Specifying Instructions' Semantics Using CSDL (Preliminary Report)

Norman Ramsey and Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

November 24, 1997

Abstract

The Zephyr project is part of an effort to build a National Compiler Infrastructure, which will support research in compiling techniques and high-performance computing. Compilers work with source code, abstract syntax, intermediate forms, and machine instructions. By using high-level descriptions of the representations and semantics of these forms, we expect to be able to create compiler components that will be usable with different source languages, front ends, and target machines.

To help deal with multiple machines, we are developing a family of Computer Systems Description Languages (CSDL) to describe properties that are relevant to the construction of compilers and other systems software. The languages describe properties of a machine's instructions or its mutable state, or both. Of particular interest is the description of the semantics of instructions, i.e., their effects on the state of the machine. This report describes our preliminary design of λ-RTL, a CSDL language for specifying instructions' semantics.

We describe the effects of instructions using register transfer lists (RTLs). A register transfer list is a collection of assignments to locations, which represent registers, memory, and all other mutable state. We prescribe a form of RTLs that makes it explicit how to compute the values assigned and on what state the computation depends. The form also makes byte order explicit and provides for instructions whose effects may be undefined.

Because our form of RTLs contains so much information, it is convenient for use by tools, but it would be tedious to write RTLs by hand. λ-RTL, which is based on the λ-calculus and on register transfer lists, is a metalanguage designed to make it easier for people to write RTLs and to associate them with machine instructions. It enables us to omit substantial information from hand-written specifications; the λ-RTL translator infers the missing information and puts the resulting RTLs into canonical form. λ-RTL also provides a "grouping" construct designed to help specify large groups of similar instructions.

We are still designing λ-RTL. This report presents a short overview of λ-RTL, followed by examples. The examples include definitions of basic operators, which we believe will be useful for describing a wide variety of machines, as well as excerpts from descriptions of the SPARC and Pentium. We have chosen excerpts that illustrate features which are characteristic of these particular machines; for example, we show a model of SPARC register windows, and we show how λ-RTL can help manage the complexity of the Pentium instruction set.

Both the machine descriptions and λ-RTL itself are under development, so this report is a snapshot of a work in progress. We issue it now to solicit feedback both on our overall approach and on the details of λ-RTL. Please send feedback by electronic mail to `zephyr-investigators@virginia.edu`.

Contents

Chapter 1

Overview of CSDL

Machine descriptions for machine-level tools

Special-purpose and general-purpose computers are designed at a rapid pace, and software tools and technology don't always keep up. The tools we need include not only the compilers, assemblers, linkers, and debuggers familiar to all programmers, but also less familiar tools, like profilers, tracers, test-coverage analyzers, and general code-modification tools. Most of these tools must work with machine instructions.

Machine-dependent detail makes it hard to build machine-level tools. For many years, compilers have used machine descriptions to capture such detail. Machine descriptions isolate target-specific information so that it can easily be examined and changed. Despite their successful use in compilers, machine descriptions are seldom used to build other systems software. Descriptions used in compilers are hard to reuse because they typically combine information about the target machine with information about the compiler. For example, "machine descriptions" written using tools like BEG (Emmelmann, Schröer, and Landwehr 1989) and BURG (Fraser, Henry, and Proebsting 1992) are actually descriptions of code generators, and they depend not only on the target machine but also on a particular intermediate language. In extreme cases (e.g., `gcc`'s `md` files), the description formalism itself depends on the compiler.

We believe we can simplify construction of compilers and other machine-level tools by developing description techniques that separate machine properties from compiler concerns. We expect this capability to be useful in a National Compiler Infrastructure because it will lead to a back-end infrastructure that should be usable not only with many different target machines, but also with different source languages, front ends, and intermediate languages.

Some existing languages for machine description, like VHDL (Lipsett, Schaefer, and Ussery 1993) and Verilog (Thomas and Moorby 1995) do describe only properties of the machine, but they are at too low a level, describing implementations as much as architectures. These description languages require too much detail that is not needed to build systems software.

We are designing a family of Computer Systems Description Languages (CSDL) to support a variety of machine-level tools while remaining independent of any one in particular. We have several goals for CSDL:

- Descriptions should be composed from simple components. Each component should describe, as much as possible, a single aspect of the target machine. Such aspects might include calling conventions, representations of instructions, pipeline implementations, memory hierarchy, or other properties. It should not always be necessary to describe aspects completely. For example, most compilers use only a subset of the instructions available on a particular machine, and a compiler writer should be required to describe only that subset.

- An application writer should be able to derive useful tools not only when the components describing individual aspects are incomplete, but also when not all aspects are described. An application writer should not have to describe aspects of the machine unless the information is needed to build his application. For example, an application writer working entirely at the assembly-language level or above should not have to describe binary representations of instructions. Someone writing a garbage collector might need to describe the stack-frame layouts determined by a calling convention, but he should not have to describe the instructions used in the calling sequences that establish those layouts.
- Components, especially those describing "core aspects," should be reusable. For example, writers of all specifications should benefit from having a common formalization of what it means to be a "SPARC instruction supported by the bare hardware."

CSDL forms a family of languages because all family members have the same view of two core aspects of machines: instructions and state. This report explains the CSDL view of these core aspects, and it presents a rationale for and examples of the most important member of the CSDL family, which describes the semantics of machine instructions.

Core aspects for CSDL

To identify core aspects to be used throughout the CSDL family, we examined descriptions used to help retarget a variety of systems-level tools. These tools included an optimizer (Benitez and Davidson 1988), a debugger (Ramsey and Hanson 1992), an instruction scheduler (Proebsting and Fraser 1994), a call-sequence generator (Bailey and Davidson 1995), a linker (Fernández 1995), and an executable editor (Larus and Schnarr 1995). Cursory inspection showed no single common aspect of a machine used in descriptions for all of these tools, but a closer look revealed that all the descriptions refer either to the machine's *instruction set* or to its *storage locations*, and some to both. For example, the specifications used by the scheduler and linker refer only to the machine's instructions and the properties thereof. The specifications used in the call-sequence generator and in the debugger's stack walker refer only to storage, explaining in detail how values move between registers and memory. The specifications used in the optimizer and the executable editor refer both to instructions and to storage, and in particular, to how instructions change the contents of storage. From these observations, we have chosen to require that languages in the CSDL family refer to instructions, storage, or both, and that they use the models of instructions and storage presented below.

Instructions

The CSDL model of an *instruction set* is a list of instructions together with some identifying information about their operands. Although instruction names in assembly languages are typically overloaded, CSDL requires instructions to have unique names, because tools often need uniquely named code for each instruction in an instruction set. For example, an assembler might use a unique C procedure to encode each instruction, or an executable editor might use a unique element of a C union to represent an instance of each instruction.

An individual instruction is viewed as a function or constructor that, when applied to operands, produces something interesting: a binary representation, an assembly-language string, a semantics, etc. Instruction descriptions that use the CSDL core will include the names and types of the operands of each instruction. Types will include integers of various sizes, but it will also be possible to introduce new types to define such machine-dependent concepts as effective addresses. Values of these new types may be created by applying suitable constructors; for example, Chapter 6 shows how a Pentium effective address may be formed by applying any of

9 constructors, each one representing a different addressing mode.

The structure defined by CSDL constructors and their operands can be viewed as a discriminated-union type. It is also equivalent to an ASDL grammar (Wang *et al.* 1997), without recursion, in which the start symbol is "instruction," other nonterminals refer to machine-level concepts like "effective address" or "integer-instruction operand," and terminal symbols are defined by integers or addresses. This structure is a simplification of the "constructor" from the Specification Language for Encoding and Decoding (SLED) of the New Jersey Machine-Code Toolkit (Ramsey and Fernández 1997). It is determined by the machine, independent of any tool, so it should be useful in any specification language that deals with individual machine instructions. For example, the `nML` machine-description language (Fauth, Praet, and Freericks 1995) uses this structure, although `nML` is otherwise quite different from SLED.

Properties of instructions

Experience with the New Jersey Machine-Code Toolkit shows that many applications can be built from specifications that discuss only instructions and their properties, with no reference to storage. Such applications include assemblers and disassemblers, as well as code generators that work by pattern matching on the names of instructions.

We specify properties of instructions in a compositional style, so instructions' properties are functions of the properties of their operands. We formalize that style using attribute grammars. For example, assembly-language representations of instructions can be computed as synthesized attributes, where the attributes are strings. Binary representations can also be computed as attributes, where the attributes are SLED "patterns." Attributes readily support machines like the 68000 family, in which the representations of effective addresses depend on context.

Storage

The CSDL model of a *storage space* is a sequence of mutable cells. A storage space is like an array; cells are all the same size, and they are indexed by integers. For example, a typical microprocessor has a memory made up of 8-bit cells (bytes) and a register file made up of 32-bit cells. The number of cells in a storage space may be left unspecified.

The state of a machine can be described as the contents of a collection of storage spaces. We use storage spaces to model main memory, general-purpose registers, special-purpose registers, condition codes, and so on.

Experience with CCL, a Calling Convention Language (Bailey and Davidson 1995), shows that applications can be built from specifications that discuss only storage, with no reference to instructions. For example, calling conventions can be described by discussing the placement of parameters in storage cells and the the effects of calls and returns on storage. From a CCL description one can generate procedure prologs and epilogs in a compiler, and one can also generate code to test compilers' implementations of calling conventions (Bailey and Davidson 1996). One might be able to derive stack walkers or exception-handling code from similar descriptions. A garbage collector may need to know which locations in storage can contain pointers; this is a property of an application, not of a machine, but it can be described using the CSDL storage core. (We want to describe application properties as well as machine properties, but we want to keep them separate.)

Languages in the CSDL family may refer to individual *locations.* Ways of writing locations may vary, but each one must resolve to a name of a storage space and an integer offset identifying a cell within that storage space.

Combining instructions and storage

An architecture manual tells programmers what constitutes the state of the processor, and it lists instructions and their operands, but the most important thing it does is explain the semantics of each instruction in terms of that instruction's effects on the state of the processor. Many applications can be built based on this information, but some require more detail than others. For example,

- Information about control flow is enough to build control-flow graphs.
- Information about calling conventions may be enough to recognize procedure calls and returns.
- Information about locations read and written is enough to build data-flow graphs. Such graphs, together with the ability to match individual instructions, may suffice to build code-editing tools like EEL (Larus and Schnarr 1995) or ATOM (Srivastava and Eustace 1994).
- Information about register-transfer semantics is enough to build code improvers in the style of PO (Davidson and Fraser 1980), vpo (Benitez and Davidson 1988), and `gcc` (Stallman 1992). These code improvers work by pattern matching, so they need not know what *all* of the register-transfer operators do. They do, however, need at least a partial semantics, to perform strength reduction, constant folding, and similar transformations.
- To build a code generator, one needs to know more about the operators used in the register transfers. In particular, one needs to know enough so that one can find a sequence of instructions to implement each operation in the compiler's intermediate code. It is sufficient to know that the intermediate code and the instructions compute the same operation; one need not know exactly what the operations do to the bits.
- To build an emulator like SPIM (Larus 1990) or a binary translator like FX!32 (Thompson 1996), one needs enough information about the operations in the register transfers to interpret the the effect of each register transfer on each bit of the processor's state.

We believe that these applications and more can be served by providing *register-transfer semantics* for instructions. The effect of a particular instruction can be specified as a *register-transfer list* (RTL), which modifies storage cells. As with other properties of instructions, the RTL is computed using an attribute grammar.

In principle, a single register-transfer description could support all of the different uses above. A single RTL could be given different interpretations, depending on its intended use. We believe we can support this mode of specification by prescribing a rigid form for RTLs, while supplying suitable abstraction mechanisms for use within that form. Abstraction mechanisms support our goals of enabling application writers to leave irrelevant facts unspecified. For example, we want to make it easy to write an RTL that means "register rd is assigned an unspecified function of registers rs and rt."

Languages in the CSDL family

We consider SLED, for specifying representations of instructions (Ramsey and Fernández 1997), and CCL, for specifying calling conventions (Bailey and Davidson 1995), to be the first languages in the CSDL family. We are developing a new language, λ-RTL, for specifying instruction semantics. We expect that CSDL will expand to include languages for specifying properties of memory hierarchies and of pipelines. Indeed, the simple functional-resource languages of Proebsting and Fraser (1994) and Bala and Rubin (1995) fit nicely into the CSDL framework.

This report focuses on λ-RTL. The report does not give a specification or definition of λ-RTL, because λ-RTL is not sufficiently developed for a specification or definition to be worthwhile. Instead, this report presents some essential properties of λ-RTL, and it gives examples of machine specifications using the current, imperfect version. Chapter 2 describes the form of register transfer lists used in λ-RTL. Chapter 3 presents λ-RTL and discusses its translation into RTLs. Chapter 4 describe some basic content which fits into that form, and which we believe will be useful for describing many machines. Chapters 5 and 6 present excerpts from λ-RTL descriptions of the SPARC and Pentium processors, respectively.

Chapter 2

Register Transfer Lists

Computer scientists have used register transfers in many different forms. For λ-RTL, we have chosen a form designed for use by tools, not by people. We have therefore insisted that as much information as possible be explicit in the RTL itself. Under our current plan,

- RTLs are represented as trees.
- All operators are fully disambiguated, e.g., as to type and size.
- There is no aliasing of locations.
- Fetches are explicit, as are changes in the size or type of data.
- Stores are annotated with the size of the data stored.
- Explicit tree nodes specify byte order. More generally, they specify how to transfer data between storage spaces of different granularity.
- RTL should be a typed representation. We plan to generate RTL type checkers from CSDL specifications. Optimizations and other transformations that are intended to preserve semantics should also preserve the property of being well typed. Typing is a good way to catch bugs in optimizers (Morrisett 1995).

