System-Level Specification of Instruction Sets

Todd A. Cook* Paul D. Franzon* Ed A. Harcourt† Thomas K. Miller III*

*Department of Electrical and Computer Engineering
†Department of Computer Science
North Carolina State University
Raleigh, NC 27695

Abstract

System-level design requires some sort of specification for a system at the level of abstraction of the system. When the system (or sub-system) is a processor, the appropriate level of abstraction is the instruction set. However, there are no good approaches for describing processors at this level.

Nevertheless, this type of specification has a number of benefits: it is more concise (and thus less error-prone) than more general alternatives; it can be re-used in later re-implementations; and it provides support for software codesign through compiler-generators (which rely on higher-level abstractions than other techniques provide). Therefore, we have developed a methodology and an embodying language for specifying processors at the instruction set level.

1 Introduction

System-level design is obviously a very critical stage in the overall design process. Poor decisions at this early stage can cause significant cost and time problems during implementation. On the other hand, the greatest potential for decreasing costs and design time is also at this stage [4].

Addressing these issues requires accurate, concise specification at the system level with the intent to support simulation, verification, and documentation. A designer will analyse such a specification to decide upon the best approach to developing a structure that implements the behavior of the system. The accuracy and completeness of the specification will have a direct influence on the accuracy and completeness of the final design; an incomplete specification can only lead to an incomplete design or a design filled out by implementation considerations (which may be contrary to system-wide considerations).

For processor-based systems, the highest level of abstraction is the instruction set. Proper implementation of an instruction set and its associated software tools (*i.e.*, assemblers, simulators, compilers, etc.) requires a detailed specification of what each individual instruction does. This requirement is particularly true of new implementations of existing instruction sets, where the behavior of the new implementation must exactly duplicate that of the old.

Our interest is in providing tool support for system-level design, where we include software development tools as part of the system (since their quality will affect system quality). However, to provide such support requires some means of specifying the system to the tools. Unfortunately, there are no adequate techniques for directly specifying the abstract behavior of an instruction set. Existing alternatives (HDLs) rely on including some level of implementation detail that unnecessarily biases the final design towards that particular implementation.

Therefore, we propose that HDLs be supplemented with languages specifically designed for the domain of instruction set description and containing the higher level abstractions of that domain. At this level, description languages should contain constructs that directly represent instruction set features, such as instructions, addressing modes, encodings, etc.

In this paper, we describe the requirements for specification at the instruction set level and present a language that embodies them; this language is what we call an *instruction set description language* (ISDL). We then contrast specification using our language with specification using Verilog. (We use Verilog as an ex-

ample for convenience; our comments also apply to other HDLs such as VHDL.)

2 Specification at the Instruction Set Level

For our purposes, we will divide a processor into four levels of abstraction:

- The *instruction set architecture* (ISA) level represents the logical operation of a processor as seen by an assembly language programmer.
- The *organization* level consists of a pipeline structure and a collection of functional units that can perform the operations specified by an instruction set.
- The *datapath* level comprises the elements (multiplexors, ALUs, latches, etc.) necessary to implement the structures of the organization level.
- The *logic* level consists of the logic equations and state machines that implement a given datapath.

An important characteristic of the above hierarchy of abstractions is that anything that can be specified at a given level can also be specified at a lower level. For example, if we start with a behavioral description of a pipeline, we can create a set of logic equations that implements the exact same behavior. However, we do not really want to start a pipeline design at the logic level: the behavioral description gives us a global view of the pipeline that is not easily derivable from logic equations.

This situation is equally true for instruction set description (embodied in ISDLs) versus generic hardware description (embodied by HDLs such as VHDL or Verilog). It is certainly possible to take an instruction set specification written in an ISDL and write a high-level, behavioral description in an HDL that produces the same behavior.

