A Functional Specification Language for Instruction Set Architectures

Todd A. Cook
Dept. of Electrical and Computer Eng.
Rutgers University
P.O. Box 909
Piscataway, NJ 08855
tac@ece.rutgers.edu

Ed Harcourt
Dept. of Computer Science
Box 8206
North Carolina State University
Raleigh, NC 27695
harcourt@eos.ncsu.edu

January 18, 1994

Abstract

Application-specific programmable processing systems consist of not only a processor, but also the software that runs on it. In order to support development of such systems, a design environment must support both hardware and *software development. Unfortunately, there are no specification languages for processors that are suitable for such dual use. Therefore, we have designed a functional-style language that is specifically intended for describing instruction sets; its functional nature allows it to describe the result that an instruction produces without having to specify the mechanism of operation. This property is key to allowing dual use of the specification, since it will not be biased towards either hardware or software development. We are currently constructing a design environment based on our language.*

1 Introduction

An important tool in the development of an application-specific programmable processor is a simulation environment for the proposed instruction set in which test programs are "run" to evaluate performance and programmability [10]. Such an environment includes simulators, compilers, assemblers, linkers, and debuggers (or any subset thereof).

Unfortunately, construction of these tools is a labor-intensive task that may require many months to complete. This lengthy time-frame means that the resulting simulation system will be largely used as a design validation tool rather than as a tool for supporting actual design decisions.

Automatic generation of the tools would significantly reduce the amount of labor involved in their construction, thus speeding their availability. Earlier availability, in turn, allows earlier evaluation of design decisions; it also allows the evaluation tools to take direct part in the modify-evaluate design loop.

However, automatic generation of instruction set evaluation tools requires some means for specifying the instruction set to the generation system. Current languages used to describe instruction sets are limited to narrow classes of applications; for example, hardware description languages are suitable for producing simulators, and compiler machine description languages are useful for producing compilers. Unfortunately, none of these existing languages is suitable for use as a unified specification language.

Therefore, we have found the need to develop a specification language that is explicitly designed for describing instruction sets. This paper presents an overview of our language (called LISAS). (Elsewhere, we have outlined its usage in simulator generation and hardware development [1, 2].)

1.1 Background

Our overall research goal is a design environment for the development of application-specific programmable processors. The details of our planned environment are beyond the scope of this paper, but it is centered on a specification from which necessary hardware and software development tools are generated. We classify the central specification language in this environment as an *instruction set description language* (ISDL).

Initially, we considered various compiler machine description languages for use as an ISDL. Unfortunately, these languages (in general) simply describe the translation from an intermediate form into a target machine language, and they provide little additional information about the instruction set [5]. In particular, they do not contain enough information to construct simulators.

We also considered various hardware description languages (HDLs) for use as as an ISDL. However, HDLs are not suitable for generation of software tools since they "bury" the relevant information inside a hardware framework. (See [2] for a detailed explanation.)

Therefore, we have created our own ISDL. A major design requirement for our language is that it be suitable as a source for deriving hardware development tools *and* software development tools. As a consequence, the language must not be biased towards any particular application; that is, instructions should be described by giving the results they produce rather than the mechanisms they use. The following sections describe our ISDL.

1.2 Some Definitions

Throughout this paper, we use a few terms whose meanings are generally assumed. However, the definitions of these terms (as used in this paper) will be made explicit:

- An *instruction set architecture* (ISA) represents the logical, abstract behavior of a processor as seen by a programmer.
- An *organization* is a collection of functional units that performs the operations specified by an instruction set. Many different organizations can correspond to the same instruction set.
- An *implementation* is a physical realization of a particular organization of a given instruction set. There can be many different implementations of a single organization.

2 A Basis for ISA Specification

Consider instruction execution in a processor. The processor executes an instruction in the context of an initial state which places the processor in a new, second state; it then executes another instruction in the context of the second state, producing a third state. This operation produces a sequence of discrete state changes, where the mechanisms that carry out the changes are invisible.

This process can be further subdivided to give two (somewhat overlapping) views of instruction set architecture. These views form a basis for instruction set specification.

2.1 Architecture as State + Transformations

The first view of an architecture is as a set of storage elements (*e.g.*, memory and registers) that represents a *state* and a set of *operations* (*e.g.*, instructions) that transform the state. Given this view, architectural descriptions consist of a set of memory and register declarations (state declarations) and a set of instruction specifications (state transformations).