The form of RTLs proposed here may be suitable not just for specification, but also for use in the implementations of compilers, binary translators, and other tools.

⟨*ASDL specification of the form of RTLs*⟩≡
```
ty = (int) -- size of a value, in bits
exp = CONST (const)
    | FETCH (location, ty)
    | APP (operator, exp*)
location = AGG(aggregation, cell)
cell = CELL(space, exp)
effect = STORE(location dst, exp src, ty)
       | KILL(location)
       | KILLALL(space)
guarded = GUARD(exp, effect)
rtl = RTL (guarded*)
```

Figure 2.1: ASDL specification of the form of RTLs

Form of RTLs

Figure 2.1 uses the Zephyr Abstract Syntax Description Language (Wang *et al.* 1997) to show the form of RTLs. A register transfer list is a list of guarded effects. Each effect represents the transfer of a value into a storage location, i.e., a store operation. The transfer takes place only if the guard (an expression) evaluates to **true**. Locations may be single cells or aggregates of consecutive cells within a storage space. Values are computed by expressions without side effects, which simplifies analysis and transformation. These expressions include integer constants, fetches from locations, and applications of *RTL operators* to lists of expressions. Effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement, so one RTL represents one change of state. This decision makes it possible to specify swap instructions without having to introduce temporary locations.

Not every effect is a true assignment. A *kill* effect changes the value in a storage location in an undefined way. Kill effects are needed to model such architectural specifications as "the effect of a logical instruction on the AF flag is undefined."

The "kill all" effect in Figure 2.1 kills all of the cells in the given storage space. It is needed to give a semantics to certain phrases of λ-RTL. We haven't demonstrated a need for it yet, but it may be useful to model stores into arbitrary memory locations.

As an example of a typical RTL, consider a SPARC load instruction using the displacement addressing mode, written in the SPARC assembly language as

⟨*sample SPARC instruction*⟩≡
```
  ld [%sp-12], %i0
```
Although we would not want to specify just a single instance of a single instruction, the effect of this load instruction might be notated in λ-RTL as follows:[1]

⟨*λ-RTL for sample instruction*⟩≡
```
  $r[24] <-- $m[$r[14]+(~12)]
```
because the stack pointer is register 14 and register `i0` is register 24. The corresponding RTL is much more verbose, with the sizes of all quantities identified explicitly, as a fully disambiguated tree:

```
                STORE #32
               /         \
   AGGB #32 #32           FETCH #32
        |                     |
     LOC 'r'              AGGB #8 #32
        |                     |
    CONST #5               LOC 'm'
        |                     |
       24                  ADD #32
                          /       \
              FETCH #32             SX #13 #32
                  |                     |
            AGGB #32 #32           CONST #13
                  |                     |
               LOC 'r'                -12
                  |
              CONST #5
                  |
                 14
```

The various constants labeled with hash marks, like `#32`, indicate the number of bits in arguments, results, or data being transferred. Such constants will fit into a generalization of the Hindley-Milner type system (Milner 1978).

[1]The ~ in ~12 is a unary minus.

Figure 2.2 shows the operators used in this tree. The left child of the `STORE` is a subtree representing the location consisting of the single register `i0`, which is register 24. The right-hand child represents a 32-bit word (a big-endian aggregration of four bytes) fetched from memory at the address given by the subtree rooted at `ADD`. This node adds the contents of the stack pointer (register 14) to the constant −12. The constant is a 13-bit constant, and the `SX` operator sign-extends it to 32 bits, so it can be added to the stack pointer.

Types in RTLs

RTL is intended to be a typed format. The type system is under development, but it is expected to include the following types:

`#`*n* `bits`	A value that is *n* bits wide.
`#`*n* `loc`	A location containing an *n*-bit value.
`#`*n* `cells`	One of a sequence of *n*-bit storage cells, which can be aggregated together to make a larger location, as by the `AGGB` nodes in the example tree.
`bool`	A Boolean condition.
`effect`	A side effect on storage.

Figure 2.2 uses these types to show the types of the operators used above. We plan to extend Milner's type inference to this system, so that those writing specifications in λ-RTL can omit types and widths. Unlike in ML, type inference alone will not guarantee that terms make sense; in general, it will be necessary to check additional constraints. For example, in the RTL shown above, it would be necessary to check that the signed integer −12 can be represented using 13 bits, and that 32 is a multiple of both 8 and 32.

RTL languages and translation

Although we have prescribed the form of RTLs, we can't write any RTLs until we choose a set of lo-

`STORE` : $\forall\#n.\#n\,\texttt{loc} \times \#n\,\texttt{bits} \rightarrow \texttt{effect}$	Store an n-bit value in a given location. The type indicates that for any n, `STORE` $\#n$ takes an n-bit location and an n-bit value and produces an effect.
`FETCH` : $\forall\#n.\#n\,\texttt{loc} \rightarrow \#n\,\texttt{bits}$	For any n, `FETCH` $\#n$ takes an n-bit location and returns the n-bit value stored in that location.
`AGGB` : $\forall\#n.\forall\#w.\#n\,\texttt{cells} \rightarrow \#w\,\texttt{loc}$	For any n and w, `AGGB` $\#n$ $\#w$ aggregates an integral number of n-bit cells into a w-bit location, making the first cell the most significant part of the new location, i.e., using big-endian byte order. w must be a multiple of n. (w and n are mnemonic for wide and narrow.)
`LOC 'm'` : $\#32\,\texttt{bits} \rightarrow \#8\,\texttt{cells}$	Given a 32-bit address, `LOC 'm'` returns the 8-bit cell in memory referred to by that address.
`LOC 'r'` : $\#5\,\texttt{bits} \rightarrow \#32\,\texttt{cells}$	Given a 5-bit register number, `LOC 'r'` returns the corresponding 32-bit register (a mutable cell).
`ADD` : $\forall\#n.\#n\,\texttt{bits} \times \#n\,\texttt{bits} \rightarrow \#n\,\texttt{bits}$	For any n, `ADD` $\#n$ takes two n-bit values and returns their n-bit sum. Carry and overflow are ignored.
`SX` : $\forall\#n.\forall\#w.\#n\,\texttt{bits} \rightarrow \#w\,\texttt{bits}$	For any n and w, `SX` $\#n$ $\#w$ takes an n-bit value, interprets it as a two's-complement signed integer, and sign-extends it to produce a w-bit representation of the same value. w must be greater than n.
`CONST` : $\forall\#n.\langle\text{constant}\rangle \rightarrow \#n\,\texttt{bits}$	For any n, `CONST` $\#n$ k represents the n-bit constant k. k must be representable in n bits. The same k could be used with different ns.

Figure 2.2: Some RTL operators and their types

cations and a set of RTL operators. These choices define an *RTL language*. Any specification written in λ-RTL can introduce new storage spaces (and therefore new locations) and new RTL operators, determining an RTL language for use in that specification.

For use during compilation, we restrict RTLs to a subset of a full RTL language. Each RTL used in a back end based on the *vpo* optimizer (Benitez and Davidson 1988) must satisfy the *vpo invariant* for the target machine. An RTL satisfies that invariant if and only if it can be represented in a single instruction on the target machine. All RTLs manipulated by *vpo* satisfy this invariant, so *vpo* can stop and emit machine code at any time. We use the name X-RTLs refer to the set of RTLs satisfying the *vpo* invariant for machine X.

Interpretations of RTLs

One reason we restrict the form of RTLs is to limit their possible meanings or interpretations. For example, access to the mutable state of a machine is available only through the fetch and store operations built into the RTL form, so we can easily tell what state is changed by an RTL and how that change depends on the previous state. Researchers and tools working with our form of RTLs have freedom to define interpretations of only two parts of the RTL form: aggregations and operators.

RTL aggregations specify byte order. More generally, aggregations make it possible to write an RTL that stores a w-bit value in (or fetches a w-bit value from) k consecutive n-bit locations, provided that $w = kn$. Such an aggregation has type $\#n\,\texttt{cells} \rightarrow \#w\,\texttt{loc}$, and its interpretation must be a bijection between a single w-bit value and k n-

bit values. Moreover, when $w = n$, the bijection must be the identity function. Storing uses the bijection, and fetching uses its inverse, making it possible to combine RTLs using forward substitution. Little-endian and big-endian aggregations will be built into λ-RTL, as will an “identity aggregation” that is defined only when $w = n$. We imagine that users could define other aggregations by giving systems of equations, as in Ramsey (1996).

RTL operators must be interpreted as pure functions on bit vectors. Consequently, the result of applying an RTL operator must not depend on processor state; the operator must give the same answer every time. Chapter 4 presents a collection of RTL operators that we expect can be used to describe many different machines. We don't expect this set to be complete; on the contrary, we expect that any machine description written in λ-RTL will introduce a handful of new RTL operators which will be unique to that machine.

Chapter 3

Using λ-RTL to specify register transfer lists

Bare RTLs are both spartan and verbose. Expressions do not include if-then-else, so conditionals must be represented by using guards on effects. There is no expression meaning "undefined;" assignments of undefined values must be specified using a kill effect. These restrictions, and the requirement that operations be fully disambiguated, make RTLs a form that is good for manipulation by tools but not so good for writing specifications.

λ-RTL is a metalanguage that enables specification writers to attach RTL trees to SLED-like constructors without having to write everything explicitly. Eventually, λ-RTL will be a higher-order, strongly typed, polymorphic, pure functional language based largely on Standard ML (Milner, Tofte, and Harper 1990). A variation on the Hindley-Milner type system will make it possible to write flexible, type-safe functions without having to write types explicitly. The current implementation of λ-RTL is, however, untyped.

Design considerations

We have drawn on our experience with SLED (Ramsey and Fernández 1997) to identify mechanisms and properties that are desirable in any CSDL language, including λ-RTL. These include:

- Use of constructors to provide an abstract view of the machine's instruction set, and use of attributes to specify properties of instructions. These mechanisms are close kin to attribute grammars and to the denotational approach to semantics, both of which have long histories of utility.
- Use of default, unnamed attributes (for example, binary representation or default constructor type), sometimes supported by special syntax. By providing a default case that does not have to be named, we can unclutter specifications. For example, when we refer to an operand of type `Address`, the location designated by that effective address should be the default meaning.
- Use of a specialized sublanguage for defining attributes. SLED uses patterns to specify binary representations; λ-RTL uses the prescribed forms of RTLs, to be augmented by a sophisticated type system.
- Language constructs that help eliminate repetition by permitting simultaneous description of many instructions at once. Such constructs reduce the number of opportunities for errors, and they help keep specifications concise and readable. λ-RTL includes two such constructs: first class functions, and a generalization of SLED's grouping construct, which itself is based on Icon generators (Griswold 1982).

- Formal notation that matches informal descriptions found in architecture manuals. The architecture manual is the standard model of the machine that is shared by all tool builders, so this close match not only helps people read specifications, it helps them write correct specifications. λ-RTL enables users to define their own infix operators, and Chapters 4, 5, and 6 show how we have used this capability to create an ISP-like notation for RTLs.

We expect to be able to analyze λ-RTL specifications for internal consistency and "plausibility." For example, it should be possible to identify cases in which a register number is mistakenly used as an immediate value instead of as an offset into the storage space modeling the register file.

Restrictions eased in λ-RTL

λ-RTL descriptions are easier to write than bare RTLs in three ways: grouping and higher-order functions can help eliminate repetition, the type system will eliminate the need to write sizes explicitly, and λ-RTL relaxes several of the restrictions on the form of RTLs. In particular,

- In λ-RTL, it is not necessary to write fetches explicitly.
- λ-RTL gives the illusion that bit slices (subfields) are locations that can be assigned to.
- λ-RTL gives the illusion that aggregates of cells are locations that can be assigned to, and it is not usually necessary to write aggregations explicitly.

Implicit fetches

Most programmers are used to writing $x := x + 1$ and having the x on the left denote a location while the x on the right denotes the value stored in that location. Typical compilers identify "lvalue contexts" and "rvalue" contexts and automatically insert fetches in rvalue contexts. We do the same in λ-RTL, but instead of using syntax to identify the contexts, we intend to use types.

We use types and not syntax because λ-RTL has no special syntax for writing RTL assignments. Instead of an assignment syntax, λ-RTL provides a built-in store function that accepts a location and a value and produces an effect. Any user-defined function might result in a call to the built-in store, so we have to recognize the right and left contexts by their types. Thus, if a location is used where a value is expected, we insert a fetch.

λ-RTL does almost everything with functions, not syntax, so the writer of a specification can normally redefine the meaning of a notation by defining a new function with the same name. This strategy doesn't work with implicit fetches because there is no explicit notation associated with a fetch. We want users to be able to control the meanings of these fetches, however, because many machines have resources that are almost, but not quite, sequences of mutable cells. For example, SPARC registers can be viewed as a collection of 32 mutable cells, except that register 0 is not mutable and always contains 0. We would like users to be able to define special meanings for "fetch from register 0" and "store into register 0" so the rest of the specification can pretend that the registers are an ordinary storage space. We do so by permitting users to attach fetch and store methods to each storage space. Methods not given explicitly default to the standard fetch and store operators. Sample fetch and store methods for the SPARC are shown on page 25. If we wanted, we could use fetch and store methods to describe the true implementation of SPARC registers, in which "registers" 8 through 31 denote locations accessed indirectly through the register-window pointer (CWP).

Slices

Many machine instructions manipulate fragments of a word stored in a mutable cell. For example, some machines represent condition codes as individual bits within a program status word. Many machines have instructions that, for example, assign to the least-significant 8 bits of a 32-bit register.