Nevertheless, information about the instruction set will be lost in the translation to the HDL. An ISDL will represent ISAs at the level of abstraction of instruction sets; for example, an addressing mode will be directly represented with an addressing-mode primitive. On the other hand, an HDL would have to represent the same addressing mode with at least one (probably two) generic function(s); there will be no indication in the description of the purpose of these functions. Explicit indication of purpose, however, is required for some uses of instruction set specifications (see below).

Thus, there is a real difference in specification at the instruction set level and specification at the hardware behavioral level. A specification at the ISA-level is written in terms of ISA entities such as "instructions", "addressing modes", and "data types" rather than in terms of organizational entities such as "buses", "pipelines", and "caches"; any use of organizational features in a specification biases the specification towards a particular implementation.

Furthermore, HDLs are generally oriented towards hardware simulation and synthesis, and they work well for these tasks. However, there are other applications for instruction set specifications besides hardware development, and unfortunately, HDLs do not serve these tasks as well. We will consider compiler generation as an example.

Traditional HDL applications do not need to understand that which is being described. For example, an HDL simulation system can operate by simply knowing the behavior of the language primitives; simulation of these behaviors then produces the behavior of the entire system without any knowledge of what that behavior represents.

However, compiler generation systems require some coarse understanding of the ISA being specified. A compiler-generator, for example, will need to be able to determine what addressing modes are available. But, even though an HDL will provide all the mechanisms necessary to describe the addressing modes, there will be no linguistic means for distinguishing such descriptions from those of other parts of the processor.

Unfortunately, while a human reader may be able to easily see that a particular function implements addressing modes, an automated system will not, since determining the purpose of an arbitrary function is incomputable [2]. Furthermore, even though heuristic algorithms can be used for locating such features, they will not be reliable in the presence of unusual variations.

The overall theme here is that the higher level of abstraction of an ISDL can increase the number of applications of instruction set specifications. This larger set promotes quicker and more efficient exploration of design alternatives because more of the support work (*i.e.*, compiler generation) can be automated.

3 An ISDL

To demonstrate the above concepts of instruction set description languages, we have developed an ISDL that we call LISAS (Language for Instruction Set Architecture Specification). We first describe the semantic basis for LISAS, and we then present a brief,

informal overview of the language through a small example.

3.1 The Semantic Basis of LISAS

We can view an ISA 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, instruction set descriptions consist of a set of memory and register declarations (state declarations) and a set of instruction specifications (state transformations).

We can also view an instruction set 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 our application, a representation is needed since the results of some operations (*i.e.*, overflow from two's complement arithmetic) depend on the representation.

We can map the second view above onto the first in order to derive a method for describing instruction sets. Thus, 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, we specify an ISA as a state and a set of transformations, where the transformations are described using the operations of the architectural data types.

3.2 An Informal Overview of LISAS

Our overview will be based on the partial specification shown in Figure 1. This example shows part of a specification of a custom microprocessor, named PERC, that was designed for biotelemetry applications [1]. This fragment is only a small portion of the entire description, but it is largely self-contained.

```
type word = twos_comp(16)

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

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)

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

selector ext_size (mode) =
    mode == 0  =>  0;
    mode == 1  =>  0;
    mode == 2  =>  0;
    mode == 3  =>  16;
  end

format two_op = (OP<4>, sm<2>, dm<2>, src<4>,
                 dst<4>, ext1<ext_size(sm)>,
                 ext2<ext_size(dm)>)

insn add two_op =
    OP = 0;
    op1 : word = a_modes (sm, src, ext1);
    op2 : word = a_modes (dm, dst, ext2);
  in
    op' = op1 + op2;
  end
```

Figure 1: An Example Specification in LISAS

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

The next two lines describe the memory elements that form the instruction set'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 references to register and memory locations are to be by default interpreted as of type *word*.

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 declaration 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.

The following six lines describe four addressing modes (*Register*, *Indirect*, *Postinc*, and *Indexed*), which in LISAS are called *access modes*. An access mode simply describes where operands are located; 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. 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 "State Changes" below). (A similar construct exists for specifying state changes that should occur *before* the address is formed.)