Such a mechanism for describing architectures is neither new nor unique. However, the problem with this technique alone is that there is no architectural basis for the operations that represent the state transformations. For example, given an instruction that performs a register-to-register addition, its state transformation can be described as `r[dst] = r[src] + r[dst]`, where `r[dst]` is given a value in the next state based on two values in the current state; the difficulty is that there is no specification of the meaning of the "+" operator. Therefore, this view of architecture must be augmented with specifications of the operations used to describe the state transformations.

2.2 Architectural Data Types

The second view of architecture is as a collection of what we call *architectural data types*, where each type encompasses a set of *abstract values* and a set of *abstract operations* on those values. Each set of abstract values also has an associated *abstract representation*.

This notion of architectural data type is very similar to the notion of *abstract data types* in high-level programming languages (HLLs). In HLLs, abstract data types represent a set of operations and values, but do not specify the representations of the values. However, for ISA specification, an abstract representation must be associated with each architectural data type since some of the abstract operations of the architecture may require knowledge of representation. For example, the definition of two's complement overflow requires knowledge of the representation since overflow occurs when the result of an operation requires more bits than are available in the representation.

Therefore, a processor is a realization of a set of architectural data types, just as it is an implementation of an architecture. The data words manipulated by the processor are the physical representations of the sets of abstract values, and the instructions are the physical representations of the abstract operations.

2.3 The Unified Specification Technique

The second view above can be mapped onto the first in order to derive a method of specification in which architectural features can *only* be defined in terms of other architectural features. In this method, the values of the architectural data types are mapped onto the various memory elements that form the state. Similarly, the operations of the architectural data types are used to describe the transformations on the state.

Therefore, the steps in creating a specification are

- describe the set of architectural data types using standard programming language techniques for specifying abstract data types;
- declare the memories and registers comprising the state;
- describe how the values of the architectural data types map onto the state; and
- describe the operations of instructions on the state in terms of the operations of the architectural data types.

Thus, an architecture is specified as a state and a set of transformations, where the transformations are described using the operations of the architectural data types. A language must now be defined which embodies this specification process in a natural manner.

3 A Functional Approach

Current approaches to instruction description (as in processor data books) largely use imperative languages. However, such languages require that the state transition mechanism be explicitly specified, which is clearly incompatible with the specification process outlined above. These considerations therefore lead to the use of functional language techniques for reasons similar to those for using functional programming languages instead of imperative programming languages [9]. (Note, however, that we are *not* claiming that a functional approach is the only approach.)

First, the style of programming in functional languages encourages non-operational specification of computations. Here, this style will have the similar effect of encouraging purely architectural (hardware-independent) specification. While it is not possible to prevent an architect from including implementation details, the functional framework will help make such inclusions either difficult or obvious.

Second, functional programming facilitates an algebraic style of specification which allows for easier equational reasoning and program manipulation than with imperative languages. These properties will simplify the processes of architectural analysis, deriving instruction patterns for compiler generation, and verifying the correctness of implementations.

Finally, functional specifications are also executable. An architect can therefore verify that the specification describes the desired behavior by simply executing the specification and checking that the results match expectations.

Within this approach, instructions are specified as functions from one processor state to another. Each instruction takes an input state and produces an output state with possible intermediate, internal states.

4 Introduction to Lisas

LISAS is a functional-style, architectural description language built upon the principles described in Section 2. The fundamental elements of specification in LISAS are architectural data type declarations (which introduce the operations that an architecture can perform) and storage element declarations (which contain the values that are manipulated by the operations). LISAS then provides the mechanisms with which these operations and storage elements can be combined to create a complete specification.

Many of the syntactic features of LISAS are modified versions of constructs often found in functional programming languages. These modifications serve two purposes:

- they help keep LISAS descriptions in the architectural domain by tailoring the constructs to architectural uses; and
- they provide convenience of notation for common architectural features.

However, though the semantics of LISAS is functional, the syntax is not. Traditionally, processor descriptions have been written in a very imperative style, and we felt that the syntax of LISAS should reflect this tradition. Therefore, LISAS uses single-assignment style statements for describing instructions; these statements are easily translated into a functional form [9].