To make it easy to specify such instructions, λ-RTL creates the illusion that a sub-range or "slice" of a cell can be a location in its own right, one that is a suitable argument for a fetch or store operation. This illusion helps keep machine descriptions readable; for example, an effect that sets the SPARC overflow bit simply assigns to it, hiding the fact that it is buried in a program status word that has to be fetched, modified, and stored.

λ-RTL uses a special syntax for slices because the slicing operation is overloaded; it can be applied to locations or to values. Examples of the syntax include

x`@[`k`]`	Bit k of x. By default, bit 0 is the least significant bit. A future version of λ-RTL may make it possible to change the numbering.
x`@[`k_1`..`k_2`]`	Bits k_1 through k_2 of x, inclusive.
x`@[`k `bits at` e`]`	A k-bit slice of x, with the least significant bit at e.

k's denote integer constants, e's denote expressions, and x's may denote values or locations. In all cases the size of the slice is known statically, so its type can be computed automatically. We use the Greek letter σ to stand for any of these slice specifications.

Given a slice specification σ, $\mathtt{SLICE}_\sigma$ is an overloaded function that maps locations to locations or values to values. The illusion that slices are locations is implemented by rewriting, according to the following rules:

$$\begin{array}{lcl} \mathtt{FETCH}(\mathtt{SLICE}_\sigma\, l) & \mapsto & \mathtt{SLICE}_\sigma(\mathtt{FETCH}\, l) \\ \mathtt{SLICE}_\sigma\, l \leftarrow n & \mapsto & l \leftarrow \mathtt{bitInsert}_\sigma(\mathtt{FETCH}\, l, n) \end{array}$$

where l is a location and n is a value. After the rewriting, all slices operate on bit vectors, and all fetches and stores operate on true locations. Invocation of user-defined fetch and store methods takes place *after* the rewriting of slices. This ordering makes it possible to use fetch and store methods to define cell-like abstractions, while ensuring that the meaning of slicing is always consistent with respect to such abstractions.

Implicit aggregation

λ-RTL provides the special syntax `$`*space*`[`*offset*`]` for references to mutable cells. The *offset* can be an arbitrary expression, but the *space* must be a literal name, so that λ-RTL can identify the storage space and use appropriate fetch and store methods. To make this cell a location, λ-RTL applies an aggregation, which is also associated with the storage space as a method. The default method is the identity aggregation, which permits only "aggregates" of a single cell.

When little-endian, big-endian, or other aggregations are used, λ-RTL will infer the size of the aggregate. We have not yet determined the rules to be used for the inference, but we hope at least to support the automatic inference that four 8-bit bytes must be aggregated to form a value that goes into a 32-bit register.

The implementation of λ-RTL used in subsequent chapters does not support aggregation.

Overview of λ-RTL

From the point of view of external tools, the output of a λ-RTL specification is a set of bindings of values to attributes of constructors. The part of λ-RTL used to specify values is a pure functional language without recursion. The implementation used in this report is untyped, but λ-RTL will eventually use a polymorphic typed calculus. Many features of λ-RTL are inspired by Standard ML.

To describe syntax, we use an EBNF grammar with standard metasymbols for $\big\{$sequences$\big\}$, $\big[$optional constructs$\big]$, and $\big($alternative $\big|$ choices$\big)$. We use large metasymbols to help distinguish them from literals. Terminal symbols given literally appear in `typewriter` font. Other terminal symbols and all nonterminals appear in *italic* font. Excerpts from the grammar begin with the name of a nonterminal followed by the $\Rightarrow$ ("produces") symbol.

A λ-RTL specification is a sequence of *structures*. λ-RTL uses structures to organize the name space of values, and structures will play a role in a modules system for λ-RTL. A value `x` declared in a structure `S` is visible within `S` as just `x`, but outside `S` it can only be referred to as `S.x`. Structures are written

structure ⇒
  `structure` *struct-name* `is struct` $\{$*declaration*$\}$ `end`

Constructors and storage spaces occupy their own individual name spaces, which are flat. RTL operators are also expected to occupy a separate, flat name space, but within λ-RTL they are not distinguished from other kinds of values. Eventually, λ-RTL will check that the same name is not used to declare distinct RTL operators in two different structures.

Declarations are either top-level or inner declarations. Inner declarations may appear anywhere, but top-level declarations may appear only outside of function and value definitions. Top-level declarations appear only in contexts that guarantee they will be evaluated exactly once.

Top-level declarations

λ-RTL's top-level declarations may introduce structures, locations, storage spaces, RTL operators, or bindings to attributes of constructors.

Storage spaces are introduced by `storage`; they name the storage space, give its granularity (width of an individual cell) and size (number of cells), and possibly fetch, store, or aggregation methods.

RTL operators are introduced by:

declaration ⇒
  `rtlop` $\big($*operator-name* $\big|$ `[` $\{$*operator-name*$\}$ `]`$\big)$

For example,

```
rtlop bit
rtlop [= <>]
```

introduces three new RTL operators.

In the current implementation of λ-RTL, `rtlop` introduces an arbitrary, abstract value. Because it is the only abstraction mechanism available, it is sometimes put to odd uses. A future version of λ-RTL will require that the type of the RTL operator(s) be specified. λ-RTL will support polymorphic operators applicable to arguments of any size. This parametric polymorphism will let programmers use to use the same operator to add both 16- and 32-bit integers. λ-RTL will not support overloading, so a different operator will be required to add floating-point values.

Attribute bindings are introduced as follows:

declaration ⇒
  attribute `of` $\{$*constructor* `is` *expression*$\}$

attribute ⇒
  `default attribute of`
  | `attribute` $\big($*attribute-name* $\big|$ `default`$\big)$ `of`

constructor ⇒
  opcode `(` $\big[$*operand-name* $\{$, *operand-name*$\}\big]$ `)`

Each *expression* must be terminated by a newline; if an expression doesn't fit on one line, it must contain an open parenthesis or brace so that the λ-RTL compiler knows to continue to the closing delimiter.

For example,

```
default attribute of
  nop() is do_nothing
```

defines the effect of a no-op.

An expression bound to an attribute must denote a legal fragment of RTL; in the current implementation, this means a value, a location, or an RTL. More precisely, an expression must denote an *RTL template*. An RTL template becomes a fragment of RTL when suitable fragments of RTL are substituted for the constructor's operands. An RTL template is an RTL, except that

- Names of constructor operands may be used to stand for RTL expressions. (Eventually we hope to use names of operands to stand for locations as well as values, but the current implementation does not support this usage.)
- Tuple or record selections from constructor operands may be used to stand for RTL expressions.
- Vectors of RTL expressions subscripted by constructor operands may be used in place of RTL expressions.

A tuple or record containing RTL templates is also considered an RTL template. The most notable consequence of these rules is that no expression whose normal form contains a λ-abstraction is an RTL template; λ-abstractions must be applied to arguments.

Eventually there will be a precise, formal definition of RTL template.

Inner declarations

Inner declarations may appear anywhere a top-level declaration may appear, and they may also appear in `let`-expressions, where they are the only kind of declaration permitted. Their typical usage is to bind local values in function bodies. Inner declarations include value bindings, function bindings, and fixity declarations. Value and function bindings are written as

declaration ⇒
  `val` *value-spec* `is` *expression*
  | `fun` *value-spec* *argument-pattern* `is` *expression*

value-spec ⇒ *name* | `[` $\{$*name*$\}$ `]`

The square brackets in *value-spec* represent λ-RTL's grouping mechanism, in which the *expression* actually denotes a *sequence* of values, and each value in the sequence is bound of the corresponding name on the left.

As examples, consider

```
val do_nothing is RTL.SKIP
fun bool n is n <> 0
val [eq ne] is [(=) (<>)]
```

λ-RTL supports user-defined infix operators with arbitrarily many levels of precedence. Precedence levels range from 999, which represents the highest possible precedence (tightest binding) for an infix operator, down to the most negative integer on the machine (loosest binding). The syntax is

declaration ⇒
  (`infixl` | `infixr` | `infixn`) *precedence* *value-spec*
  | `nonfix` *value-spec*

Infix operators must be declared to be left-associative, right-associative, or non-associative with other operators of the same precedence. For example, we can make equality testing infix and non-associative using

```
infixn 5 [= <>]
```

A single instance of an infix operator can be made "nonfix" (treated as an ordinary value) by enclosing it in parentheses. The `nonfix` declaration makes this behavior permanent.

Expressions

λ-RTL expressions include records, tuples, integers, conditionals, and functions (λ-abstractions). As in ML, the juxtaposition of two expressions indicates function application, and *function application has higher precedence than any infix operator*. Figure 3.1 shows the syntax for expressions.

Grouping

A *group* may appear in place of an expression anywhere in λ-RTL. Whereas an expression denotes a single value, a group denotes a sequence of values. All operations except binding distribute over the group, so if the expression in a declaration contains a group, then that expression denotes a sequence of values, no matter how deeply nested the group is. The most common operations to distribute over a group are function application and tuple formation.

The usual syntax for a group is a list of expressions in square brackets. The expressions need not be of the same type. Ordinary function application is inoperative at the top level of a group, so the group `[f x]` is a group containing two values, as is `[f (x)]`. To use function application or infix operators within a group, one must include the whole application in parentheses, e.g., `[(f x)]` or `[(f(x))]`. In addition to the usual syntax, λ-RTL provides two kinds of special syntax to create groups of integers. All use square brackets:

expression ⇒
  `[` $\{$*expression*$\}$ `]`
  | `[` *expression* `..` *expression* `]`
  | `[` *expression* `to` *expression*
    [`by` *expression* | `columns` *expression*] `]`

```
expression ⇒
    ( expression {, expression} )                                       Tuple
  | { member-name is expression{, member-name is expression} }        Record
  | [. expression {, expression} .]                                     Vector
  | let {declaration} in expression end                                 Local declarations
  | if expression then expression else expression fi                    Conditional
  | \ argument-pattern . expression                                     Function
  | expression.member-name                                              Selection
  | $ space-identifier [ expression ]                                   Location
  | expression @ [slice-specification]                                  Slice
  | expression expression                                               Function application
  | identifier                                                          Named value
  | literal-constant                                                    Constant
```

Figure 3.1: λ-RTL expressions

In the latter two cases, the constituent expressions must be compile-time constants.

Groups are evaluated left to right, last-in, first-out, so for example the expression

```
([a b], [1 2])
```

denotes the same sequence of four expressions as

```
[(a, 1) (a, 2) (b, 1) (b, 2)]
```

The evaluation rule and the design of groups in general are based on the *generator* mechanism of the Icon programming language (Griswold 1982).

The primary role of groups is to make it easy to specify multiple machine instructions in a single attribute binding. This is done by using *value-spec* (either a single name or a sequence of names in square brackets) to form the opcodes of constructors. Multiple *value-spec*s may be used to combine, for example, a group of operations with a group of suffixes. Chapters 5 and 6 have examples.

opcode ⇒ *value-spec*{^*value-spec*}

Lexical conventions

Like identifiers in ML, λ-RTL identifiers may be alphanumeric or symbolic. An alphanumeric identifier begins with a letter and continues with letters, digits, underscores, and primes. A symbolic identifier is a sequence of the symbols

```
!&+/\:<=>?@~|*‘%;-^#.
```

Such an identifier may begin with any symbol except `#`.

Identifiers consisting solely of two or more dashes introduce comments; the comment includes the dashes and continues to the end of the line. For example, `--` and `---` introduce comments, but `-->` and `--*--` do not. A comment is equivalent to a newline.

An integer literal is a sequence of decimal digits; λ-RTL has no floating-point literals. λ-RTL also supports hexadecimal literals beginning with `0x`, octal literals beginning with `0`, and binary literals beginning with `0b`.

Literals and identifiers preceded by `#` are intended to be used to represent widths in λ-RTL's forthcoming type system; in the current implementation, they are equivalent to ordinary literals and identifiers.

The initial basis

Most of the RTL-specific content of λ-RTL is in the initial basis, i.e., the collection of predefined functions and values. Figure 3.2 shows λ-RTL's initial basis.

Boolean constants:	
`true`	Truth
`false`	Falsehood
RTL operators and other functions used to create RTLs:	
`RTL.STORE`	Takes location and value, produces effect.
`RTL.FETCH`	Fetches value from location.
`RTL.SKIP`	The empty RTL; an effect that does nothing.
`RTL.GUARD`	Takes a Boolean expression and an effect and produces an effect.
`RTL.NOT`	Boolean negation.
`RTL.EFFECTS`	Takes two effects (RTLs) and composes them so they take place simultaneously (list append on list of effects).
Functions on vectors:	
`sub`	Vector subscript, a left-associative infix operator with precedence 5.
`Vector.spanning`	A curried function of two arguments. `Vector.spanning` x y produces the vector `[.`$x, x+1, \ldots, y$`.]`.
`Vector.foldr`	A higher-order function used to visit every element of a vector. Like `Vector.foldr` in Standard ML '97.

Figure 3.2: λ-RTL's initial basis

Style

λ-RTL provides expressive power with few restrictions. Writers of specifications use the functions in λ-RTL's initial basis to create RTLs that correspond to the form prescribed in Chapter 2. Writers can also introduce new RTL operators. This freedom gives us ample scope for experimenting with different styles of description—as well as ample rope with which to hang ourselves. We expect our experiments to lead to a library of useful RTL operators, as well as useful λ-RTL functions. Chapter 4 gives a preliminary collection of RTL operators.