Next, there are two `selector` function 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 operation 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 to instructions.

An actual instruction encoding, named `two_op`, is specified by the following `format` declaration. This declaration simply lists the format's fields, giving their names and sizes. In cases where the size of a field can vary (such as an optional immediate value), the size can be specified using a selector function; in the example, `ext1` and `ext2` are two such fields.

The last part of the example is a declaration of a two's complement addition instruction. Its first line declares that the instruction is named `add` and that it is encoded using the `two_op` format; the specification of a particular format also introduces the names of its fields into the scope of the instruction description. The next line specifies that `OP` field of the encoding should be 0 (in general, assignment of a constant to a field of the encoding is a specification of an opcode field). The declaration then uses the *a_modes* selector to select an operand access mode for each of the two operands (*op1* and *op2*) using fields from the instruction encoding as parameters.

The actual operation that an instruction performs is enclosed within the `in` and `end` keywords. This instruction simply specifies that in the next state of the processor, the location described by op2 should contain the sum of the current values of op1 and op2. Since both op1 and op2 are declared to be `words`, the addition will be two's complement addition. The prime notation denotes state changes and is described below.

Note that the *syntax* of the instruction specification is imperative. We chose this syntactic style because instruction specification has traditionally relied on RTL-like imperative statements.

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* 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, we want these accesses to 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.

4 LISAS and Verilog

Figure 2 shows one possible translation into Verilog of the example of Figure 1. There are a number of alternatives to this translation, some of which are shorter; however, the given version most closely matches the details of Figure 1.

```
module example;

reg     [15:0]  r[15:0];
reg     [15:0]  M[65535:0];


function [15:0] Register_read;
  input   [3:0] rg;
  begin
    Register_read = r[rg];
  end
endfunction

task Register_write;
  input   [3:0] rg;
  input  [15:0] data;
  begin
    r[rg] = data;
  end
endfunction

function [15:0] Indirect_read;
  input   [3:0] rg;
  begin
    Indirect_read = M[r[rg]];
  end
endfunction

task Indirect_write;
  input   [3:0] rg;
  input  [15:0] data;
  begin
    M[r[rg]] = data;
  end
endfunction

reg     [15:0]  Postinc_addr;

function [15:0] Postinc_read;
  input   [3:0] rg;
  begin
    Postinc_addr = r[rg];
    Postinc_read = M[r[rg]];
    r[rg] = add_2s (r[rg], 1);
  end
endfunction

task Postinc_write;
  input  [15:0] data;
  begin
    M[Postinc_addr] = data;
  end
endfunction

function [15:0] Indexed_read;
  input   [3:0] rg;
  input  [15:0] index;
  begin
    Indexed_read = M[add_2s(r[rg],index)];
  end
endfunction

task Indexed_write;
  input   [3:0] rg;
  input  [15:0] index;
  input  [15:0] data;
  begin
    M[add_2s(r[rg],index)] = data;
  end
endfunction

function [15:0] a_modes_read;
  input   [1:0] mode;
  input   [3:0] rg;
  input  [15:0] index;
  begin
    case (mode)
      0 : a_modes_read = Register_read (rg);
      1 : a_modes_read = Indirect_read (rg);
      2 : a_modes_read = Postinc_read (rg);
      3 : a_modes_read = Indexed_read (rg, index);
    endcase
  end
endfunction

task a_modes_write;
  input   [1:0] mode;
  input   [3:0] rg;
  input  [15:0] index;
  input  [15:0] data;
  begin
    case (mode)
      0 : Register_write (rg, data);
      1 : Indirect_write (rg, data);
      2 : Postinc_write (data);
      3 : Indexed_write (rg, index, data);
    endcase
  end
endfunction

task decode
  input  [15:0] ir;
  input  [15:0] ext1;
  input  [15:0] ext2;
  begin
    case (ir[15:12])
      0 : add (ir[11:10], ir[9:8], ir[7:4],
               ir[3:0], ext1, ext2);
    endcase
  end
endtask;

task add;
  input   [1:0] sm;              input   [1:0] dm;
  input   [3:0] src;             input   [3:0] dst;
  input  [15:0] ext1;            input  [15:0] ext2;
  wire   [15:0] op1;             wire   [15:0] op2;
  begin
    op1 = a_modes_read (sm, src, ext1);
    op2 = a_modes_read (dm, dst, ext2);
    a_modes_write (dm, dst, ext2, add_2s(op1,op2));
  end
endtask

endmodule
```

