 to 7}
  FXCH                              is D9; mod = 3 & reg_opcode = 4
  Fconstants is any of [ FLD1 FLDL2T FLDL2E FLDPI FLDLG2 FLDLN2 FLDZ _ ], which
                                    is D9; mod = 3 & reg_opcode = 6 & 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 = 6 & 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 = 2 & r_m = [2 3]
  [ FRSTOR _ FSAVE FSTSW ]           is reg_opcode = {4 to 7} ...
  [ FFREE FST.st FUCOM _ _ FSTP.st 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 = 5 & r_m = 1
  FSTSW.AX  is DF; mod = 3 & reg_opcode = 3 & r_m = 0
```

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

44b ⟨*pattern specs for other patterns* 42⟩+≡

```
patterns
  .STi      is DD; mod = 3
  Fstack is any of [ .ST.STi .STi.St P.STi.ST ], which is [ D8 DC DE ]; mod = 3
  Fmem   is any of [.R32 .R64 .I32 .I16], which is [D8 DC DA DE]
  FlsI   is any of [.lsI16 .lsI32], which is [DF DB]
  FlsR   is any of [.lsR32 .lsR64], which is [D9 DD]
```

3.8 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.

45a ⟨*arithmetic constructors* 45a⟩≡

```
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, reg   is      arith & Eb.Gb; Eaddr & reg_opcode = reg ...
  arith^"mr"^ov  Eaddr, reg   is  ov; arith & Ev.Gv; Eaddr & reg_opcode = reg ...
  arith^"rmb"    reg, Eaddr   is      arith & Gb.Eb; Eaddr & reg_opcode = reg ...
  arith^"rm"^ov  reg, Eaddr   is  ov; arith & Gv.Ev; Eaddr & reg_opcode = reg ...
```

This code is used in chunk 34.

3.9 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).

45b ⟨*alphabetical constructors* 45b⟩≡

```
AAA
AAD is AAD; i8 = 10
AAM is AAM; i8 = 10
AAS
# ADC, ADD, AND are in arith group
ARPL     Eaddr, reg  is  ARPL;      Eaddr & reg_opcode = reg ...
```

This definition is continued in chunks 46–56.
This code is used in chunk 34.

46a ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

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

46b ⟨*constructors for displacements* 46b⟩≡

```
constructors
  rel8  reloc : Rel8  { reloc = $pc + 1 + i8!  } is i8
  rel16 reloc : Rel16 { reloc = $pc + 2 + i16! } is i16
  rel32 reloc : Rel32 { reloc = $pc + 4 + i32! } is i32
```

This code is used in chunk 34.

47a ⟨*alphabetical constructors* 45b⟩+≡

```
CALL.Jv^ow reloc  is  ow; CALL.Jv; rel16(reloc)
CALL.Jv^od reloc  is  od; CALL.Jv; rel32(reloc)
CALL.Ev^ov Eaddr  is  ov; (grp5; Eaddr) & CALL.Ev
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.Ep^ov Eaddr  is  ov; (grp5; Eaddr) & CALL.Ep
CBW  is ow; CBW
CWDE is od; CBW
CLC
CLD
CLI
CLTS
CMC
# 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
```

47b ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

47c ⟨*alphabetical constructors* 45b⟩+≡

```
ENTER   i16, i8   is  ENTER; i16; i8
```

47d ⟨*alphabetical constructors* 45b⟩+≡

```
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`.

48a ⟨*alphabetical constructors* 45b⟩+≡
```
FABS
FADD^Fmem   Mem  is  Fmem; Mem & FADD ...
FADD^Fstack idx  is  Fstack & ... (FADD & r_m = idx)
```

48b ⟨*alphabetical constructors* 45b⟩+≡
```
FBLD        Mem  is  DF; Mem & FBLD
FBSTP       Mem  is  DF; Mem & FBSTP
```

48c ⟨*alphabetical constructors* 45b⟩+≡
```
FCHS
```

48d ⟨*synthetics with* `WAIT` 48d⟩≡
```
FCLEX             is  WAIT; FCLEX
```
This definition is continued in chunks 49–51.
This code is used in chunk 56b.

48e ⟨*alphabetical constructors* 45b⟩+≡
```
FNCLEX            is  FCLEX
   patterns FCOMs is FCOM | FCOMP
   constructors
FCOMs^Fmem     Mem  is  Fmem; Mem & FCOMs ...           # includes FICOM, FICOMP
FCOMs^.ST.STi  idx  is  .ST.STi & ... (FCOMs & r_m = idx)
FCOMPP
FCOS
```

48f ⟨*alphabetical constructors* 45b⟩+≡
```
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)
```

48g ⟨*alphabetical constructors* 45b⟩+≡
```
FFREE         idx  is  DD; FFREE & r_m = idx
```