λ-RTL provides no looping or recursion constructs. Loops whose sizes are known in advance can be simulated by using `Vector.foldr` and `Vector.spanning`. Because this style will be familiar only to those well versed in functional programming, we expect eventually to provide some syntactic sugar for it.

Differences between λ-RTL and Standard ML

Although λ-RTL is inspired in large part by Standard ML, there are significant differences, which may interest ML programmers. Syntactic differences include:

- The defining connective is `is`, not `=`.
- `if` expressions must be terminated by `fi`.
- Identifiers declared infix must be explicitly identified as left-, right-, or non-associative. Arbitrarily many levels of precedence are available.
- Dot notation is used to select elements from records as well as from structures.
- Comments begin with an identifier consisting of n dashes, where n is at least 2. Comments end at the end of a line.

There are also some semantic restrictions:

- There are no side effects (mutation, exceptions), so order of evaluation does not matter.
- Functions may not be recursive, so the normal form of all expressions can be computed at compile time.
- There are no datatype constructors. As a corollary, the only patterns used in function definitions are those that match tuples or records.

Chapter 4

Basic RTL operators

This chapter describes RTL operators and functions that we expect to be useful for describing many different machines. As we do throughout this document, we use the `noweb` literate-programming tool (Ramsey 1994) to present examples of λ-RTL. `Noweb` extracts the code from the same files used to produce this document, so we can run the examples through the λ-RTL compiler and make sure everything compiles.

A `noweb` file contains explanatory text interleaved with named "code chunks," which contain source code and references to other code chunks. The names of chunks appear italicized and in angle brackets, as in the following C code:

18a ⟨*summarize the problem* 18a⟩≡ 18c ▷
```
  printf("Inputs are: ");
  ⟨for every i that is an input, print i and a blank 18b⟩
  printf("\n");
```

18b ⟨*for every* `i` *that is an input, print* `i` *and a blank* 18b⟩≡ (18a)
```
  for (i = 1; i < argc; i++)
    printf("%s ", argv[i]);
```

The ≡ sign indicates the definition of a chunk. Definitions of a chunk can be continued in a later chunk; `noweb` concatenates their contents. Such a concatenation is indicated by a +≡ sign in the definition:

18c ⟨*summarize the problem* 18a⟩+≡ ◁ 18a
```
  printf("Must process %d files\n", argc-1);
```

`noweb` adds navigational aids to the document. Each chunk name ends with the number of the page on which the chunk's definition begins. When more than one definition appears on a page, they are distinguished by appending lower-case letters to the page number. When a chunk's definition is continued, `noweb` includes pointers to the previous and next definitions, written "◁ 18a" and "18c ▷." The notation "(18a)" shows where a chunk is used.

Because λ-RTL does not yet have modules, we use `noweb` to include the chunk ⟨*RTL basics* 19a⟩ in our descriptions of the SPARC and the Pentium.

4.1 Building RTLs

λ-RTL is intended for building RTLs, not analying them, so it doesn't expose all the structure of RTLs; single effects, guarded effects, and full RTLs (lists of guarded effects) all have the same type `effect`. The initial basis provides ways to create and combine effects. We define a more compact, infix notation for parts of the basis.

19a ⟨*RTL basics* 19a⟩≡ 19b ▷

```
val do_nothing is RTL.SKIP  -- type will be effect
val --> is RTL.GUARD        -- type will be bool * effect -> effect
val <-- is RTL.STORE        -- type will be #n loc * #n bits -> effect
infixr 1 -->
infixn 2 <--
```

In the underlying representation, the semantics of these operations is extended to full RTLs in the obvious way. For example, `RTL.STORE` produces an RTL containing a single effect whose guard is `true`.

We don't need an explicit `FETCH`, because fetches are inserted automatically, and we don't need an explicit `CONST`, because λ-RTL compiles literal constants into RTL `CONST` nodes.

We use a semicolon to combine simultaneous effects. We plan eventually to have an `andthen` operator to indicate sequential composition of effects, which would presumably be implemented by forward substitution. Substitution isn't implemented in the current λ-RTL, so we make `andthen` equivalent to simultaneous effects, even though this semantics for `andthen` is incorrect.

19b ⟨*RTL basics* 19a⟩+≡ ◁ 19a 19c ▷

```
val ; is RTL.EFFECTS        -- type will be effect * effect -> effect
val andthen is ;
infixl 0 ;
infixl ~1 andthen
```

4.2 Booleans

`true`, `false`, and `RTL.NOT` are the Boolean operators in λ-RTL's initial basis. We prefer to use `not` to stand for Boolean negation.

19c ⟨*RTL basics* 19a⟩+≡ ◁ 19b 20c ▷

```
val not is RTL.NOT
```

For writing other Boolean functions λ-RTL has only `if-then-else-fi`, which λ-RTL rewrites into suitable guards. We would like to have connectives `andalso` and `orelse`, but introducing them presents a choice about level of detail: should they be introduced as new RTL operators, whose semantics must be defined outside of λ-RTL, or should they be defined in terms of the existing λ-RTL primitives? We would like to try both alternatives, because they provide different levels of detail in the generated RTLs.

Eventually, λ-RTL will have mechanisms to support multiple levels of abstraction, probably by using some sort of parameterized modules. For now, we resort to some literate-programming tricks that are equivalent to conditional compilation. `noweb` can extract this code in two versions, designated **simple** and **full**. Code chunks tagged ⟦**simple**⟧ and ⟦**full**⟧ appear only in the corresponding versions, and untagged code chunks appear in both versions.

In the simple version, the boolean connectives are left abstract. In the full version, we give them their standard definitions. The simple version has the advantage that the RTLs are smaller and easier to read, but the disadvantage that the semantics have to be specified elsewhere. In the full version, these connectives are expanded to primitives, so they automatically get their proper semantics.

20a ⟨*RTL basics* ⟦**simple**⟧ 20a⟩≡ 20e ▷
```
  rtlop [andalso orelse]  -- Boolean connectives
```
Conditional definition.

20b ⟨*RTL basics* ⟦**full**⟧ 20b⟩≡ 20f ▷
```
  fun andalso (p, q) is if p then q    else false fi
  fun orelse  (p, q) is if p then true else q     fi
```
Conditional definition.

No matter what their definitions, we want these connectives to be left-associative infix operators.

20c ⟨*RTL basics* 19a⟩+≡ ◁ 19c 20d ▷
```
  infixl 3 orelse
  infixl 4 andalso
```

In the long run, equality and inequality will probably have to have some built-in semantics, but for now, we leave them abstract.

20d ⟨*RTL basics* 19a⟩+≡ ◁ 20c 20g ▷
```
  rtlop    [= <>]
  infixn 5 [= <>]
```

We have chosen to make inequality a primitive RTL operator, although we could have defined it in terms of equality by writing `val <> is \(x,y). not (x = y)`.

We can convert between booleans and bits. Again we have the choice of making things abstract or concrete.

20e ⟨*RTL basics* ⟦**simple**⟧ 20a⟩+≡ ◁ 20a 21f ▷
```
  rtlop bit  -- type will be bool -> #1 bits   -- turn boolean into bit
  rtlop bool -- type will be #1 bits -> bool   -- turn bit into boolean
```

20f ⟨*RTL basics* ⟦**full**⟧ 20b⟩+≡ ◁ 20b 22a ▷
```
  fun bit  p is if p then 1 else 0 fi
  fun bool n is n <> 0
```

4.3 Integer arithmetic and comparisons

These operations represent integer arithmetic performed on the bit-vector representation. Multiplication and division come in two flavors. One interprets its operands as unsigned integers; the other interprets its operands as two's-complement signed integers. There are two sets of signed division operators because division may truncate towards $-\infty$ (`divs` and `mods`) or towards 0 (`quots` and `rems`). There is, of course, no such distinction for unsigned division.

20g ⟨*RTL basics* 19a⟩+≡ ◁ 20d 21a ▷
```
  rtlop [add subtract neg muls mulu divu modu divs mods quots rems < <= > >=]
          -- standard arithmetic operators of several different types
```

The intended semantics of `add` and `subtract` is "add with carry" and "subtract with borrow." We define shorthands for the common case where carry (or borrow) in is 0 and carry out is uninteresting:

21a ⟨*RTL basics* 19a⟩+≡ ◁20g 21b▷

```
fun +(n, m) is (add(n, m, 0)).result      -- add returns {result, carry}
fun -(n, m) is (subtract(n, m, 0)).result -- subtract returns {result, borrow}
```

The arithmetic operators obey the usual rules of precedence.

21b ⟨*RTL basics* 19a⟩+≡ ◁21a 21c▷

```
infixl 7 [divs mods quots rems divu modu quotu remu]
infixl 6 [+ -]
infixn 5 [< <= > >=]
```

Comparisons are left abstract instead of being defined in terms of `<` and `=`.

We can sign-extend or zero-extend integers. These operators should be polymorphic in the sizes of the arguments and results.

21c ⟨*RTL basics* 19a⟩+≡ ◁21b 21d▷

```
rtlop [sx zx] -- type will be #n bits -> #m bits
```

4.4 Logical operators

Most processors offer a variety of bitwise operations on bit vectors, including those shown here.

21d ⟨*RTL basics* 19a⟩+≡ ◁21c 21e▷

```
rtlop [and or xor] -- #n bits * #n bits -> #n bits
rtlop com          -- #n bits -> #n bits -- ‘complement’ because ‘not’ is boolean
rtlop bitInsert    -- {value : #w bits, lsb : #k bits} -> #n bits -> #w bits
rtlop bitExtract   -- {value : #w bits, lsb : #k bits} -> #n bits
```

`bitInsert` and `bitExtract` insert into or extract from a `w`-bit ("wide") `value`. The thing inserted or extracted is `n` bits ("narrow"), and the widths of both wide and narrow values are normally inferred by the type system. (We may change the types planned for `bitInsert` and `bitExtract`.)

There are instructions that manipulate bit fields whose widths aren't known statically. These computations can't be expressed with `bitInsert` and `bitExtract`, because we can't assign a type (width) to the results. But we can define `bitTransfer` as a combination extraction and insertion, and in fact, this is the way such instructions must behave. Even an instruction that extracts k bits from a 32-bit word, where k is not known until run time, must still insert the result into some storage location of known size—typically a 32-bit word of all zeros.

21e ⟨*RTL basics* 19a⟩+≡ ◁21d 22b▷

```
rtlop bitTransfer
-- {src: {value:#w bits, lsb:#k bits}, dst:{value:#w bits, lsb:#k bits}, width:#j bits} -> #w bits
-- extract width bits from src at lsb, insert into dst at lsb
```

Most processors have shift instructions. We could make the shift operations primitive RTL operators, or we could define them in terms of bit transfer.

21f ⟨*RTL basics* ⟦**simple**⟧ 20a⟩+≡ ◁20e

```
rtlop [shl shrl shra]
```

22a ⟨*RTL basics* ⟦**full**⟧ 20b⟩+≡ ◁20f
```
  fun shl (w, n, k) is
    bitTransfer {src is {value is n, lsb is 0},
                 dst is {value is 0, lsb is k},
                 width is w - k}
  fun shrl (w, n, k) is
    bitTransfer {src is {value is n, lsb is k},
                 dst is {value is 0, lsb is 0},
                 width is w - k}
  fun shra (w, n, k) is
    bitTransfer {src is {value is n, lsb is k},
                 dst is {value is sx n@[1 bit at w-1], lsb is 0},
                 width is w - k}
```

4.5 Byte order

The current implementation of λ-RTL does not support aggregations, but we would like to be able to use them in our sample specifications, so we define both big- and little-endian aggregations to be the identity function.

22b ⟨*RTL basics* 19a⟩+≡ ◁21e 22c▷
```
  fun bigEndian    size is \x.x
  fun littleEndian size is \x.x
```

4.6 Undefined effects

The current implementation of λ-RTL does not support the kill effects. In a later implementation, we plan to use the question mark to stand for an "undefined" value. All λ-RTL functions and RTL operators will be strict in this value, and `RTL.STORE(loc, ?)` will become `RTL.KILL loc`. We would like to use the question mark in our example descriptions, even though λ-RTL doesn't yet have the machinery to give it its eventual meaning, so we introduce it as a nullary RTL operator.

22c ⟨*RTL basics* 19a⟩+≡ ◁22b 22d▷
```
  rtlop ?    -- type will be #k bits
```

4.7 Floating-point arithmetic

Floating-point operations need to come in different families, so for example we could distinguish IEEE 754 floating point from VAX floating point. Right now the name space of RTL operators is flat, so such distinctions are impossible. It nevertheless seems reasonable to put these operators in a structure. Our treatment of floating-point operators is not complete; we only sketch some possibilities.

22d ⟨*RTL basics* 19a⟩+≡ ◁22c
```
  structure IEEE754 is struct
    ⟨IEEE 754 floating point 23a⟩
  end
```

We use operators surrounded by slashes to designate floating-point arithmetic operations.

23a ⟨*IEEE 754 floating point* 23a⟩≡ (22d) 23b▹
```
rtlop [/+/ /-/ /*/ /]  -- basic floating-point arithmetic
rtlop [fabs fsqrt]     -- special floating-point operators
```

All these operators will eventually be parameterized by the size (type) of their operands.

When we get a type system, the conversions will be polymorphic in two widths: the argument and result widths. We may need a whole family of float-to-integer operators in order to account properly for rounding modes.

23b ⟨*IEEE 754 floating point* 23a⟩+≡ (22d) ◃23a 23c▹
```
rtlop [i2f f2i]        -- conversions to/from integer.
```

There are the special values ± 0 and $\pm\infty$.

