Reprinted from the *ACM SIGPLAN Workshop on Languagse, Compilers, and Tools for Embedded Systems (LCTES '98)*

Machine Descriptions to Build Tools for Embedded Systems

Norman Ramsey and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
nr@cs.virginia.edu jwd@cs.virginia.edu

Abstract. Because of poor tools, developing embedded systems can be unnecessarily hard. Machine descriptions based on register-transfer lists (RTLs) have proven useful in building retargetable compilers, but not in building other retargetable tools. Simulators, assemblers, linkers, debuggers, and profilers are built by hand if at all—previous machine descriptions have lacked the detail and precision needed to generate them. This paper presents detailed and precise machine-description techniques that are based on a new formalization of RTLs. Unlike previous notations, these RTLs have a detailed, unambiguous, and machine-independent semantics, which makes them ideal for supporting automatic generation of retargetable tools. The paper also gives examples of λ-RTL, a notation that makes it possible for human beings to read and write RTLs without becoming overwhelmed by machine-dependent detail.

Machine Descriptions for Machine-Level Tools

Developers for embedded systems often work without the benefit of the best software tools. Embedded systems can have unusual architectural features, and new processors can be introduced rapidly. Development is typically done on stock processors, and cross-development can make it hard to get basic compilers, assemblers, linkers, and debuggers, let alone profilers, tracers, test-coverage analyzers, or general code-modification tools. One reason such tools are seldom available is that machine-dependent detail makes it hard to build them.

This paper describes work in progress on Computer Systems Description Languages (CSDL). CSDL descriptions are intended not only to provide precise, formal notations for describing machine-dependent detail, but also to support automatic generation of useful tools. Moreover, CSDL descriptions are intended to be *reusable*, so we can build up a body of descriptions, e.g., of popular embedded processors, that will be useful for building future as well as current tools.

The design goals for CSDL are

– CSDL should support a variety of machine-level tools while remaining independent of any one in particular.

– Descriptions should be composed from simple components. Each component should describe, as much as possible, a single property of the target machine. Such properties might include calling conventions, representations of instructions, semantics of instructions, power consumption, code size, pipeline implementations, memory hierarchy, or other properties.
– An application writer should be able to derive useful tools from partial descriptions. For example, an application writer working entirely at the assembly-language level or above should not have to describe binary representations of instructions.
– Components of descriptions should be reusable. For example, many different tools targeted to the ARM might benefit by reusing a standard formalization of the "ARM Thumb instruction set."

The contributions of this paper are semantic and notational. CSDL uses *register transfers* to specify the semantics of machine instructions. In previous work, the exact meaning of register transfers is known only in the context of a particular machine. By contrast, CSDL gives register transfers a detailed, unambiguous, and machine-independent semantics. The detail will make CSDL useful for building a variety of machine-level tools, because information (e.g., byte order) that is left implicit in other formalisms is made explicit in CSDL. The notational contribution is a metalanguage, called λ-RTL, which makes it possible to write register-transfer semantics without having to write all of the detail explicitly. The *λ-RTL translator* bridges the gap between the concise metalanguage and the fully explicit register transfers. λ-RTL also has a semantic abstraction mechanism that gives the author of a specification some freedom to choose the level of detail at which to specify the effects of particular instructions. This paper describes our detailed form of register transfers, then shows how λ-RTL makes it possible to omit much of the detail from the form of specification that is read and written by people. The paper is illustrated with excerpts from our semantic descriptions of popular microprocessors.

Related Work

Machine descriptions have been successful in building retargetable compilers, but the 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. CSDL separates machine properties from compiler concerns.

Some existing languages for machine description, like VHDL (Lipsett, Schaefer, and Ussery 1993) and Verilog (Thomas and Moorby 1995) do describe only properties of machines, 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.

CSDL and the `nML` description language (Fauth, Praet, and Freericks 1995) address similar goals and use similar techniques, but they differ significantly. `nML` requires explicit attribute equations to write assembly-language syntax and binary representations; the CSDL language SLED uses an implicit syntax for assembly language and a more sophisticated, less error-prone sublanguage for describing binary representations (Ramsey and Fernández 1997). `nML` has no mechanism for abbreviating common idioms, making it harder to specify semantics in detail. The published papers suggest that the register transfers used in `nML` do not carry as much information as the register transfers described in this paper.