4.1 Lisas Example

This section presents an overview of the LISAS architectural specification language. The overview will be based on the partial specification shown in Figure 1, which shows part of a specification of a custom microprocessor designed for biotelemetry applications [4]. This fragment is only a small portion of the entire description, but it is largely self-contained. The example will be considered in sections.

4.1.1 Architectural Data Types

The first line of the description declares `word` to be a 16-bit, two's complement integer type, where `twos_comp` is a module containing a specification of two's complement values and operations. LISAS predefines several of these modules, including ones for unsigned inte-

```
type word = twos_comp(16)

// processor state
register r : word = [16]<16>
memory   M : word = [65536]<16>

// condition codes
flag C = <1>
flag N = <1>
flag V = <1>
flag Z = <1>

// addressing modes
access Register (reg, _) = (r, reg)
access Indirect (reg, _) = (M, r[reg])
access Postinc  (reg, _) = (M, r[reg])
                           with  r[reg]' = r[reg] + 1;
                           end
access Indexed  (reg, ext) = (M, r[reg] + ext)

// addressing mode encoding
selector a_modes (mode, reg, ext) =
    mode == 0  =>  Register (reg, ext);
    mode == 1  =>  Indirect (reg, ext);
    mode == 2  =>  Postinc (reg, ext);
    mode == 3  =>  Indexed (reg, ext);
  end

// opcode extension word sizes
selector e_size (mode) =
    mode == 0  =>  0;
    mode == 1  =>  0;
    mode == 2  =>  0;
    mode == 3  =>  16;
  end

// instruction encoding format
format two_op = (OP<4>, sm<2>, dm<2>, src<4>, dst<4>,
                ext1<e_size(sm)>, ext2<e_size(dm)>)

// ADD instruction
insn add two_op =
    OP = 0;
    op1 : word = a_modes (sm, src, ext1);
    op2 : word = a_modes (dm, dst, ext2);
  in
    op2' = op1 + op2;
    C' = op2'!carry;
    N' = op2'!negative;
    V' = op2'!overflow;
    Z' = op2'!zero;
  end
```

Figure 1: An Example Specification in LISAS

gers, IEEE single precision floats, IEEE double precision floats, and a number of others.

Users can also create specification modules for data types that cannot be derived from the predefined modules. The specifics of this mechanism have not yet been defined, but future work can adapt one of the methods of data type specification associated with programming language definition (Watt's text [11] describes several alternatives). However, data type specification is, by design, orthogonal to the rest of LISAS, and its absence has no effect on the remainder of the language.

4.1.2 State

The next six lines describe the memory elements that form the architecture's state. The element `r` is a register file containing sixteen 16-bit registers, and `M` is a memory with 64K 16-bit words (which are the smallest addressable units of the memory); the declarations specify that the individual registers and memory locations are to be by default interpreted as being of type `word`. The elements `C`, `N`, `V`, and `Z` are 1-bit flags representing condition code bits.

4.1.3 Mappings

The above register and memory declarations contain implicit mapping declarations that specify how the values of the `word` data type map onto the memory elements. The specification that elements of `r` and `M` should be interpreted as being of type `word` actually specifies that values of type `word` should be bitwise mapped onto the elements of the memories.

Explicit mappings are also allowed. In this form, the values of a type are mapped into a memory by giving the bitwise correspondence between the values and a group of bits in the memory which are specified relative to some base address. (See Section 5.3.)

4.1.4 Addressing Modes

Operand access modes simply describe where operands are located. The example shows declarations for four **access** modes: `Register`, `Indirect`, `Postinc`, and `Indexed`. Each access mode returns a pair, where the first member is a storage element, and the second member is the address of the operand within that element. (The underscores represent place holders for arguments that will be passed to the access mode but not used.)

Access modes may also have side-effects. For example, `Postinc` causes a register to be incremented *after* the register is used as an address (see Section 4.1.7, "State Changes", below).

4.1.5 Encodings

Instruction encodings are specified using **selector** functions and **format** declarations. The purpose of selectors is to specify encodings or decodings that are orthogonal to the behavior of the function being encoded or decoded; the justification is that the same function may be encoded in different ways in different places. In the example, the `a_modes` selector encodes the addressing modes, and the `ext_size` selector decodes the size (or presence) of extension words.