23c ⟨*IEEE 754 floating point* 23a⟩+≡ (22d) ◃23b 23d▹
```
rtlop [pzero mzero]   -- plus and minus zero     -- type will be #n bits
rtlop [pinf  minf]    -- plus and minus infinity -- type will be #n bits
```

There are NaNs.

23d ⟨*IEEE 754 floating point* 23a⟩+≡ (22d) ◃23c 23e▹
```
rtlop NaN  -- type will be function from significand to value
```

There are four rounding modes required by the standard.

23e ⟨*IEEE 754 floating point* 23a⟩+≡ (22d) ◃23d
```
structure Round is struct
  rtlop [nearest zero down up] -- what to round toward
end
```

Chapter 5

Describing the SPARC, Version 8

This chapter illustrates λ-RTL by specifying a handful of SPARC instructions. Load and store instructions illustrate the basic techniques used to move data of different sizes. Logical instructions and add instructions show simple groups of computational instructions; each group offers a slightly different treatment of condition codes. Specifications of save and restore instructions illustrate one of several possible treatments of register windows. In this case we have not yet achieved our goal of separating hardware behavior from software conventions; our model of register windows describes the abstraction that is presented by the combination of hardware, calling convention, and operating system.

5.1 Storage spaces

Storage locations manipulated by SPARC instructions include memory and several kinds of registers. We have little to say about memory other than that the machine is byte-addressed and uses big-endian byte order.

24 ⟨*SPARC basics* 24⟩≡ (35c) 25c ▷

```
storage
  'm' is    cells of  8 bits called "memory" aggregate using bigEndian
val bigE is bigEndian -- useful abbreviation
```

We introduce the abbreviation `bigE` for cases where we want to write aggregation explicitly.

Integer Registers

We normally expect that a machine's registers can be modeled as a simple collection of mutable cells, but the SPARC has two architectural features that don't fit this model: an immutable register and register windows. Register 0 is immutable; fetches from register 0 always return zero, and stores into register 0 have no effect. Registers 1–7 are ordinary mutable cells, but registers 8–31 are aliases for locations in a collection of *register windows*. The register windows are overlapping sets of 16 registers; an implementation may provide from 2 to 32 sets. Bits 0–4 of the processor state register constitute a *current window pointer* (CWP). When an instruction refers to a register numbered 8–31, the processor uses the CWP and the register number to identify a window of 16 registers and a register in that window. The SPARC `save` and `restore` instructions manipulate the CWP, as do traps and returns from traps.

This low-level model would be easy to describe using λ-RTL, but it is not the model used by most compiler writers. Compiler writers seldom need to use register 0 explicitly, because SPARC assembly languages provide "synthetic" instructions that use register 0 as needed. Instead of using the detailed semantics of register windows, compiler writers adhere to the SPARC calling convention, which (with some help from the operating system) gives the illusion of an infinite collection of register windows, and which allocates one register window to each activation of each procedure.[1] The compiler must reserve space on the stack for use as backing store, and it must use `save` and `restore` in procedure prologs and epilogs.

λ-RTL is not biased toward any particular model of register windows, and in fact it could be used to specify different models which might be useful in different situations. Eventually, λ-RTL will have a modules system that will enable us to create different models, at different levels of abstraction, of such things as SPARC register windows. For this document, we have chosen a fairly abstract model that is convenient for compiler writers. We hide most of the low-level hardware behavior, and we describe registers (except register 0) as a simple collection of mutable cells. Section 5.2 shows the details of the model that are needed to specify the effects of the `save` and `restore` instructions.

Dealing with register 0 is fairly simple; there are only two reasonable models. One is the full semantics; the other is a model which specifies only that register 0 is immutable. The simpler model suffices for some compilers, and the resulting RTLs are easier to understamd. As in Chapter 4, we use ⟦**simple**⟧ and ⟦**full**⟧ to show the two alternatives.

25a ⟨*SPARC basics* ⟦**simple**⟧ 25a⟩≡ 26c▹
```
structure Reg is struct
  fun fetch n is $r[n]
  fun store (n, v) is n <> 0 --> $r[n] <-- v
  ⟨alias functions for in, out, local, and global registers 29a⟩
end
```
Conditional definition.

We make register 0 immutable by ensuring that attempts to store into it have no effect. The full semantics also shows that fetches return 0.

25b ⟨*SPARC basics* ⟦**full**⟧ 25b⟩≡ 26b▹
```
structure Reg is struct
  fun fetch n is if n = 0 then 0 else $r[n] fi
  fun store (n, v) is n <> 0 --> $r[n] <-- v
  ⟨alias functions for in, out, local, and global registers 29a⟩
end
```
Conditional definition.

In this straightforward model of registers, we use these special fetch and store methods, and we also permit registers to be aggregated into pairs, so we can describe instructions like `ldd`.

25c ⟨*SPARC basics* 24⟩+≡ (35c) ◃24 26a▹
```
storage
  'r' is 32 cells of 32 bits called "registers"
            store using Reg.store
            fetch using Reg.fetch
            aggregate using bigEndian
```

[1] "Optimized leaf procedures" may use their caller's register window.

Integer Unit control/status registers

This specification gives a bare minimum of information about special registers. It emphasizes condition codes.

26a ⟨*SPARC basics* 24⟩+≡ (35c) ◁25c
```
  storage 'i' is 6 cells of 32 bits called "IU control/status registers"
  locations
    [PSR WIM TBR Y PC nPC] is $i[[0..5]]
```
Properly speaking, the integer condition codes are part of the processor state register (PSR).

26b ⟨*SPARC basics* ⟦**full**⟧ 25b⟩+≡ ◁25b
```
  structure icc is struct
    locations [N Z V C] is PSR@[1 bit at [23 22 21 20]]
  end
```
Because references to bits 20–23 of the PSR can be hard to understand, we would like to refer to the condition codes more abstractly. Eventually, the λ-RTL modules system will provide for such abstractions, but for now we resort to dirty tricks. We create a nullary RTL operator for each bit, then bind the names to the locations in an imaginary - space. In the ⟦**simple**⟧ version, references to, e.g., the N bit come out as `$-[N]` instead of `$i[0]@[23]`.

26c ⟨*SPARC basics* ⟦**simple**⟧ 25a⟩+≡ ◁25a
```
  structure icc is struct
    storage '-' alias bogus is
      cells of 32 bits called "bogus space for abstract locations"

    rtlop [N Z V C]
    locations [N Z V C] is $bogus[[N Z V C]]
  end
```

5.2 SPARC instructions

SPARC addresses and operands

The SPARC has a simple structure for effective addresses and operands. As far as the hardware is concerned, there are only two addressing modes, depending on whether the immediate bit is used. We've chosen to let the default attribute of the addressing modes be the address, not the location in memory denoted by that address. The address is the value of register `rs1`, plus either the value of register `rs2` or the result of sign-extending a 13-bit immediate value.

26d ⟨*address modes and operands* 26d⟩≡ (35c) 26e▷
```
  default attribute of
    indexA(rs1, rs2)     is  $r[rs1] + $r[rs2]
    dispA (rs1, simm13)  is  $r[rs1] + sx simm13
```
The `rmode` and `imode` constructors produce the values used in operands of type `reg_or_imm`.

26e ⟨*address modes and operands* 26d⟩+≡ (35c) ◁26d
```
  default attribute of
    rmode (rs2)     is  $r[rs2]
    imode (simm13)  is  sx simm13
```

Load instructions

The SPARC architecture provides for not one but 256 possible address spaces. By default, load instructions use space 0x0A in user mode and space 0x0B in supervisor mode. The mode is determined by the value of the S bit in the processor state register. Values from other address spaces may be obtained by using the "load from alternate space" instructions, but these instructions are privileged. We omit all this complexity from our description, treating the machine as if it were always in user mode. This omission is partly for simplicity, but partly because λ-RTL does not deal well with numbered collections of storage spaces.

The specifications of the load-integer instructions give us our first opportunity to factor λ-RTL descriptions. On the left-hand side, the phrases `[s u]` and `[b h]` are like **for** loops, and the carets (`^`) join parts of constructor names, so the opcode on the left-hand side expands to the list of constructors `ldsb`, `ldsh`, `ldub`, and `lduh`. Corresponding to `[s u]` on the right-hand side is the "expression group" `[sx zx]`. `sx` and `zx` were introduced in Section 4.3 to stand for sign extension and zero extension, respectively. Corresponding to `[b h]` is the expression group `[#8 #16]`. The `#` sign introduces an integer constant representing a width, and the effect here is to produce big-endian aggregations of 8 and 16 bits. Combining the two groups specifies four instructions—signed and unsigned versions of byte and halfword loads.

27a ⟨*instruction defaults* 27a⟩≡ (35c) 27b ▷
```
default attribute of
  ld^[s u]^[b h] (address, rd)  is  $r[rd] <-- [sx zx] (bigE [#8 #16] $m[address])
```

This double factoring works smoothly because the two 2-groups are in the same order on the left- and right-hand sides.

We use another group to define load and load-double instructions, which don't require sign extension or zero extension. For `ldd`, two 32-bit registers are implicitly aggregated to hold a 64-bit value.

27b ⟨*instruction defaults* 27a⟩+≡ (35c) ◁ 27a 28a ▷
```
ld^["" d]      (address, rd)  is  $r[rd] <-- bigE [#32 #64] $m[address]
```

The chunks named ⟨*instruction defaults* 27a⟩ define the default attributes of instructions. We have chosen to use the default attributes to specify the "main effect" of these instructions, but there are circumstances in which the processor traps instead of executing this main effect. Traps are not relevant to all specifications, so we have chosen to give the trap semantics separately, by binding them to an attribute named `trap`. Again, with a proper modules system, trap semantics could be omitted from some specifications.

All loads except byte loads could trap if the `address` is improperly aligned. The load-double also traps if `rd` is an odd-numbered register.

27c ⟨*trap semantics* 27c⟩≡ (35c) 28b ▷
```
attribute trap of
  ld^["" sh uh] (address, rd)  is  alignTrap(address, [4 2 2])
  ldd(address, rd) is alignTrap(address, 8); rd@[0] <> 0 --> trap(illegal_instruction)
```

27d ⟨*SPARC utilities* 27d⟩≡ (35c) 29b ▷
```
fun alignTrap (address, k)  is  address modu k <> 0 --> trap(mem_address_not_aligned)
```

The interaction of the trap semantics with the main semantics is not specified in λ-RTL.

The instructions that load floating-point registers or coprocessor registers don't illustrate anything new, so we have omitted them from this example specification.

Store instructions

The store-halfword and store-byte instructions use bit extraction to show what part of the register is stored.

28a ⟨*instruction defaults* 27a⟩+≡ (35c) ◁27b 28c▷
```
  sth       (rd, address)  is  $m[address] <-- $r[rd]@[16 bits at 0]
  stb       (rd, address)  is  $m[address] <-- $r[rd]@[ 8 bits at 0]
  st^["" d] (rd, address)  is  $m[address] <-- bigE [#32 #64] $r[rd]
```
The trap semantics for store instructions is the same as for load instructions. A more aggressively factored specification might merge the trap semantics of the two groups.

28b ⟨*trap semantics* 27c⟩+≡ (35c) ◁27c 28d▷
```
  st^["" h] (address, rd)  is  alignTrap(address, [4 2])
  std(rd, address) is rd@[0] <> 0 --> trap(illegal_instruction); alignTrap(address, 8)
```

Swapping load-store instructions

Because the semantics of a register-transfer list is that of simultaneous assignment, we can specify swapping instructions without resorting to temporaries.

28c ⟨*instruction defaults* 27a⟩+≡ (35c) ◁28a 30c▷
```
  ldstub(address, rd) is $r[rd] <-- zx $m[address]; $m[address] <-- 0xff
  swap  (address, rd) is $r[rd] <-- bigE #32 $m[address]; bigE #32 $m[address] <-- $r[rd]
```
We plan some experiments to determine under what circumstances we want not to require explicit aggregation.

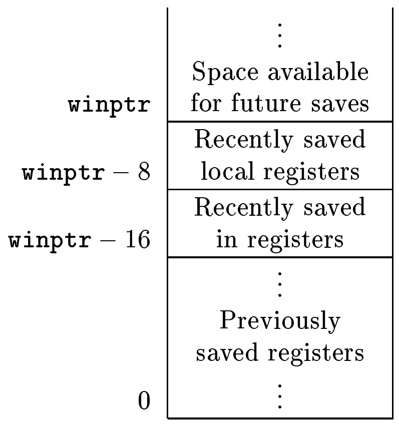
Trap semantics for `swap` is like that for load and store.

28d ⟨*trap semantics* 27c⟩+≡ (35c) ◁28b 35b▷
```
  attribute trap of swap(address, rd) is alignTrap(address, 4)
```

Save and restore instructions

To describe the effects of `save` and `restore`, we introduce a fictional storage space `w` to stand for the locations where registers are saved. Some of these locations correspond to hardware register windows; others correspond to reserved locations on the stack.

28e ⟨*register windows* 28e⟩≡ (35c)
```
  storage
    'w'          is    cells of 32 bits called "register windows"
    '.' alias dot is  1 cell  of 32 bits called "register-window pointer"
  locations
    winptr is $dot[0]
```
The location called `winptr` is analogous to the current window pointer (CWP), but they are not identical. At any moment during the execution of a program, cells `$w[0..winptr-1]` hold the contents of registers that have been saved with previous `save` instructions. Other cells in `w` space have undefined contents. Figure 5.1 shows the layout of `w` space.



winptr
Space available for future saves
Recently saved local registers
winptr − 8
Recently saved in registers
winptr − 16
Previously saved registers
0

Figure 5.1: Layout of `w` space used to model register windows

The registers have aliases, which are shown in Table 4-1 in the SPARC manual (SPARC 1992). To make it easier to specify the save and restore instructions correctly, we create functions that implement these aliases.