49a ⟨*alphabetical constructors* 45b⟩+≡

```
FICOM16   Mem is FCOM.I16(Mem)
FICOM32   Mem is FCOM.I32(Mem)
FICOMP16  Mem is FCOMP.I16(Mem)
FICOMP32  Mem is FCOMP.I32(Mem)
FILD^FlsI Mem is FlsI; Mem & FILD
FILD64    Mem is DF;  Mem & FLD.ext ...
FINIT
   patterns FISTs is FIST | FISTP
   constructors
FISTs^FlsI  Mem  is  FlsI; Mem & FISTs
FISTP64     Mem  is  DF;   Mem & FSTP.ext
```

49b ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

49c ⟨*alphabetical constructors* 45b⟩+≡

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

49d ⟨*alphabetical constructors* 45b⟩+≡

```
FNOP
```

49e ⟨*alphabetical constructors* 45b⟩+≡

```
FPATAN
FPREM
FPREM1
FPTAN
```

49f ⟨*alphabetical constructors* 45b⟩+≡

```
FRNDINT
FRSTOR Mem  is DD; Mem & FRSTOR
```

49g ⟨*alphabetical constructors* 45b⟩+≡

```
FNSAVE Mem  is DD; Mem & FSAVE
```

49h ⟨*synthetics with* `WAIT` 48d⟩+≡

```
FSAVE  Mem  is WAIT; FNSAVE(Mem)
```

50a ⟨*alphabetical constructors* 45b⟩+≡
```
FSCALE
FSIN
FSINCOS
FSQRT
   patterns
FSTs is FST | FSTP
FSTs.st is FST.st | FSTP.st
   constructors
FSTsˆFlsR     Mem  is  FlsR; Mem & FSTs
FSTP80        Mem  is  DB;   Mem & FSTP.ext
FSTs.stˆ.STi  idx  is  .STi & ... (FSTs.st & r_m = idx)
FSTCW         Mem  is  D9;   Mem & FSTCW
```

50b ⟨*synthetics with* WAIT 48d⟩+≡
```
FNSTCW        Mem  is  WAIT; FSTCW(Mem)
```

50c ⟨*alphabetical constructors* 45b⟩+≡
```
FSTENV        Mem  is  D9;   Mem & FSTENV
```

50d ⟨*synthetics with* WAIT 48d⟩+≡
```
FNSTENV       Mem  is  WAIT; FSTENV(Mem)
```

50e ⟨*alphabetical constructors* 45b⟩+≡
```
FSTSW         Mem  is  DD;   Mem & FSTSW
FSTSW.AX
```

50f ⟨*synthetics with* WAIT 48d⟩+≡
```
FNSTSW        Mem  is  WAIT; FSTSW(Mem)
FNSTSW.AX          is  WAIT; FSTSW.AX()
```

50g ⟨*alphabetical constructors* 45b⟩+≡
```
FSUBˆ Fmem    Mem  is  Fmem; Mem & FSUB ...
FSUBˆ Fstack  idx  is  Fstack & ... (FSUB & r_m = idx)
FSUBRˆFmem    Mem  is  Fmem; Mem & FSUBR ...
FSUBRˆFstack  idx  is  Fstack & ... (FSUBR & r_m = idx)
```

50h ⟨*alphabetical constructors* 45b⟩+≡
```
FTST
```

50i ⟨*alphabetical constructors* 45b⟩+≡
```
    patterns FUCOMs is FUCOM | FUCOMP
    constructors
FUCOMs  idx   is  DD; FUCOMs & r_m = idx
FUCOMPP
```

51a ⟨*synthetics with* WAIT 48d⟩+≡

```
FWAIT is WAIT
```

51b ⟨*alphabetical constructors* 45b⟩+≡

```
FXAM
FXCH idx  is  FXCH & ... r_m = idx
FXTRACT
```

51c ⟨*alphabetical constructors* 45b⟩+≡

```
FYL2X
FYL2XP1
```

51d ⟨*alphabetical constructors* 45b⟩+≡

```
HLT
```

51e ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

52a ⟨*alphabetical constructors* 45b⟩+≡

```
Jb^cond    reloc  is      Jb & cond;     rel8(reloc)
Jv^cond^ow reloc  is  ow; Jv & ... cond; rel16(reloc)
Jv^cond^od reloc  is  od; Jv & ... cond; rel32(reloc)
JMP.Jb     reloc   is      JMP.Jb; rel8(reloc)
JMP.Jv^ow  reloc   is  ow; JMP.Jv; rel16(reloc)
JMP.Jv^od  reloc   is  od; JMP.Jv; rel32(reloc)
JMP.Ap^ow  CS, IP  is  ow; JMP.Ap; i16 = CS; i16 = IP
JMP.Ap^od  CS, IP  is  od; JMP.Ap; i16 = CS; i32 = IP
JMP.Ev^ov Eaddr    is  ov; (grp5; Eaddr) & JMP.Ev
JMP.Ep^ov Eaddr    is  ov; (grp5; Eaddr) & JMP.Ep
```