The actual instruction formats are described using **format** declarations. The example shows a single format called `two_op` that gives the encoding used with two-address instructions; it simply lists the names and sizes of each field of the instruction. By default, instructions map into memory with the leftmost fields at lower addresses; different mappings may be obtained using explicit mapping functions. Note also that variable sized fields can be specified using selectors.

4.1.6 Operations

The single operation in the example is a two's complement addition that is encapsulated in the `add` instruction declaration. The first line of the instruction declares its name and gives the name of the format used to encode it; the names of the fields of this format are then brought into the scope of the instruction declaration.

The next three lines set a fixed value for the `OP` field of the instruction's format and declare its two operands. These declarations use the `a_modes` selector to choose an operand access mode for each of the operands (`op1` and `op2`).

The body of the instruction description specifies the actual results of the instruction. It states that new values for `op2`, `C`, `N`, `V`, and `Z` are produced by the two's complement sum of `op1` and `op2`. The "!" notation denotes an *attribute*, where the right-hand name is an attribute that is attached to the left-hand name (see Section 4.1.8, "Attributes"); the prime notation denotes state changes and is described below.

Note that the *syntax* of the instruction specification is imperative. This syntactic style was chosen because instruction specification has traditionally relied on RTL-like imperative statements. However, the synchronous and explicit nature of state changes in LISAS allows easy transformation of this syntax into a more traditional, functional form.

4.1.7 State Changes

In LISAS, operations are specified as state changes. An instruction description takes the current state as an implied argument and generates a new state in which some of the storage elements may have new values. The prime notation indicates the new values; for example

```
op2' = op1 + op2
```

means that in the next state, `op2` will have the value of `op1 + op2` as evaluated in the current state.

Side effects from access modes are propagated into the set of state changes generated by the corresponding instruction. For example, any side effects generated by the access mode for `op2` occur with the state changes in the body of the `add` instruction.

However, in some cases, intermediate states are necessary to produce deterministic behavior. For example, both operands of the `add` instruction can use the `Postinc` access mode with the same register, and for this processor, these accesses must occur sequentially. When such conflicts occur (or might occur), LISAS resolves them by ordering the conflicting state updates as they are written, thus producing intermediate transitions. (Any tool that might process a LISAS specification should give a warning when these conflicts are detected.)

Given the above semantics, state transitions are easy to locate, and the sequence of state transitions is easy to identify; there are no random or "asynchronous" state changes. An instruction simply accepts as "input" the current state of the processor and produces a new state as "output".

4.1.8 Attributes

Attributes provide a mechanism to capture ancillary results of operations that might not be frequently used. An operation that generates such results attaches them as attributes to the "primary" result; they may then be accessed if needed. In the example, the two's complement addition has attached attributes giving the carry out, sign, overflow status, and zero status to the result, and they are then used to set the condition code flags. Attributes are also used to indicate that an operation has generated an exception; for example, divide operations will attach a "div_by_zero" attribute to the result.

The attribute mechanism is simply syntactic sugar for operations that return tuples. For example, the body of the `add` instruction in Figure 1 could also be written as

```
(op2', C', N', V', Z') = op1 + op2;
```

In general, most operations will have a primary result and a number of ancillary results; however, in many instances, only the primary result will be of interest. Therefore, most operations can be written as returning a tuple or as returning a primary result with ancillary results attached as attributes.

5 Additional Lisas Constructs

This section describes several language constructs that are not utilized or are not fully described in the example of Section 4. However, this section does not provide a complete description of the entire language.

5.1 Operands and Storage Access

One particular problem with HDLs is due to the mechanism they use to describe operands. The semantics of HDLs are such that the use of an operand represents an actual fetch of that operand. Thus, two different mechanisms are needed for operand access: one to fetch the operand and one to store it.

Unfortunately, use of such mechanisms contaminates a description with organization level information. At the ISA level, only the location of an operand and the computation of its address are of interest; the specifics of its access are not relevant. Therefore, some alternate semantics for operand access must be found; such an alternative can be derived by looking at operand access at an abstract level.