29a ⟨*alias functions for* `in`, `out`, `local`, *and* `global` *registers* 29a⟩≡ (25)
```
fun in'    n is $r[n+24]
fun local' n is $r[n+16]
fun out    n is $r[n+8]
fun global n is $r[n]
```
We have to use `in'` and `local'` because `in` and `local` are reserved words in λ-RTL.

The true behavior of a `save` instruction is to decrement CWP and to trap if the new value is invalid (according to the Window Invalid Mask register). The normal trap handler provides the illusion of infinite register windows, by saving register windows on the stack and by adjusting the WIM register. We model this illusion as movement of out registers to in registers and movement of in and local registers to the `w` space.

Global registers are unchanged by `save`; Figure 5.1 helps clarify what happens to the others.

29b ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁27d 29c▷
```
fun saveOut   n is Reg.in' n <-- Reg.out n
fun saveLocal n is $w[winptr+8+n] <-- Reg.local' n
fun saveIn    n is $w[winptr+n]   <-- Reg.in' n
```
We use functions from the built-in `Vector` structure to create `do8` such that `do8 f` applies `f` to the integers 0 to 7 and returns the simultaneous composition of the resulting effects. If this idiom proves useful, we will provide syntactic sugar for it.

29c ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁29b 30b▷
```
fun do8 f is Vector.foldr (\(n, effects). f n ; effects) RTL.SKIP (Vector.spanning 0 7)
```
We would like simply to write

29d ⟨*incorrect instruction specifications* 29d⟩≡ 30a▷
```
save (rs1, reg_or_imm, rd) is (
  do8 saveOut; do8 saveLocal; do8 saveIn; winptr <-- winptr + 16;
  $r[rd] <-- $r[rs1] + reg_or_imm)
```
Conditional definition.

but the problem with this specification is that it is ill-formed whenever `rd` is an “in” register—the explicit assignment to `rd` conflicts with the assignment created by `do8 saveOut`, and there is no way to say which one takes priority. We do not want to write

30a ⟨*incorrect instruction specifications* 29d⟩+≡ ◁29d

```
save (rs1, reg_or_imm, rd) is (
  do8 saveOut; do8 saveLocal; do8 saveIn; winptr <-- winptr + 16
      andthen
  $r[rd] <-- $r[rs1] + reg_or_imm)
```

because the right-hand side of the assignment would use values from the new register window, not from the old one. One possible, if unpleasant, way out of this dilemna is to guard the assignments created by `saveOut`, so that register `rd` is not touched:

30b ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁29c 30d▷

```
fun saveRegs rd is
  let fun saveOut   n is rd <> n+24 --> Reg.in' n <-- Reg.out n
      fun saveLocal n is $w[winptr+8+n] <-- Reg.local' n
      fun saveIn    n is $w[winptr+n]   <-- Reg.in' n
  in  do8 saveOut; do8 saveLocal; do8 saveIn; winptr <-- winptr + 16
  end
```

30c ⟨*instruction defaults* 27a⟩+≡ (35c) ◁28c 30e▷

```
save (rs1, reg_or_imm, rd) is saveRegs rd; $r[rd] <-- $r[rs1] + reg_or_imm
```

We use similar tactics to specify `restore`.

30d ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁30b 30f▷

```
fun restoreRegs rd is
  let fun restoreOut   n is rd <> n+8  --> Reg.out    n <-- Reg.in' n
      fun restoreLocal n is rd <> n+16 --> Reg.local' n <-- $w[winptr-8+n]
      fun restoreIn    n is rd <> n+24 --> Reg.in'    n <-- $w[winptr-16+n]
  in  do8 restoreOut; do8 restoreLocal; do8 restoreIn; winptr <-- winptr - 16
  end
```

30e ⟨*instruction defaults* 27a⟩+≡ (35c) ◁30c 31e▷

```
restore (rs1, reg_or_imm, rd) is restoreRegs rd; $r[rd] <-- $r[rs1] + reg_or_imm
```

Logical instructions

The SPARC doesn’t have a bitwise-complement instruction; instead, each logical instruction has a variant that complements its second operand. Using existing RTL operators, we define functions to represent these “logical-complement” operations. We use `com` for bitwise complement; `not` works only on booleans.

30f ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁30d 31a▷

```
fun [andn orn xnor] (a, b)  is  [and or xor](a, com b)
```

The logical instructions include variants that set condition codes and variants that leave condition codes unchanged. Because this is a common pattern among SPARC instructions, we define `leave_cc` and `set_cc` to help specify these two kinds of effects. Instructions that do set condition codes typically set the `N` and `Z` bits according to the value of an integer result. There is no single typical treatment of the `O` and `C` bits, so we require that values for these bits be passed in.

31a ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁30f 31b▷
```
fun set_cc(result, overflow, carry) is
  icc.N <-- bit (result < 0);
  icc.Z <-- bit (result = 0);
  icc.V <-- overflow;
  icc.C <-- carry
fun leave_cc (result, overflow, carry) is do_nothing
```
The logical instructions clear overflow and carry.

31b ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁31a 31c▷
```
fun logical_cc (result) is set_cc(result, 0, 0)
```
The logical instructions are among the SPARC instructions that take the form

$\oplus$ `rs1, reg_or_imm, rd`

where $\oplus$ is a binary operator. We define the function `binary` to get the effect of this common form.

31c ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁31b 31d▷
```
fun binary (operator, rs1, r_o_i, rd) is $r[rd] <-- operator($r[rs1], r_o_i)
```
This function by itself isn't sufficient for variants that set the condition codes. `binary_with_cc` combines the main effect with whatever effects are produced by `special_cc`, a code-setting function that is passed in. If this function is `leave_cc`, the condition codes won't be changed.

31d ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁31c 32b▷
```
fun binary_with_cc (operator, rs1, r_o_i, rd, special_cc) is
  let val result is operator($r[rs1], r_o_i)
  in  $r[rd] <-- result; special_cc result
  end
```
We use factoring and these utility functions to specify all the logical instructions at once.

31e ⟨*instruction defaults* 27a⟩+≡ (35c) ◁30e 32c▷
```
[and or xor andn orn xnor]^[cc ""] (rs1, reg_or_imm, rd) is
  binary_with_cc([and or xor andn orn xnor], rs1, reg_or_imm, rd, [logical_cc leave_cc])
```

Add instructions

Our treatment of the add instructions is similar to our treatment of the logical instructions, except

- These instructions can set the overflow and carry bits, and
- Addition is a *trinary* operator, because there may be "carry in."

Overflow semantics is a bit tricky, so in the ⟦**simple**⟧ version of the specification we leave it abstract.

31f ⟨*SPARC utilities* ⟦**simple**⟧ 31f⟩≡ 33c▷
```
rtlop add_overflows
```
Conditional definition.

32a ⟨*SPARC utilities* ⟦**full**⟧ 32a⟩≡ 34a▷
```
  fun add_overflows (x, y, c) is
    let val {result, carry} is add(x, y, c)
    in  x@[31] = y@[31] andalso x@[31] <> result@[31]
    end
```
Conditional definition.

An add-like operator takes three arguments and returns two results, so we can't used `binary_with_cc`. The main effect always takes one result and stores it in `rd`, plus there may be a secondary effect on the condition codes. This secondary effect requires all the operands.

32b ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁31d 32d▷
```
  fun add_like (operator, rs1, operand2, operand3, rd, special_cc) is
    let val main is (operator($r[rs1], operand2, operand3)).result
    in  $r[rd] <-- main; special_cc ($r[rs1], operand2, operand3)
    end
  fun add_cc (x, y, c) is
    let val {result, carry} is add(x, y, c)
    in  set_cc(result, bit(add_overflows(x, y, c)), carry)
    end
```
These functions suffice to specify the add instructions, but the naming of the instructions makes the factoring a bit tricky. A 4-group on the left-hand side corresponds to two 2-groups on the right-hand side. Groups are evaluated in left-to-right LIFO order, so the rightmost group varies most rapidly, and so for example the `addcc` constructor corresponds to the use of `0` for `operand3` and the use of `add_cc` for `special_cc`.

32c ⟨*instruction defaults* 27a⟩+≡ (35c) ◁31e 33b▷
```
  [add addcc addx addxcc] (rs1, reg_or_imm, rd) is
    add_like(add, rs1, reg_or_imm, [0 (icc.C)], rd, [leave_cc add_cc])
```
One might wish that `icc.C` did not require parentheses here.

Multiply instructions

The multiplication and division instructions treat a general-purpose register and the Y register as a single 64-bit register. The `Reg64` structure shows how to use such a pair to hold a 64-bit value. To put the value in the pair, we put the most significant 32 bits in the Y register and the least significant 32 bits in the general-purpose register. To recover the value, we zero-extend the general-purpose register to 64 bits, then insert the contents of the Y register in place of the most significant 32 bits (which we know to be zeroes).

32d ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁32b 33a▷
```
  structure Reg64 is struct
    fun set (reg, n) is Y <-- n@[32 bits at 32]; $r[reg] <-- n@[32 bits at 0]
    fun get reg      is bitInsert {value is zx $r[reg], lsb is 32} Y
  end
```

The multiply instructions produce a 64-bit result, and they have their own way of setting condition codes, so we define another pair of auxiliary functions.

33a ⟨*SPARC utilities* 27d⟩+≡ (35c) ◁32d

```
fun mul_like (operator, rs1, r_o_i, rd, special_cc) is
  let val result is operator($r[rs1], r_o_i)
  in  Reg64.set(rd, result); special_cc result
  end
fun mul_cc(result) is set_cc(result, ?, ?)
```

An undefined, rather than unspecified, value is the right thing for the V and C bits, because the manual says that "specification of this condition code may change in a future revision to the architecture. Software should not test this condition code."

33b ⟨*instruction defaults* 27a⟩+≡ (35c) ◁32c 34b▷

```
[u s]^mul^["" cc] (rs1, reg_or_imm, rd) is
  mul_like([mulu muls], rs1, reg_or_imm, rd, [leave_cc mul_cc])
```

Branch on integer condition codes

The SPARC processor can branch on any of 16 conditions, each of which is a function of the 4 condition-code bits. Here, we give the tests abstractly, without showing the functions. The `val` declaration of the 16 tests hides the previous declaration of the same identifiers as RTL operators.

33c ⟨*SPARC utilities* ⟦**simple**⟧ 31f⟩+≡ ◁31f

```
structure IccTest is struct
  rtlop [A N NE E G LE GE L GU LEU CC CS POS NEG VC VS]
     -- type is function from integer condition codes to boolean
  val [A N NE E G LE GE L GU LEU CC CS POS NEG VC VS] is
      [A N NE E G LE GE L GU LEU CC CS POS NEG VC VS] icc
     -- functions are applied to icc, and results are boolean
end
```

Here, we define the test conditions as specified in the manual. We let `Z`, `N`, `V`, and `C` stand for the proper bits within the condition code, and we use **`or`**, **`xor`**, and **`not`** as bit operations, not boolean operations. Finally, after all the bit computations, we use the utility function `bool` to turn the result into a boolean, so it can be used as a guard.

34a ⟨*SPARC utilities* ⟦**full**⟧ 32a⟩+≡ ◁32a

```
  structure IccTest is struct
    val [A N NE E G LE GE L GU LEU CC CS POS NEG VC VS] is
      let val [Z N V C] is [(icc.Z) (icc.N) (icc.V) (icc.C)]
          val not is com -- make usage conform to manual
          infixn 3 [or xor]
      in  bool [ 1                      0
                 (not  Z)               Z
                 (not (Z or (N xor V))) (Z or (N xor V))
                 (not       (N xor V))        (N xor V)
                 (not (C or Z))               (C or Z)
                 (not C)                       C
                 (not N)                       N
                 (not V)                       V]
      end
  end
```

The way we specify the branch instructions may look a bit odd, because the selection of a condition, which we are used to thinking of as a postfix notation like x.`A`, becomes an ordinary functions by virtue of being parenthesized, e.g., `(.A)`.[2] These functions are then grouped and applied to `IccTest`. The result of the application is a group of tests, each of which is the appropriate member of the `IccTest` structure. That group is then used to guard an assignment to **`nPC`**, which models a delayed branch.

34b ⟨*instruction defaults* 27a⟩+≡ (35c) ◁33b 35a▷

```
  b^[a n ne e g le ge l gu leu cc cs pos neg vc vs] (target, annul) is
    let val conditionOf is [(.A) (.N) (.NE) (.E) (.G) (.LE) (.GE) (.L)
                           (.GU) (.LEU) (.CC) (.CS) (.POS) (.NEG) (.VC) (.VS)]
    in  conditionOf IccTest --> nPC <-- target
    end
```

To specify when the instruction in the delay slot should be annulled, we define an attribute **`annul_delay`**. Annuling occurs only if the annul bit is set. For the conditional branches, there is an additional requirement that the branch not be taken.

34c ⟨*other attributes of instructions* 34c⟩≡ (35c)

```
  attribute annul_delay of
    b^[_ _ ne e g le ge l gu leu cc cs pos neg vc vs] (target, annul) is
      (annul <> 0 andalso
       not ([(IccTest.A) (IccTest.N) (IccTest.NE) (IccTest.E)
             (IccTest.G) (IccTest.LE) (IccTest.GE) (IccTest.L)
             (IccTest.GU) (IccTest.LEU) (IccTest.CC) (IccTest.CS)
             (IccTest.POS) (IccTest.NEG) (IccTest.VC) (IccTest.VS)]))
    [ba bn] (target, annul) is annul <> 0
```

[2]It's possible that we should drop this notation in favor of the somewhat clearer `(\x.x.A)`.