LISAS (Cook and Harcourt 1994) is another specification language that includes distinct semantic and syntactic descriptions. It specifies binary representations by mapping sequences of named fields onto sequence of bits, a technique that works well for RISC machines, but is awkward for CISC. The underlying model of instructions used in LISAS is less general and flexible than the model used in CSDL and `nML`. LISAS supports only "instructions" and "addressing modes," and LISAS addressing modes lump together values and side effects. CSDL copes with side effects more cleanly by enabling specification writers to use different attributes for describing values and side effects. LISAS also permits "overlapping register sets," which imply that two apparently different registers can be aliased to the same location. In CSDL, any aliasing is purely notational; the λ-RTL translator eliminates apparent aliasing, making the resulting register transfers easier to analyze.

The Core of CSDL

All languages in the CSDL family have the same view of two core aspects of machines: instructions and state. We chose these aspects based on our study of 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). We saw no single aspect used in descriptions for all of these tools, but we did see that all the descriptions refer either to a machine's instruction set or to its storage locations. For example, the descriptions used by the scheduler and linker refer only to the machine's instructions and the properties thereof. The descriptions 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. Some descriptions, like those used in the optimizer and the executable editor, refer both to instructions and to storage, and in particular, they show how the execution of instructions changes the contents of storage.

Given these observations, we 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

In CSDL, an *instruction set* is a list of instructions together with information about their operands. The model is based on experience with the New Jersey Machine-Code Toolkit (Ramsey and Fernández 1997), which has been used to build several machine-level tools. 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 is applied to operands. Instruction descriptions include the names and types of the operands of each instruction. Operand types include integers of various sizes; it is also possible to introduce new types to define such machine-dependent concepts as effective addresses. Values of these new types are created by applying suitable constructors, as defined in a CSDL description. For example, the SPARC supports two addressing modes,[1] the semantics of which can be specified as follows:

```
default attribute of
  indexA(rs1, rs2)    : Address  is  $r[rs1] + $r[rs2]
  dispA (rs1, simm13) : Address  is  $r[rs1] + sx simm13
```

`indexA` and `dispA` are the names of the constructors, and they create values of type `Address`. Such values denote 32-bit addresses. Values of type `Address` can be used as operands to instructions like store:

```
default attribute of  st (rd, Address)  is  $m[Address] := $r[rd]
```

The semantics shows that the store moves data from register `rd` into memory.

We also use CSDL constructors to describe 9 of the addressing modes used on the Pentium,[2] but because the meanings of the Pentium effective addresses depend on context, the description is more complicated, requiring 35 lines of λ-RTL.

As shown above, we specify attributes of instructions in a compositional style; instructions' attributes are functions of the attributes of their operands. In addition to the unnamed `default` attribute, there may be arbitrarily many named attributes. Attributes might describe not only semantics but also binary representations, assembly-language representations, power consumption, code size, cycle counts, or other costs.

[1] The SPARC assembly language appears to support four addressing modes, but the other two are variations of the ones shown, obtained in the special case when `rs1` is 0. We have chosen not to define constructors for these modes, since the semantics specifies elsewhere that register 0 is always zero.

[2] These are the effective addresses available in 32-bit mode without using the address-prefix byte.

Storage

In CSDL, 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. Each cell contains either a bit vector or the distinguished value $\perp$, which is used to model the results of instructions whose effects are undefined. The number of cells in a storage space may be left unspecified. For example, we specify the general-purpose registers and memory of the Intel Pentium as follows:

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

The Pentium has a register file made up of 32-bit cells and a memory made up of 8-bit cells (bytes). The `aggregate` directive tells the λ-RTL translator the default byte order to use with references to memory.

The state of a machine can be described as the contents of its storage spaces. We use storage spaces to model not only main memory and general-purpose registers, but also special-purpose registers, condition codes, and so on.

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. For example, on the Pentium, `$r[0]` stands for general-purpose register 0, i.e., register EAX.