Access to operands within storage elements involves the concepts of *storage bases*, *storage classes*, *access modes*, and *access classes* [3]:

- A *storage base* represents the smallest addressable unit of the corresponding storage element.
- A *storage class* represents the mapping of a logical storage unit onto a smaller storage base.
- An *access mode* describes the use of an addressing mode with a given storage class; thus, for each addressing mode, there are as many access modes as there are valid storage classes. Note that while all of the access modes associated with a specific addressing mode may yield the same address, they may have different side effects.
- An *access class* is the set of all access modes that can be used to obtain a given operand.

Given these definitions, access to an operand proceeds as follows:

1. An access mode is selected from an access class specified by an instruction (selection of address computation).
2. The access mode is evaluated to obtain an address into a storage element (address computation).
3. A storage class specified by the instruction is used to determine the elements of the storage base to access (the operand mapping within the storage element).
4. Those elements are accessed.

The operand access semantics of LISAS are based upon the above process (refer to Figure 1). A selector is used to construct an access class from which an access mode is chosen (the `a_modes` selector). The access mode itself describes the address of the operand and designates the storage element (storage base) in which it may be found (the four access modes). Finally, the type of the operand and its mapping (implicit in the example) onto the selected storage base determine the storage class of the operand. The key difference from the HDL approach is that an operand specifier in LISAS represents a description of where an operand may be found rather than the operand itself.

5.2 The `with` and `let` Constructs

The example of Figure 1 uses a **with** statement to define the `Postinc` addressing mode. In general, **with** statements are used within access declarations to compute an expression that is coupled to a following state change. The general form is

$$E \textbf{ with } S \textbf{ end}$$

where E is the expression and S is the state change that occurs after E is evaluated.

LISAS also includes a mechanism the **let** expression, to define expressions that are coupled to preceding state changes (again, for use within access declarations). The general form of the **let** expression is

$$\textbf{let } S \textbf{ in } E \textbf{ end}$$

where S is the state change and E is the expression that is to be evaluated in the new state produced by S. For example, the access declaration

```
access Predec (reg, _) =
          let  r[reg]' = r[reg] - 1;
          in   (M, r[reg]')
          end
```

describes a predecrement addressing mode.

5.3 Mappings

LISAS uses mapping declarations to describe how abstract representations (*e.g.*, of architectural data types or instructions) are physically represented in the processor's state (registers or memory). A mapping declaration basically gives the bit-wise correspondence between an abstract representation and its physical representation in a given memory element (relative to some base address).

An example will be used to illustrate mapping declarations. The following LISAS fragment specifies a 32-bit, two's complement numeric type, a byte-addressable memory, and a little-endian mapping of the type onto the memory:

```
type word = twos_comp(32)
memory M = [2**32]<8>
map word <-> M =
    (word<7::0>, M[$])
    (word<15:8>, M[$ + 1])
    (word<23:16>, M[$ + 2])
    (word<31::24>, M[$ + 3])
    end
align word -> M = 4
```

where $ represents the base address. This base address corresponds to the address applied to `M` during an access of type `word`.

The fragment also specifies how a `word` must be aligned in `M`. The "4" means that addresses of `words` in `M` must be integer multiples of 4; in general, any multiple can be used. Alternatively, the special alignment `natural` could be used as follows:

```
align word -> M = natural
```

to indicate that `word` is aligned on multiples of its size in terms of the base units of `M` (an 8-bit byte in this example).

Specifying mappings as above for all combinations of types and memory elements can be quite tedious. Therefore, LISAS provides shorthand notations for common cases. The example above could be rewritten as

```
map word <-> M = little_endian
```

A `big_endian` mapping is also provided.

5.4 Constraints

In some situations, description of special cases can be simplified by first describing the general case and then separately describing the special case in the form of a *constraint*. Therefore, LISAS provides a mechanism for attaching constraints to any arbitrary part of a specification.

The general form of a constraint specification is

where *E*

where *E* is a boolean expression that describes the constraint. For example, the declaration

```
memory r = [32]<32> where (r[0] = 0)
```

means that `r` has 32 registers of 32 bits each and that the value of `r[0]` is always 0.

5.5 Hints

There are a number of factors that are not actually features of an instruction set architecture but which may be considered part of a system architecture. Two examples are procedure calling conventions (including which registers are caller-save and which are callee-save) and registers assigned to a particular function (such as a frame pointer). It may also be desirable to be able to pass information directly to a specific tool that processes LISAS descriptions (*e.g.*, implementation hints to a synthesis tool). Therefore, LISAS provides a mechanism, *hints*, for specifying this information.