Call and jump instructions

Both call and jump instructions save the current value of the program counter.

35a ⟨*instruction defaults* 27a⟩+≡ (35c) ◁34b

```
call (target)      is nPC <-- target;  Reg.out 7 <-- PC
jmpl (address, rd) is nPC <-- address; $r[rd]    <-- PC
```

35b ⟨*trap semantics* 27c⟩+≡ (35c) ◁28d

```
attribute trap of jmpl (address, rd) is alignTrap(address, 4)
```

5.3 Putting it all together

This code chunk is expanded into our λ-RTL description of the SPARC. Actually, it is expanded into two different descriptions, depending on whether the ⟦**simple**⟧ or ⟦**full**⟧ alternative is chosen. The `Sparc` structure includes not only the code from this chapter but also the ⟨*RTL basics* (conditional)⟩ from Chapter 4.

35c ⟨*sparc.rtl* 35c⟩≡

```
structure Sparc is struct
  ⟨RTL basics (conditional)⟩
  ⟨SPARC basics 24⟩
  ⟨register windows 28e⟩
  ⟨address modes and operands 26d⟩
  ⟨window dressing for trap semantics 35d⟩
  ⟨SPARC utilities 27d⟩
  default attribute of
    ⟨instruction defaults 27a⟩
  ⟨other attributes of instructions 34c⟩
  attribute trap of
    ⟨trap semantics 27c⟩
end
```

Code written to file `sparc.rtl`.

All that is left to do is to define what we mean by `trap`. Ideally, we would like a λ-RTL abstraction mechanism that would let us define `trap` as "an unspecified function from a 7-bit code to an effect." Without a type system or a modules system, we have to specify a concrete effect. Rather than try to specify the complete semantics of traps, we model a trap as an assignment to a nonexistent location.

35d ⟨*window dressing for trap semantics* 35d⟩≡ (35c) 35e▷

```
storage 't' is 1 cell of 7 bits called "bogus trap value"
fun trap k is $t[0] <-- k
```

These bindings include only the "trap type" of a handful of traps. We have omitted trap priorities.

35e ⟨*window dressing for trap semantics* 35d⟩+≡ (35c) ◁35d

```
val [data_store_error
     illegal_instruction
     mem_address_not_aligned
     tag_overflow
    ] is [0x2b 0x02 0x07 0x0a] -- 7-bit constants to be passed to trap
```

Chapter 6

Describing the Intel Pentium

This chapter shows how we use λ-RTL to describe some aspects of the Pentium instruction set that don't have obvious analogs on the SPARC. Most Pentium instructions come in byte, word (16-bit), and doubleword variants, and these variants treat parts of the Pentium registers as first-class locations. We show how to work with these locations and how to give them their usual names. A single effective address can refer to different parts of a register depending on the opcode with which it is used, so we specify the meanings of effective addresses as triples from which one element is selected depending on context.

By a mixture of opcodes and instruction prefixes, the Pentium offers many variants of each basic operation, so factoring the specification is a concern. We use the logical instructions to illustrate aggressive factoring, different notations for instructions that implement binary operations, and a slightly different treatment of condition codes.

The large-scale structure of the Pentium description resembles that of the SPARC description, except that we add some examples that use infix **and**.

36 ⟨*intel.rtl* 36⟩≡

```
structure Pentium  is  struct
  ⟨RTL basics (conditional)⟩
  ⟨basics 37a⟩
  ⟨effective-address utilities 39e⟩
  ⟨effective addresses 40a⟩
  val [OF CF AF SF ZF PF] is [(Reg.OF) (Reg.CF) (Reg.AF) (Reg.SF) (Reg.ZF) (Reg.PF)]
  ⟨utilities 39d⟩
  local
    infixl 3 and
  in
    default attribute of ⟨instruction defaults using infix and 41b⟩
  end
  default attribute of ⟨instruction defaults 43a⟩
end
```

Code written to file `intel.rtl`.

6.1 Storage spaces

We begin with the basics—registers and memory. The Intel registers require no special fetch or store methods. The machine uses little-endian byte order.

37a ⟨*basics* 37a⟩≡ (36) 37b▷

```
storage
  'r' is 8 cells of 32 bits called "registers"
  'm' is   cells of  8 bits called "memory"
                           aggregate using littleEndian
```

Registers

In the Intel architecture, parts of registers have their own names. Figure 6.1 shows the various parts of the general-purpose registers and the names used to refer to them. We put the definitions for the registers into their own λ-RTL sub-structure so we can more easily manage the name space.

37b ⟨*basics* 37a⟩+≡ (36) ◁37a

```
structure Reg is struct
  ⟨registers 37c⟩
end
```

It is straightforward to define these locations in λ-RTL.

37c ⟨*registers* 37c⟩≡ (37b) 37d▷

```
locations
  [ EAX ECX EDX EBX ESP EBP ESI EDI ]  is  $r[[0..7]]
     --  EAX is $r[0], etc...

  [ AX CX DX BX SP BP SI DI ]  is  $r[[0..7]] @ [0..15]

  [ AL CL DL BL ]  is  [EAX ECX EDX EBX] @ [8 bits at 0]
  [ AH CH DH BH ]  is  [EAX ECX EDX EBX] @ [8 bits at 8]
```

Dealing with the instruction encoding is more complex. Registers are referred to by number, but as can be seen from Figure 6.1, the number doesn't uniquely determine the location. In fact, a register number maps to one of three locations, depending on context. These three arrays specify which register, or part thereof, is denoted by which number in which context:

37d ⟨*registers* 37c⟩+≡ (37b) ◁37c 39a▷

```
val byte  is [. AL, CL, DL, BL, AH, CH, DH, BH .]
val word  is [. AX, CX, DX, BX, SP, BP, SI, DI .]
val dword is [. EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI .]
```

To find out which location is denoted by a register number, we use the number to index the array chosen to represent the proper context.

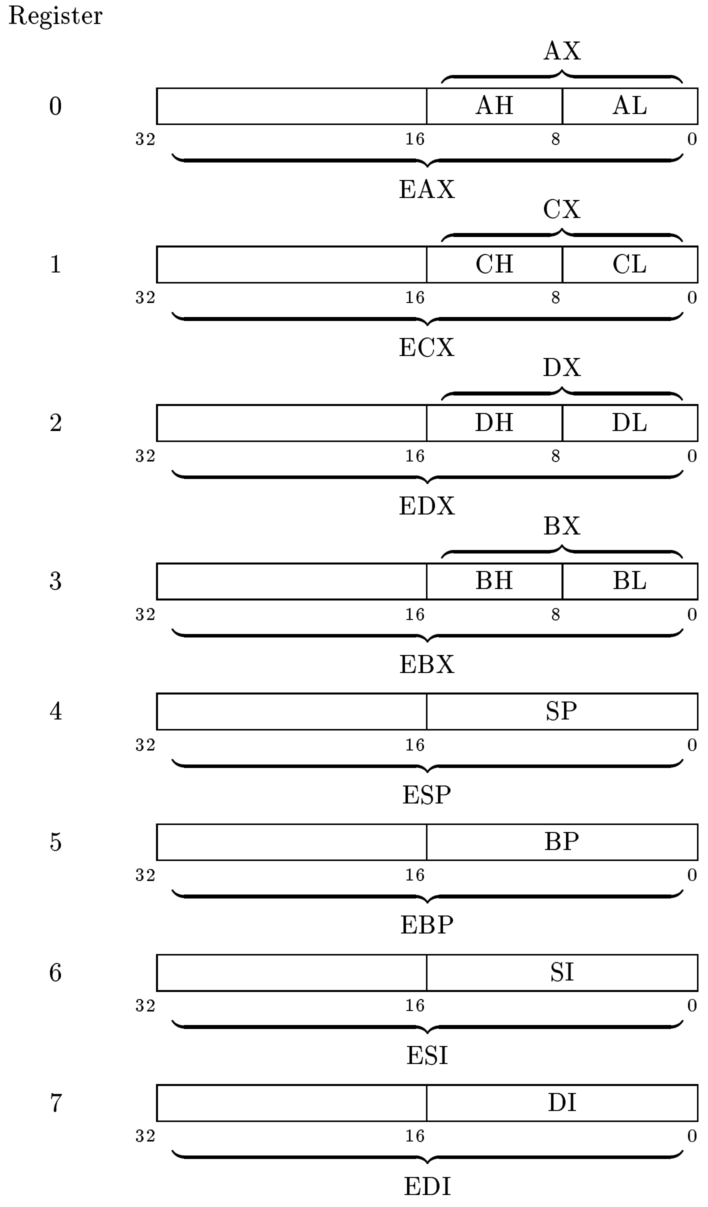


Figure 6.1: Intel Pentium registers

Status and Control Registers

The Pentium has two 32-bit status and control registers, `EFLAGS` and `EIP`. `EIP` is the instruction pointer. `EFLAGS` is a collection of 1-bit status flags and 1- or 2-bit control and system flags.

39a ⟨*registers* 37c⟩+≡ (37b) ◁37d

```
storage
  'c' is 2 cells of 32 bits called "control registers"
locations
  [ EFLAGS EIP ] is $c[[0..1]]
```

What are commonly called condition codes are referred to as status flags in Intel documentation. The Pentium has six.

39b ⟨*registers* ⟦**full**⟧ 39b⟩≡

```
locations
  [ CF PF AF ZF SF OF ]  is  EFLAGS @ [1 bit at [0 2 4 6 7 11]]
  -- carry parity auxiliary-carry zero sign overflow
```

Conditional definition.

In the simplified specification, we pretend that flags occupy their own storage space.

39c ⟨*registers* ⟦**simple**⟧ 39c⟩≡

```
storage 'F' is 6 cells of 1 bit called "abstraction representing flag bits"
locations [ CF PF AF ZF SF OF ]  is $F[[0..5]]
```

Conditional definition.

Most instructions set flags in stylized ways, as shown in Table 3-2 on page 3-14 of the Pentium manual (Intel 1993). In particular, the values of SF, ZF, and PF are determined by simple tests of the result of an operation. We capture this common semantics in the λ-RTL function `set_flags`. `OF`, `AF`, and `CF` are set based on conditions that are different for different kinds of instructions, so we pass their values to `set_flags`. We pass a record, not a tuple, to `set_flags`, so the order in which values are given does not matter.

39d ⟨*utilities* 39d⟩≡ (36) 41a▷

```
rtlop parity -- (number of bits set) mod 2
fun set_flags {result, o, a, c} is
  SF <-- bit (result < 0);
  ZF <-- bit (result = 0);
  PF <-- bit (parity (result@[0..7]) = 0);
  OF <-- o; AF <-- a; CF <-- c
```

Defining the other flags within `EFLAGS` is straightforward, and they are omitted from this example.

Effective addresses

Within an effective address, a register number may denote one of three locations, depending on context. We define functions `regb`, `regw`, and `regd` to be used in these contexts.

39e ⟨*effective-address utilities* 39e⟩≡ (36) 40e▷

```
fun regb n  is  Reg.byte sub n   -- returns an 8-bit location
fun regw n  is  Reg.word sub n   -- returns a 16-bit location
fun regd n  is  Reg.dword sub n  -- returns a 32-bit location
```

The context is usually determined by the opcode, not within the effective address itself. We would nevertheless like an effective address to have a denotation (a default attribute) independent of any context. Our solution to this problem is to let the denotation be a record with three elements—one for each context. The R function creates such a record from a register number. We name the elements b, w, and d to stand for the byte, word, and doubleword locations.

40a ⟨*effective addresses* 40a⟩≡ (36) 40b ▷

```
fun R n        is  {b is regb n, w is regw n, d is regd n}
```

An address in memory also stands for one of three locations, depending on context. (Because a location includes *size* as well as address, we get one byte, two bytes, or four bytes, all at the same address.) The M function does for memory what R does for registers: turn a number into a collection of locations, one of which will be used based on context.

40b ⟨*effective addresses* 40a⟩+≡ (36) ◁ 40a 40c ▷

```
fun M address  is  { b is littleEndian #8  $m[address]
                   , w is littleEndian #16 $m[address]
                   , d is littleEndian #32 $m[address]
                   }
```

Finally, in order to make immediate operands look like register or memory operands, we define an I function, which creates a record that provides the *same* immediate value, no matter what the context.

40c ⟨*effective addresses* 40a⟩+≡ (36) ◁ 40b 40d ▷

```
fun I immed is { b is immed, w is immed, d is immed }
```

Effective addresses denote registers or memory locations.

40d ⟨*effective addresses* 40a⟩+≡ (36) ◁ 40c

```
default attribute of
  Indir   (r)                  is M (regd r)
  Disp8   (d, r)               is M (sx d + regd r)
  Disp32  (d, r)               is M (sx d + regd r)
  Index   (   base, index, ss) is M (regd base +        regd index << ss)
  Index8  (d, base, index, ss) is M (regd base + sx d + regd index << ss)
  Index32 (d, base, index, ss) is M (regd base + sx d + regd index << ss)
  ShortIndex (d, index, ss)    is M (            sx d + regd index << ss)
  Abs32 (a)                    is M a
  Reg (r)                      is R r
```

We define C notation for 32-bit shifts.