Combining instructions and storage

Specifications of instructions and storage come straight out of architecture manuals. Manuals list instructions, their operands, and the storage locations that constitute the state of a processor. Most importantly, manuals say what instructions do; i.e., they explain how each instruction affects the state of the processor. We believe that a formal description of this information will enable us to build many different kinds of tools, including control-flow analyzers, code-editing tools like EEL (Larus and Schnarr 1995) and ATOM (Srivastava and Eustace 1994), code improvers in the style of PO (Davidson and Fraser 1980), vpo (Benitez and Davidson 1988), and `gcc` (Stallman 1992), and even emulators like SPIM (Larus 1990) and EASE (Davidson and Whalley 1990).

In λ-RTL, we specify the effect of each instruction as a *register-transfer list* (RTL), which describes a way of modifying storage cells. Like other properties of instructions, the RTL is a synthesized attribute.

Register Transfer Lists

CSDL's register transfer lists designed to be used by tools, not by people. To simplify analysis, we make their form simple, detailed, and unambiguous. We insist that as much information as possible be explicit in the RTL itself. It doesn't matter if individual RTLs grow large, as long as they are composed from simple

```
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)
guarded  = GUARD (exp, effect)
rtl      = RTL   (guarded*)
```

`const`	bit vector
`operator`	function
`aggregation`	bijection
`space`	mutable store

Meanings of unspecified terminal symbols

Fig. 1. ASDL specification of the form of RTLs

parts using only a few rules. This design choice distinguishes CSDL's RTLs from earlier work, which has used smaller RTLs that make implicit assumptions about details like operand sizes and byte order. In CSDL,

- RTLs are represented as trees.
- All operators are fully disambiguated, e.g., as to type and size.
- There is no covert aliasing of locations—locations with different spaces or offsets are always distinct.
- 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 with different cell sizes.

Figure 1 uses the Zephyr Abstract Syntax Description Language (Wang *et al.* 1997) to show the form of RTLs. Working from the bottom, 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**. Effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement; an RTL represents a single change of state. For example, we can specify swap instructions without introducing bogus temporaries. Locations may be single cells or aggregates of consecutive cells within a storage space. Values are computed by expressions without side effects. Eliminating side effects simplifies analysis and transformation. Expressions may be integer constants, fetches from locations, or applications of *RTL operators* to lists of expressions.

Not every effect assigns a value; a *kill* effect stores $\perp$ in a location. Kill effects are needed to specify instructions that change values in an undefined way; for example, Intel (1993) states that "the effect of a logical instruction on the AF flag is undefined."

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

```
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 written in λ-RTL as follows:[3]

```
$r[24] := $m[$r[14]+sx(~12)]
```

because the stack pointer is register 14 and register `i0` is register 24. (Throughout the paper we use the Pascal assignment operator `:=` to write the built-in store operation.) The corresponding RTL is much more verbose, with the sizes of all quantities identified explicitly, as a fully disambiguated tree:

```
                 STORE #32
                /         \
   AGG B #32 #32           FETCH #32
        |                      |
     CELL 'r'              AGG B #8 #32
        |                      |
     CONST #5                CELL 'm'
        |                      |
        24                 APP ADD #32
                          /           \
               FETCH #32               APP SX #13 #32
                   |                         |
             AGG B #32 #32               CONST #13
                   |                         |
               CELL 'r'                     -12
                   |
               CONST #5
                   |
                   14
```

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

Figure 2 shows the meanings and types of 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 `APP ADD`. This node adds the contents of the stack pointer (register 14) to the constant -12. The constant is a 13-bit constant, and applying the `SX` operator sign-extends it to 32 bits, so it can be added to the stack pointer.

In a real machine description, we wouldn't specify just one instance of a load instruction; we would give the semantics of all possible instances:

```
default attribute of  ld (Address, rd)  is  $r[rd] := $m[Address]
```

This specification relies on the semantics of `Address`, which denotes a 32-bit address, as shown above.

We can also use λ-RTL to specify exceptional behaviors of instructions. The following lines specify that load instructions cause traps unless they load from addresses that are aligned on 4-byte boundaries.