52b ⟨*alphabetical constructors* 45b⟩+≡

```
LAHF
LAR^ov  reg, Eaddr  is  ov; LAR; Eaddr & reg_opcode = reg ...
   patterns lfp is LDS | LES | LFS | LGS | LSS
   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
    patterns LOOPs is LOOP | LOOPE | LOOPNE
    constructors
LOOPs  reloc  is  LOOPs; rel8(reloc)
LSL^ov  reg, Eaddr  is  ov; LSL; Eaddr & reg_opcode = reg ...
LTR  Eaddr  is  (grp6; Eaddr) & LTR
```

53a ⟨*alphabetical constructors* 45b⟩+≡

```
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    Eaddr, reg   is  ow; MOV.Ew.Sw;   Eaddr & reg_opcode = reg ...
MOV.Sw.Ew    Eaddr, reg   is      MOV.Sw.Ew;   Eaddr & reg_opcode = reg ...
# assume 32-bit address mode
MOV.AL.Ob      offset     is      MOV.AL.Ob;  i32 = offset
MOV.eAX.Ov^ov  offset     is  ov; MOV.eAX.Ov; i32 = offset
MOV.Ob.AL      offset     is      MOV.Ob.AL;  i32 = offset
MOV.Ov.eAX^ov  offset     is  ov; MOV.Ov.eAX; i32 = offset
MOVib          r8, i8     is      MOVib & r8; i8
MOViw          r32, i16   is  ow; MOViv & r32; i16
MOVid          r32, i32   is  od; MOViv & r32; i32
MOV.Eb.Ib      Eaddr, i8  is      MOV.Eb.Ib; Eaddr & reg_opcode = 0 ...; i8
MOV.Eb.Iv^ow   Eaddr, i16 is  ow; MOV.Ev.Iv; Eaddr & reg_opcode = 0 ...; i16
MOV.Eb.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    reg, Eaddr  is ov; MOVSX.Gv.Eb; Eaddr & reg_opcode = reg ...
MOVSX.Gv.Ew       reg, Eaddr  is     MOVSX.Gv.Ew; Eaddr & reg_opcode = reg ...
MOVZX.Gv.Eb^ov    reg, Eaddr  is ov; MOVZX.Gv.Eb; Eaddr & reg_opcode = reg ...
MOVZX.Gv.Ew       reg, Eaddr  is     MOVZX.Gv.Ew; Eaddr & reg_opcode = reg ...
MUL.AL      Eaddr  is      (grp3.Eb; Eaddr) & MUL.AL.eAX
MUL.AX^ov   Eaddr  is  ov; (grp3.Ev; Eaddr) & MUL.AL.eAX
```

53b ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

54a ⟨*alphabetical constructors* 45b⟩+≡

```
# 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
```

54b ⟨*alphabetical constructors* 45b⟩+≡

```
POP.Ev^ov   Mem   is  ov; POP.Ev; Mem & reg_opcode = 0 ...
POP^ov      r32   is  ov; POP & r32
   patterns POPs is POP.ES | POP.SS | POP.DS | POP.FS | POP.GS
            POPv is POPA | POPF
   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
   patterns PUSHs is PUSH.CS | PUSH.SS | PUSH.DS | PUSH.ES | PUSH.FS | PUSH.GS
            PUSHv is PUSHA | PUSHF
   constructors
PUSHs
PUSHv^ov  is  ov; PUSHv
```

54c ⟨*alphabetical constructors* 45b⟩+≡

```
# 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
RDMSR
REP
REPNE
RET
RET.far
RET.Iw      i16  is RET.Iw; i16
RET.far.Iw  i16  is RET.far.Iw; i16
RSM
```

55a ⟨*alphabetical constructors* 45b⟩+≡

```
SAHF
# SAL SAR SHL SR above with rot
# SBB is in the arith group
SCASB
SCASv^ov  is  ov; SCASv
SETb^cond  Eaddr  is  SETb & ... cond; Eaddr
SGDT  Mem  is (grp7; Mem) & SGDT
SIDT  Mem  is (grp7; Mem) & SIDT
  patterns shdIb is SHRD.Ib | SHLD.Ib
           shdCL is SHRD.CL | SHLD.CL
  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  Eaddr  is  (grp6; Eaddr) & STR
# SUB is in the arith group
```

55b ⟨*alphabetical constructors* 45b⟩+≡

```
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 ...
```

55c ⟨*alphabetical constructors* 45b⟩+≡

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

55d ⟨*alphabetical constructors* 45b⟩+≡

```
WAIT
WBINVD
WRMSR
```

56a ⟨*alphabetical constructors* 45b⟩+≡

```
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
```

4 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.

56b ⟨*i486-synth.spec* 56b⟩≡

```
constructors
  ⟨synthetics with WAIT 48d⟩
```

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

References

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

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

Knuth, Donald E. 1992.
Literate Programming.
Stanford, California: Stanford University.

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

———. 1995 (January).
The New Jersey machine-code toolkit.
In *Proceedings of the Winter 1995 USENIX Conference*, New Orleans, LA.
To appear. Preliminary version available as Technical Report TR-469-94, Department of Computer Science, Princeton University.

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

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