Figure 2: Verilog Version of LISAS Example

The Verilog version turns out to be much longer than the LISAS version. The three main reasons for this expansion are that

- addressing mode descriptions are more complicated;
- side-effects require additional support; and
- there is no direct way to specify instruction formats.

(All of these are equally applicable to HDLs in general, not just Verilog.) We elaborate on each of these below.

Addressing Modes. LISAS describes addressing modes as a pair, where one element represents the memory to be accessed and the other element represents the address into this memory. An operand declaration thus represents the location of the operand instead of the operand itself; the location is implicitly dereferenced when the operand is used.

In Verilog, however, we must represent the addressing mode using the actual addressing operation. Furthermore, since fetching and storing an operand are actually two different operations, we must have two different representations of the addressing mode. Thus, the Verilog version in Figure 2 requires a function and task for each mode, representing fetching and storing, respectively.

Side Effects. It is common for addressing modes to have side effects, such as post-incrementing a register. In most cases, the side effect only occurs once when the operand is fetched; if a result is to be stored, it is put into the location from which the operand was fetched. LISAS directly supports this common case by way of the `with ... end` clause: the given side effect only occurs when the operand is fetched, and the same address is used for fetching and storing.

In Verilog, however, we must explicitly construct a similar mechanism ourselves. Since the `Postinc` mode of Figure 1 increments the register used to compute the operand address for fetching, the address must be preserved so that it can be used for the store operation. Therefore, the Verilog version introduces a register, `Postinc_addr`, to hold this address, and `Postinc_write` uses the register as the address for storing data. We should note that `Postinc_addr` represents an *implementation* detail since it is not defined by the ISA.

Instruction Formats. LISAS provides a mechanism for explicitly describing instruction formats. They are specified as a named list of fields, where each field is given a size in bits.

There is no equivalent mechanism in Verilog for describing formats directly. Instead, we describe them indirectly using a decoder, which selects an instruction and then extracts the fields that instruction needs. The `decode` task in Figure 2 performs this function; if any other instructions were to be added, they would be included in `decode`'s case statement.

Note that there is a substantial semantic difference between the LISAS and Verilog versions. The LISAS format declaration simply describes an arrangement of bits. The Verilog version describes such an arrangement as well, but it also introduces a decoding structure that is *not* part of the ISA.

5 Conclusion

The status of our work is that we have completed the design of the LISAS specification language and have implemented a language processor for it that does error analysis on specifications. We are currently working on two projects: integration of LISAS with Verilog for lower-level systems development, and derivation of compiler-code generators. Furthermore, we are also working on formal and rigorous techniques (based on Milner's process calculus [3]) for specifying system-level timing; these techniques will be integrated with functional specifications in LISAS to provide more complete system-level specifications.

References

[1] **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.

[2] **C. A. R. Hoare** and **D. C. S. Allison.** "Incomputability". *ACM Computing Surveys*, pages 169–178, September 1972.

[3] **Robin Milner.** *Communication and Concurrency.* Prentice-Hall, New York, 1989. QA267 M533 1989.

[4] **Steven E. Schulz.** "An Overview of System Design". *ASIC & EDA: Technologies for System Design*, pages 12–21, January 1993.