A hint consists of a compound name for the hint and a body, where the contents of the body are the responsibility of the author of the hint. The general form of a hint is

%<*class*>.<*name*> { <*body*> }

where *class* specifies the general class of the hint, *name* gives the specific name, and *body* is the body which is made directly available (without processing) to other tools. For example,

```
%compiler.fp { R[13] }
```

informs the compiler-generator that it should use R[13] as the frame pointer.

5.6 Instruction Sequencing

LISAS contains a special construct, called a *fetch block*, for describing a processor's instruction sequencing mechanism. The general form of a fetch block is

fetch = <*decls*> **in** <*rtl*> **end**

where *decls* are declarations of values used in the fetch block, and *rtl* is a set of RTL statements describing the actual fetch operation.

The RTL language in LISAS contains an `execute` statement for specifying an instruction to execute. The general form of this statement is

execute <*ref*>;

where *ref* is a reference to a storage location containing the instruction to execute.

The following is an example of a simple fetch block in which all instructions are two memory units long:

```
fetch = in
    PC' = PC + 2;
    execute M[PC];
  end
```

Note that the program counter (`PC`) is incremented before the instruction is actually executed.

6 Discussion

There are two (related) issues regarding the nature of LISAS that require further discussion:

- the imperative syntax vs. the functional nature of LISAS, and
- the presence of side-effects.

The following subsections elaborate upon these topics.

6.1 The Functional Nature of Lisas

As mentioned previously, LISAS represents instructions as functions from one processor state to another. That is, instructions are functions that take the current state as input and produce the next state as output.

However, LISAS does not make the complete input and output states of the instructions explicit. Instead, the input state is implicitly assumed to be the current state, and the output state is specified by only giving the modifications to the current state. This style was chosen to permit a syntax design that would be at least partially familiar to processor designers.

In fact, syntax design was a major focus in the development of LISAS[1]. Traditionally, processor developers have used imperative languages to (partially) describe their designs, and the *syntax* of LISAS was chosen to reflect this tradition. Thus, the output state of an instruction is specified by only giving the changes to the input state, and these changes are specified using syntax that has the *appearance* of a sequence of imperative assignments.

Nevertheless, this syntax does not mean that LISAS is not a functional language. The "assignment" statements, including side effects caused by addressing modes, can be easily rendered into a purely functional form.

```
fun get (e,a) nil = 0
  | get (e,a) ((e',a',v')::ss) = if e=e' andalso a=a'
                                 then v'
                                 else get (e,a) ss

fun sum x y s = (get x s) + (get y s)
fun assign (e,a) v s = (e,a,v)::s

fun Register (reg, _) = fn s => ("r", reg)
fun Indirect (reg, _) = fn s => ("M", get("r",reg) s)
fun PostInc (reg, _) = fn s => ("M", get("r",reg) s)
fun Indexed (reg, e) =
                 fn s => ("M", (get("r",reg) s) + e)

fun Register_side _  = fn s => s
fun Indirect_side _  = fn s => s
fun PostInc_side  (reg, _) =
          fn s => ("r",reg,(get("r",reg) s)+1)::s
fun Indexed_side  _  = fn s => s

fun a_modes (mode, reg, ext) = case mode of
      0 => Register (reg, ext)
    | 1 => Indirect (reg, ext)
    | 2 => PostInc  (reg, ext)
    | 3 => Indexed  (reg, ext)

fun a_modes_side (mode, reg, ext) = case mode of
      0 => Register_side (reg, ext)
    | 1 => Indirect_side (reg, ext)
    | 2 => PostInc_side  (reg, ext)
    | 3 => Indexed_side  (reg, ext)

fun add (sm, dm, src, dst, ext1, ext2) s =
    let
      val op1 = a_modes (sm, src, ext1) s
      val s1  = a_modes_side (sm, src, ext1) s
      val op2 = a_modes (dm, dst, ext2) s1
      val s2  = a_modes_side (dm, dst, ext2) s1
      val op2' = op2
    in
      assign op2' (sum op1 op2 s2) s2
    end
```

Figure 2: ML Representation of Figure 1

SML Representation of the Example

Figure 2 gives the SML equivalent of Figure 1 using the pure subset of SML. For the sake of brevity, however, the state and format declarations, along with the condition code settings, have been ommitted. (Note that Figure 2 is but one of many possible variations.)