40e ⟨*effective-address utilities* 39e⟩+≡ (36) ◁ 39e

```
val [>> <<] is \(n, k).[shrl shl](32, n, k)
infixl 8 [>> <<]
```

Instructions

The Pentium architecture provides more opportunities for factoring than does the SPARC. Most instructions come in three size variants, and each variant uses one of the three contexts. These variants would be tedious to specify repetitiously, so we use λ-RTL's higher-order functions and generators to reduce repetition. A very common pattern of execution is the "two-operand instruction," which we can write

$$l \leftarrow l \oplus r$$

where $\oplus$ is a generic binary operator, l is an effective address dependent on context, and r could be a context-dependent effective address or an immediate value that has been "put in context" by the `I` function.

The λ-RTL function `llr` embodies this pattern of computation. It takes two (curried) arguments: the triple $(l, \oplus, r)$, and a `size` function that determines the context by choosing the `b`, `w`, or `d` element from each operand record.[1]

41a ⟨*utilities* 39d⟩+≡ (36) ◁39d 42a▷
```
  fun llr   (left, op, right) is \size. size left <-- op(size left, size right)
  val [b w d] is [(.b) (.w) (.d)]
```

We also define functions `b`, `w`, and `d` to be used as `size`. The parentheses around `.b`, `.w`, and `.d` are required because record-field selection is normally a postfix operation.

Logical instructions

The Pentium has 42 instructions that implement the three logical operations AND, OR, and XOR. (λ-RTL counts different size variants as different instructions.) All these instructions have side effects on the status flags (condition codes) as well as the $l \leftarrow l \oplus r$ effect. This section shows progressively more aggressive ways to use λ-RTL to describe such large groups of similar instructions.

In λ-RTL, one can declare any identifier to be an infix operator. Infix operators can be convenient when specifying single instructions, as shown in the specification of these three variants of `AND`, which operate on the A register.

41b ⟨*instruction defaults using infix* `and` 41b⟩≡ (36) 41c▷
```
  ANDiAL  (i8)                is  Reg.AL  <-- Reg.AL   and  i8
  ANDiAX  (i16)               is  Reg.AX  <-- Reg.AX   and  i16
  ANDiEAX (i32)               is  Reg.EAX <-- Reg.EAX  and  i32
```

Infix notation is less convenient when describing groups of instructions. Here we use the `llr` function defined above to define some variants that work on two or three sizes of values. The parentheses around `(and)` convert it from an infix operator to an ordinary value that can be passed as a parameter to `llr`.

41c ⟨*instruction defaults using infix* `and` 41b⟩+≡ (36) ◁41b 42b▷
```
  ANDi ^[b w d] (addr, i)     is  llr (addr', (and), I i)      [b w d]
  ANDio^[w d]^b (addr, i)     is  llr (addr', (and), I (sx i)) [  w d]
  ANDmr^[b ow od] (addr, reg) is  llr (addr', (and), R reg)    [b w d]
  ANDrm^[b ow od] (addr, reg) is  llr (R reg, (and), addr')    [b w d]
```

[1]This function won't work quite the way we want under a Hindley-Milner type system, because some uses require the `size` parameter to be polymorphic. The problem can be corrected by making `size`, `b`, `w`, and `d` pairs, and by applying one element to `left` and another to `right`.

The `I` and `R` functions create three-element records that are selected from by the size functions also passed to `llr`. Groups in square brackets are used to define multiple variants at once. Each variant uses either `b`, `w`, or `d` as a `size` function.

We would like the parameter `addr` also to represent a three-element record, since that is the type of the default attribute of an effective address, as defined in Section 6.1. Unfortunately, the current implementation of λ-RTL has no type information, and it cannot infer that `addr` must represent a record containing locations. We therefore use the bogus value `addr'` in place of the address parameter `addr`. When λ-RTL is improved, these definitions will go away, and `addr'` will be replaced with `addr`.

42a ⟨*utilities* 39d⟩+≡ (36) ◁41a 42c▷

```
rtlop stands_for_addr
val addr' is M stands_for_addr
```

The `llr` function helps factor the specification, but the ordinary function-application syntax is not very evocative of what is going on. Here, we use the infix operators `:=`, `<`, and `>` with non-standard meanings, creating a sort of "mixfix" notation that is more suggestive of the semantics:

$$dst \; \texttt{:=} \; l \; \texttt{<}\oplus\texttt{>} \; r.$$

We use the notation first, then define it.

42b ⟨*instruction defaults using infix* `and` 41b⟩+≡ (36) ◁41c

```
ANDiAL  (i8)                is  Reg.AL  <-- Reg.AL   and  i8
ANDiAX  (i16)               is  Reg.AX  <-- Reg.AX   and  i16
ANDiEAX (i32)               is  Reg.EAX <-- Reg.EAX  and  i32
ANDi^[b w d](addr, i)       is  bin (addr' := addr' <(and)> I i)      [b w d]
ANDio^[    w d]^b (addr, i) is  bin (addr' := addr' <(and)> I (sx i)) [  w d]
ANDmr^[b ow od] (addr, reg) is  bin (addr' := addr' <(and)> R reg)    [b w d]
ANDrm^[b ow od] (addr, reg) is  bin (R reg := R reg <(and)> addr')    [b w d]
```

We implement the notation by defining `:=`, `<`, and `>` to build up records, then defining `bin` to create the effect describing the assignment. Since `<` and `>` have existing definitions from Chapter 4, we save those definitions in `lt` and `gt`.

42c ⟨*utilities* 39d⟩+≡ (36) ◁42a 43b▷

```
nonfix [< >]
val [lt gt] is [< >]  -- save these for later
fun < (l, operator) is { l is l, operator is operator }
fun > ({l, operator}, r) is { l is l, operator is operator, r is r }
fun := (dst, {l, operator, r}) is { dst is dst, l is l, operator is operator, r is r}
infixn ~10 <
infixn ~11 >
infixn ~12 :=

fun bin {dst, l, operator, r} is \size . size dst <-- operator (size l, size r)
```

It turns out that OR and XOR have the same structure as AND, so we factor the description to include those instructions as well. Once we do that, there's no point in making the operators infix.

43a ⟨*instruction defaults* 43a⟩≡ (36) 44a▷
```
  [AND OR XOR]^iAL  (i8)               is  Reg.AL  <-- [and or xor] (Reg.AL, i8)
  [AND OR XOR]^iAX  (i16)              is  Reg.AX  <-- [and or xor] (Reg.AX, i16)
  [AND OR XOR]^iEAX (i32)              is  Reg.EAX <-- [and or xor] (Reg.EAX, i32)
  [AND OR XOR]^i^[b w d](addr, i)      is
                        bin (addr' := addr' <[and or xor]> I i)      [b w d]
  [AND OR XOR]^i^[ow od]^b (addr, i)   is
                        bin (addr' := addr' <[and or xor]> I (sx i)) [  w d]
  [AND OR XOR]^mr^[b ow od] (addr, reg) is
                        bin (addr' := addr' <[and or xor]> R reg)    [b w d]
  [AND OR XOR]^rm^[b ow od] (addr, reg) is
                        bin (R reg := R reg <[and or xor]> addr')    [b w d]
```

Now that we have machinery in place, a minor extension will enable us also to specify the effects of these instructions on the status flags (condition codes). Section 4.4.1 of the Pentium manual says "The AND, OR, and XOR instructions clear the OF and CF flags, leave the AF flag undefined, and update the SF, ZF, and PF flags." The function `logical_flags` implements this behavior.

43b ⟨*utilities* 39d⟩+≡ (36) ◁42c 43c▷
```
  fun logical_flags n is set_flags {result is n, o is 0, c is 0, a is ?}
```

In our description of the SPARC, we defined a "main effect" function like `bin` and passed it a function that set the condition code. Here, we use a slightly different approach—we define `bin'` to return both the main effect and the result of the instruction, and we pass `bin'` and its arguments to a function that sets the condition code.

43c ⟨*utilities* 39d⟩+≡ (36) ◁43b 43d▷
```
  fun bin' {dst, l, operator, r} is
    \size .
    let val result is operator (size l, size r)
    in  {result is result, effect is size dst <-- result}
    end
```

`logical` combines the "main effect" and the effects on the flags.

43d ⟨*utilities* 39d⟩+≡ (36) ◁43c 44c▷
```
  val logical is \main.\arg.\size.
    let val {result, effect} is main arg size
    in  effect; logical_flags result
    end
```

Here's the final definition of 42 Pentium instructions, including their effects on the flags. We have re-cast the first three groups so we can apply `logical`. Because `Reg.AL` represents a single location, not a triple, we use `\x.x` (the identity function) as a "size selector." The same trick applies to `Reg.AX` and `Reg.EAX`.

44a ⟨*instruction defaults* 43a⟩+≡ (36) ◁43a 44b▷
```
[AND OR XOR]^iAL  (i8)                 is
                        logical bin' (Reg.AL  := Reg.AL  <[and or xor]> i8)  (\x.x)
[AND OR XOR]^iAX  (i16)                is
                        logical bin' (Reg.AX  := Reg.AX  <[and or xor]> i16) (\x.x)
[AND OR XOR]^iEAX (i32)                is
                        logical bin' (Reg.EAX := Reg.EAX <[and or xor]> i32) (\x.x)
[AND OR XOR]^i^[b w d](addr, i)       is
                        logical bin' (addr' := addr' <[and or xor]> I i)      [b w d]
[AND OR XOR]^i^[ow od]^b (addr, i)    is
                        logical bin' (addr' := addr' <[and or xor]> I (sx i)) [  w d]
[AND OR XOR]^mr^[b ow od] (addr, reg) is
                        logical bin' (addr' := addr' <[and or xor]> R reg)    [b w d]
[AND OR XOR]^rm^[b ow od] (addr, reg) is
                        logical bin' (R reg := R reg <[and or xor]> addr')    [b w d]
```

The remaining logical instruction, NOT, comes in only three variants, and it does not affect the condition codes.

44b ⟨*instruction defaults* 43a⟩+≡ (36) ◁44a
```
NOT^[b ow od] (addr) is unary (com, addr') [b w d]
```

44c ⟨*utilities* 39d⟩+≡ (36) ◁43d
```
fun unary (op, v) is \size . size v <-- op (size v)
```

Chapter 7

Index

The indices are split into two pieces, one for the SPARC and one for the Intel. We'll combine them one day.

7.1 List of chunks

7.2 Identifier index

References

Bailey, Mark W. and Jack W. Davidson. 1995 (January).
A formal model and specification language for procedure calling conventions.
In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA.

———. 1996 (May).
Target-sensitive construction of diagnostic programs for procedure calling sequence generators.
Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 31(5):249–257.

Bala, Vasanth and Norman Rubin. 1995 (November 29–December 1,).
Efficient instruction scheduling using finite state automata.
In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 46–56, Ann Arbor, Michigan.

Benitez, Manuel E. and Jack W. Davidson. 1988 (July).
A portable global optimizer and linker.
Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 23(7):329–338.

Davidson, Jack W. and Christopher W. Fraser. 1980 (April).
The design and application of a retargetable peephole optimizer.
ACM Transactions on Programming Languages and Systems, 2(2):191–202.

Emmelmann, Helmut, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. 1989 (July).
BEG — a generator for efficient back ends.
Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 24(7):227–237.

Fauth, Andreas, Johan Van Praet, and Markus Freericks. 1995 (March).
Describing instruction set processors using nML.
In *The European Design and Test Conference*, pages 503–507.

Fernández, Mary F. 1995 (November).
A Retargetable Optimizing Linker.
PhD thesis, Dept of Computer Science, Princeton University.

Fraser, Christopher W., Robert R. Henry, and Todd A. Proebsting. 1992 (April).
BURG—fast optimal instruction selection and tree parsing.
SIGPLAN Notices, 27(4):68–76.

Griswold, Ralph E. 1982 (October).
The evaluation of expressions in Icon.
ACM Transactions on Programming Languages and Systems, 4(4):563–584.

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

Larus, James R. and Eric Schnarr. 1995 (June).
EEL: machine-independent executable editing.
Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 30(6):291–300.

Larus, James R. 1990 (September).
SPIM S20: A MIPS R2000 simulator.
Technical Report 966, Computer Sciences Department, University of Wisconsin, Madison, WI.

Lipsett, R., C. Schaefer, and C. Ussery. 1993.
VHDL: Hardware Description and Design. 12 edition.
Kluwer Academic Publishers.

Milner, Robin, Mads Tofte, and Robert W. Harper. 1990.
The Definition of Standard ML.
Cambridge, Massachusetts: MIT Press.

Milner, Robin. 1978 (December).
A theory of type polymorphism in programming.
Journal of Computer and System Sciences, 17:348–375.

Morrisett, Greg. 1995 (December).
Compiling with Types.
PhD thesis, Carnegie Mellon.
Published as technical report CMU–CS–95–226.

Proebsting, Todd A. and Christopher W. Fraser. 1994 (January).
Detecting pipeline structural hazards quickly.
In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 280–286, Portland, OR.

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

Ramsey, Norman and David R. Hanson. 1992 (July).
A retargetable debugger.
ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 27(7):22–31.

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

———. 1996 (April).
A simple solver for linear equations containing nonlinear operators.
Software—Practice & Experience, 26(4):467–487.

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

Srivastava, Amitabh and Alan Eustace. 1994 (June).
ATOM: A system for building customized program analysis tools.
Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 29(6):196–205.

Stallman, Richard M. 1992 (February).
Using and Porting GNU CC (Version 2.0).
Free Software Foundation.

Thomas, Donald and Philip Moorby. 1995.
The Verilog Hardware Description Language. 2nd edition.
Norwell, USA: Kluwer Academic Publishers.

Thompson, T. 1996 (February).
An Alpha in PC clothing.
Byte, pages 195–196.

Wang, Daniel C., Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997 (October).
The zephyr abstract syntax description language.
In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages ??–??, Santa Barbara, CA.