```
fun alignTrap (address, k) is address modu k <> 0 --> trap(not_aligned)
attribute trap of ld (address, rd)  is  alignTrap(address, 4)
```

[3] The `~` in `~12` is a unary minus.

`STORE` : $\forall$`#`n.`#`n `loc` $\times$ `#`n `bits` $\rightarrow$ `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 `loc` $\rightarrow$ `#`n `bits`
: For any n, `FETCH #`n takes an n-bit location and returns the n-bit value stored in that location.

`AGG B` : $\forall$`#`n.$\forall$`#`w.`#`n `cells` $\rightarrow$ `#`w `loc`
: For any n and w, `AGG B #`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.)

`CELL 'm'` : `#`32 `bits` $\rightarrow$ `#`8 `cells`
: Given a 32-bit address, `CELL 'm'` returns the 8-bit cell in memory referred to by that address.

`CELL 'r'` : `#`5 `bits` $\rightarrow$ `#`32 `cells`
: Given a 5-bit register number, `CELL 'r'` returns the corresponding 32-bit register (a mutable cell).

`ADD` : $\forall$`#`n.`#`n `bits` $\times$ `#`n `bits` $\rightarrow$ `#`n `bits`
: For any n, `ADD #`n takes two n-bit values and returns their n-bit sum. `ADD` ignores carry and overflow, which can be computed using other RTL operators.

`SX` : $\forall$`#`n.$\forall$`#`w.`#`n `bits` $\rightarrow$ `#`w `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.$\langle$constant$\rangle$ $\rightarrow$ `#`n `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.

Fig. 2. Some RTL operators and their types

Throughout the paper we use the right arrow `-->` to write the built-in `GUARD` operator, which connects a guard to an effect. If the address is properly aligned, the guard on the effect returned by the `alignTrap` function ensures that nothing happens. The `alignTrap` function can be used with other values of `k` to specify the trapping semantics of load-halfword and load-double instructions.

CSDL RTLs are typed. Tools like compilers and analyzers may work directly with RTLs, and because optimizations and other semantics-preserving transformations should also preserve well-typedness, type-checking RTLs can help find bugs (Morrisett 1995). Figure 3 shows the types used in the λ-RTL type system. We have extended Milner's type inference to this system; λ-RTL specifications omit types and widths. Unlike in ML, type inference alone does not always guarantee that terms make sense; in general, there are additional constraints. For example, in the sample tree, the signed integer -12 must be representable using 13 bits, and 32 must be a multiple of both 8 and 32.

`#`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 `AGG B` nodes in the example tree.
`bool`	A Boolean condition.
`effect`	A state transformer (side effect on storage).

Fig. 3. Types in λ-RTL's type system

In contrast to RTLs used in earlier work, CSDL's RTLs have detailed and precise semantics independent of any particular machine. Space limitations prevent us from giving a formal semantics here, but the basic idea should be clear: CSDL `storage` declarations specify the state of a machine as a collection of mutable cells, and each RTL denotes a function from states to states. Figure 1 leaves four elements of RTLs unspecified. `space` is an identifier denoting one of the storage spaces declared with `storage`. `const` must denote a bit vector. The denotations of `operator` and `aggregation` warrant more discussion.

RTL operators, written in Fig. 1 as `operator`, must be interpreted as pure, strict functions on values. In particular, the result of applying an RTL operator cannot depend on processor state, and if an RTL operator is applied to $\perp$, it must produce $\perp$. Within these restrictions, users may introduce any RTL operators that seem useful—this abstraction mechanism gives users the ability to say that *something* specific happens, without saying exactly *what*. For example, the rules for determining when a SPARC signed divide instruction overflows are both complicated and implementation-dependent. Rather than attempt to write them using primitives like remainder, absolute value, etc., we might simply introduce a new RTL operator:

```
rtlop sparc_sdiv_overflow : #64 bits * #32 bits -> #1 bits
```

This operator accepts a 64-bit dividend and a 32-bit divisor, and it produces a 1-bit value, which is stored in the V bit by the SPARC SDIVcc instruction.

Most users won't define new RTL operators; they will use the 57 operators defined in our basic RTL library (Ramsey and Davidson 1997). This library includes integer arithmetic and comparison, bitwise operations, and IEEE floating-point operations and rounding modes.

RTL aggregations, written in Fig. 1 as `aggregation`, specify byte order. For example, in the sample tree, the `AGG B #8 #32` between `FETCH` and `CELL 'm'` specifies that the machine builds a 32-bit word by aggregating four 8-bit bytes in big-endian order. In general, 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 `cells` $\rightarrow$ `#`w `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 are built into λ-RTL, as is an "identity aggregation" that is defined only when $w = n$. We imagine that users could define other aggregations by giving systems of equations, e.g., using the bit-slicing operators of Ramsey (1996).

The precise, machine-independent semantics of RTLs will simplify construction of many useful software tools. A processor simulator is one such tool that is useful for embedded-system development and for architectural research. For development, a simulator allows software to be written, tested, and debugged in a mature programming environment. For research, a simulator gathers detailed measurements of the performance and behavior of the processor. Because CSDL's RTLs are meaningful independent of any machine, they will support the creation of a single interpreter capable of simulating any program expressed in RTL form. Previous versions of register-transfer lists required some machine-dependent code for each target machine of interest.

Precise, simple RTLs also make it easier to build tools that analyze RTL programs. For example, access to the mutable state of a machine is available only through the built-in fetch and store operations, so we can easily tell what state is changed by an RTL and how that change depends on the previous state. For embedded and real-time systems, we are interested in developing retargetable tools that analyze RTL programs to determine upper bounds on execution speed, power consumption, and space requirements. Such tools need detail about memory accesses and about the sizes of constants. For mobile code applications, we are interested in performing on-the-fly analysis to detect possible security violations. Of critical importance are the RTLs' lack of aliasing, explicit trap semantics, exposure of fetches, and explicit changes in size or type of data.

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 good for manipulation by tools but not so good for writing specifications.

The λ-RTL metalanguage enables specification writers to attach RTL trees to the CSDL constructors that describe instructions and addressing modes. Tools and tool generators have access to all the details of the full RTLs, but people can write λ-RTL specifications *without* having to write everything explicitly, because λ-RTL operates at a slightly higher level of abstraction. The *λ-RTL translator* bridges the gap.

λ-RTL is a higher-order, strongly typed, polymorphic, pure functional language based on Standard ML (Milner, Tofte, and Harper 1990). λ-RTL descriptions are easier to write than bare RTLs; higher-order functions help eliminate repetition, and the type system infers sizes of operands. Also, λ-RTL relaxes several of the restrictions on the form of RTLs:

- In λ-RTL, one need not 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. It is seldom necessary to write aggregations explicitly.
– One can define RTLs by sequential composition. The λ-RTL translator uses forward substitution to rewrite a sequence of RTLs into a single RTL.

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 programming languages define "lvalue contexts" and "rvalue contexts," and compilers automatically insert fetches in rvalue contexts. λ-RTL works similarly, but instead of using syntax to identify the contexts, it uses types. This technique enables the specification writer to define arbitrary functions that change the state of the machine (e.g., to set condition codes), instead of restricting state change to a few built-in assignment constructs. For example, in our Pentium specification, we have written a function that implements the common pattern

$$l \leftarrow l \oplus r$$

where $\oplus$ is a generic binary operator:

```
fun llr (left, op, right) is left := op(left, right)
```

The λ-RTL translator infers that `left` must refer to a location, and it inserts a fetch above the instance that is passed to `op`.

To define the precise meaning of fetches and stores, including implicit fetches, users can attach fetch and store methods to each storage space. This technique makes it possible to model 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 because it is always 0. The fetch and store methods needed to implement this behavior are simple:

```
storage
  'r' is 32 cells of 32 bits called "registers"
     fetch using \n. if n = 0 then 0 else RTL.TRUE_FETCH $r[n] fi
     store using \(n, v).  n <> 0 --> RTL.TRUE_STORE($r[n], v)
```

The \ represents λ-abstraction, which is a way of defining functions without requiring that they be named. The fetch method accepts a value `n`, representing the offset within storage space `r`, i.e., the register number. If `n` is zero, the result of the fetch is zero, otherwise it is the result of the true fetch (`RTL.TRUE_FETCH`) from that location. (The `if-then-else-fi` construct is present in λ-RTL, but the λ-RTL translator converts it into guarded effects, so tools that analyze RTLs need not deal with conditionals, only guards. Guards are easier to analyze because they appear only at the top level.) The store method accepts a register number `n` and a value `v`, and it returns an effect that stores `v` into register `n`, except when `n` is zero, in which case it does nothing.

Fetch and store methods offer substantial power and flexibility. For example, 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). For our current specification, however, we have chosen a more abstract view of register windows (Ramsey and Davidson 1997, Chap. 5).

Slices. Many machine instructions manipulate fragments of words stored in mutable cells. For example, some machines represent condition codes as individual bits within a program status word, and user instructions may change only those bits. Some 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 is a location that is can be used in 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, which can be applied to locations or to values. Examples include

x`@loc[`k`]`	Bit k of x. (Bit 0 is the least significant bit. In a future version of λ-RTL, it may be possible to change the numbering.)
x`@loc[`k_1`..`k_2`]`	Bits k_1 to k_2 of x, inclusive.
x`@loc[`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 denote locations. (If `@bits` is used instead of `@loc`, x's denote values.) In all cases, the size of the slice is known statically, so its type can be computed automatically. For example, Pentium programmers are accustomed to thinking of AX as a register in its own right, but it is in fact the least significant 16-bit word of register EAX:

```
locations AX is EAX@loc[16 bits at 0]
```

Henceforth `AX` has type `#16 loc`, and it can be used anywhere any other 16-bit location can be used. The illusion that slices are locations is implemented by rewriting fetches and stores so that all slices operate on values 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, which λ-RTL can use to identify the storage space and the 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 one-cell "aggregates."

Boolean constants:	
`true`	Truth
`false`	Falsehood
Functions used to create RTLs:	
`RTL.TRUE_STORE`	Takes location and value, produces effect.
`RTL.STORE`	Invokes a space's store method.
`RTL.TRUE_FETCH`	Fetches value from location.
`RTL.FETCH`	Invokes a space's fetch method.
`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.PAR`	Takes two effects (RTLs) and composes them so they take place simultaneously (list append on list of effects).
`RTL.SEQ`	Takes two effects (RTLs) and composes them so they take place sequentially (forward substitution).
`RTL.AGGB`	Big-endian aggregation.
`RTL.AGGL`	Little-endian aggregation.
Functions on vectors:	
`sub`	Vector subscript.
`Vector.spanning`	`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.

Fig. 4. λ-RTL's initial basis

When little-endian, big-endian, or other aggregations are used, the λ-RTL translator will infer the size of the aggregate. We have tentatively decided to infer aggregates of a size up to the largest cell size in an RTL, so, for example, the translator will infer a 32-bit aggregation in `$r[rd] := $m[Address]`, but a 64-bit aggregation (for a doubleword load) would have to be given explicitly. In the translator's output, all aggregations are explicit. Explicit aggregations are especially useful for building tools like binary translators, which must transform non-native aggregations into byte-swapping.

Writing λ-RTL

λ-RTL provides expressive power with few restrictions. Most of the RTL-specific content of λ-RTL is in the initial basis, i.e., the collection of predefined functions and values. Most of the basis, shown in Fig. 4, is used to create RTLs in the form we have prescribed. The rest contains vector functions that substitute for looping and recursion constructs. (As a way of making sure specifications are well defined, λ-RTL omits looping and recursion.) Loops whose sizes are known in advance can be simulated by using `Vector.foldr` and `Vector.spanning`. Because this style is familiar only to those well versed in functional programming, we expect eventually to provide syntactic sugar for it. We hope that a similar strategy will help specify instructions that are normally considered to have internal control flow, e.g., string-copy instructions.

In addition to using the initial basis, specification writers can introduce new RTL operators. They can also define functions, but the functions are interpreted by the λ-RTL translator and do not appear in the resulting RTLs. Functions are useful only for making specifications more concise and readable. The freedom to define functions and to introduce abstract RTL operators gives us ample scope for experimenting with different styles of description.

Status

Our prototype translator implements λ-RTL as described in this paper, except it omits some size checks, and it does not implement forward substitution. We have used about 300 lines of λ-RTL to describe 160 SPARC instructions, including register windows, control flow, load and store, and all integer and floating-point ALU instructions, including effects on condition codes. The only instructions omitted are some coprocessor instructions, a few privileged load and store instructions, and cache flush. In this description, we used 37 of the 57 RTL operators defined in our machine-dependent library. We also introduced two new, SPARC-specific operators to avoid having to specify exactly how the machine decides when signed and unsigned integer division have overflowed. The Appendix gives some excerpts from this description.

On the Pentium, we have described the registers and their aliases, effective addresses (which may have three different meanings depending on the contexts in which they are used), and the logical instructions. We have also explored ways of using λ-RTL to make descriptions concise; given suitable auxiliary functions, we can specify the semantics of 42 logical instructions, including effects on condition codes, in 7 lines of λ-RTL. The descriptions, as well as more lengthy expositions of RTLs and λ-RTL, are available in a technical report (Ramsey and Davidson 1997).

Acknowledgements

This work has been supported by NSF Grant ASC-9612756 and by DARPA Contract MDA904-97-C-0247. Members of the program committee provided useful suggestions about presentation.

Appendix: Excerpts from the SPARC description

This appendix presents a few more excerpts from our SPARC description. The basic RTL library has been omitted, as have the declarations of the `'r'` and `'m'` spaces, which appear in the text. With these omissions restored, the excerpts (as extracted from the source of this paper) compile with our prototype translator. The complete description, with commentary, is available as part of a technical report (Ramsey and Davidson 1997).

The excerpts begin with more storage spaces and locations.