The code in Figure 2 represents the state of the processor using a list of triples. Each triple consists of the name of a storage element (*e.g.*, "r" for the register file and "M" for the memory), an address of a location within that element, and the value at that location.

In other situations, (*e.g.*, the addressing mode functions), locations within the state are represented by a pair. The first component of the pair in the name of the storage element being accessed, and the second component is the address within that element.

Each addressing mode is represented by two functions: the first computes the actual address, and the second computes any state changes that occur as a side-effect. Thus, whenever an address is computed, its corresponding side-effect function must also be executed to update the processor's state. The body of the `add` function demonstrates this coupling.

6.2 Side-Effects

One complication within LISAS is the use of side-effects within addressing mode declarations. While

[1]While syntax may not be important to the semantics of a language, it is very important to that language's potential acceptance.

there are some (complicated) alternatives available, this structure was chosen because real addressing modes in real processors have side-effects; LISAS simply reflects this reality.

However, the location of potential side-effects is very restricted within LISAS. Side-effects are only allowed within addressing mode declarations so that the scope of their effects is limited to the instructions within which they are used. The practical consequence is the need for intermediate states discussed in Section 4.1.7.

7 Conclusions and Further Work

At the present time, the initial design of LISAS is complete. We have implemented a language processor that checks a specification for validity and produces an intermediate representation. We have also written LISAS specifications for several processors, including DEC's Alpha and Intel's i386.

In addition, we are currently pursuing three projects based on LISAS:

- integration of LISAS with commercial hardware development tools [1, 2];
- generation of compiler code generators (through a process based on formal semantics of instruction sets [7, 6]); and
- augmentation of LISAS to include timing specifications [8] based on Milner's process calculus (SCCS).

Once these projects are complete, LISAS will provide a unified basis for developing *both* the hardware and software components of an application-specific processing system.

References

[1] **Todd A. Cook**, **Paul D. Franzon**, **Ed Harcourt**, and **Thomas K. Miller III**. "Behavioral Modeling of Processors from Instruction Set Specifications". Presented at the *Second Annual International Verilog HDL Conference* (1993).

[2] **Todd A. Cook**, **Paul D. Franzon**, **Ed Harcourt**, and **Thomas K. Miller III**. "System-Level Specification of Instruction Sets". In Proceedings of the 1993 IEEE International Conference on Computer Design, pages 552–557, 1993.

[3] **Annie Despland**, **Monique Mazaud**, and **Raymond Rakotozafy**. "PAGODE: A Back End Generator Using Attribute Abstract Syntaxes and Term Rewritings". In *Compiler Compilers (Springer-Verlag LNCS 477)*, pages 86–105, 1990.

[4] **Kenneth W. Fernald**, **Todd A. Cook**, **Thomas K. Miller III**, and **John J. Paulos**. "A Microprocessor-Based Implantable Telemetry System". *IEEE Computer*, 24(3):23–30, March 1991.

[5] **Christopher W. Fraser**. "A Language for Writing Code Generators". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 238–245, 1989.

[6] **Robert Giegerich**. "Code Selection by Inversion of Order-Sorted Derivors". *Theoretical Computer Science*, pages 177–211, 1990.

[7] **Ed Harcourt**, **Jon Mauney**, and **Todd Cook**. "Functional Specification and Simulation of Instruction Set Architectures". In *Proceedings of the International Conference on Simulation and Hardware Description Languages*. SCS Press, 1994. To appear.

[8] **Ed Harcourt**, **Jon Mauney**, and **Todd A. Cook**. "Specification of Instruction-Level Parallelism". In *Proceedings of the North American Process Algebra Workshop*, 1993. Available from the Dept. of Computer Science at Cornell University as technical report TR93-1369.

[9] **Paul Hudak**. "Conception, Evolution, and Application of Functional Programming Languages". *ACM Computing Surveys*, pages 359–411, September 1989.

[10] **Pierre G. Paulin**. "DSP Design Tool Requirements for the Nineties: An Industrial Perspective". In *Sixth International Workshop on High-Level Synthesis*, Nov. 1992.

[11] **David A. Watt**. *Programming Language Syntax and Semantics*. Prentice-Hall, New York, 1990. QA76.7 W4 1990.