```
storage 'i' is 6 cells of 32 bits called "control/status registers"
locations [PSR WIM TBR Y PC nPC] is $i[[0..5]]
structure icc is struct
  locations [N Z V C] is PSR@loc[1 bit at [23 22 21 20]]
end
storage 'f' is 32 cells of 32 bits called "floating-point registers"
```

Here are some simple instructions. Control transfer is by assignment to `nPC`.

```
default attribute of
  ldstub(address, rd) is $r[rd] := zx $m[address] | $m[address] := 0xff
  call (target)       is nPC := target  | $r[15] := PC
  jmpl (address, rd)  is nPC := address | $r[rd] := PC
  [sll srl sra] (rs1, reg_or_imm, rd) is
   $r[rd] := [shl shrl shra](32, $r[rs1], reg_or_imm@bits[5 bits at 0])
```

Here is machinery for setting condition codes and for specifying binary operators that may set condition codes.

```
fun set_cc(result, overflow, carry) is
  icc.N := bit (result < 0) | icc.Z := bit (result = 0) |
  icc.V := overflow         | icc.C := carry
fun dont_set_cc _ is RTL.SKIP
fun binary_with_cc (operator, rs1, r_o_i, rd, special_cc) is
  let val result is operator(RTL.FETCH $r[rs1], r_o_i)
  in  $r[rd] := result | special_cc result
  end
```

Here are the logical instructions, which set condition codes. Operators `and`, `or`, and `xor` must be parenthesized because the basic library declares them to be infix. The square brackets are an iterative grouping construct that enables repeated specification in a single expression; this code specifies 4 functions and 12 constructors.

```
fun logical_cc (result) is set_cc(result, 0, 0)
fun [andn orn xnor] (a, b)  is  [(and) (or) (xor)](a, com b)
default attribute of
  [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 dont_set_cc])
```

These functions pair an ordinary register with the `Y` register, to hold a 64-bit value.

```
structure Reg64 is struct
  fun set (reg, n) is
    Y := n@bits[32 bits at 32] | $r[reg] := n@bits[32 bits at 0]
  fun get reg is bitInsert {wide is zx #32 #64 $r[reg], lsb is 32} Y
end
```

Here are signed and unsigned division, which operate on 64-bit values.

```
fun div_like((operator, overflow), rs1, r_o_i, rd, ccguard) is
  let val result is operator(Reg64.get rs1, r_o_i)
      val V      is overflow(Reg64.get rs1, r_o_i)
  in  $r[rd] := result | ccguard --> set_cc (result, V, 0)
  end
rtlop sparc_sdiv_overflow : #64 bits * #32 bits -> #1 bits
rtlop sparc_udiv_overflow : #64 bits * #32 bits -> #1 bits
default attribute of
  [u s]^div^["" cc] (rs1, reg_or_imm, rd) is
    div_like([((divu), sparc_udiv_overflow)
               ((quots), sparc_sdiv_overflow)],
              rs1, reg_or_imm, rd, [false true])
```

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.

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.

Cook, Todd and Ed Harcourt. 1994 (May). A functional specification language for instruction set architectures. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 11–19.

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.

Davidson, Jack W. and David B. Whalley. 1990 (May). Ease: An environment for architecture study and experimentation. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 259–260, Boulder, CO.

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.

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 Jack W. Davidson. 1997 (November). Specifying instructions' semantics using CSDL (preliminary report). Technical Report CS-97-31, Department of Computer Science, University of Virginia. Revised, May 1998.

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

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.

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 213–227, Santa Barbara, CA.