Functional Specification and Simulation of Instruction Set Architectures

Ed Harcourt Jon Mauney
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695

Todd Cook
Dept of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08855

Abstract

We use a functional Language to formally specify the semantics of an instruction set architecture. The resulting specification is more readable, modular, and concise than other semantic specification methods. The description is made clear and concise by creating *semantic actions* that represent *basic architectural operations* and by expressing each processor instruction and addressing mode in terms of these actions. The semantic description is directly coded in the functional language ML consequently providing an "executable specification" or simulator for the architecture. The semantic description is formal and can be used in other domains where a formal semantics of an architecture is needed, such as compiler/hardware verification and processor simulation.

1 Introduction

An *Instruction Set Architecture*, or ISA, represents a view of a processor as seen by the assembly language programmer. An *architectural* specification includes all information needed to write *correct* programs and is an abstract view of the architecture that hides implementation detail (*i.e.*, the *organization* of the processor). Abstract, implementation independent, views of computer systems (hardware or software) are necessary for good design, development, and implementation.

DEC's new Alpha architecture is a good example of an architecture specified independent of a particular implementation. A goal now is to make architecture specifications *formal*. Formal specifications have many desirable properties and are used for:

- Precise documentation
- Hardware and Compiler verification
- Automatic generation of software (*e.g.*, simulators, assemblers, and compilers).
- Rapid prototyping

We present a methodology for formally specifying a denotational semantics of an instruction set architecture. Moreover, since there are basic architectural operations common to all architectures (*e.g.*, reading and writing registers and memory) we create an *abstract data type* of primitive architectural operations suitable for representing a wide variety of architectures.

1.1 Denotational Semantics

A denotational semantics of a programming language represents each syntactic construct of the language in terms of mathematical objects (*e.g.*, sets, functions, relations). An architecture is essentially a low-level programming language (but a high-level view of a processor). An instruction is a syntactic object that represents an operation on the state of the processor. That is, an instruction is a state transformation function

$$\mathcal{I} : State \rightarrow State$$

and *State* is the type of a function from locations (register numbers or memory addresses) to values (bit strings).

$$\begin{aligned} State &= Locations \rightarrow Values \\ Locations &= RegNums + Addresses \\ Values &= integer \end{aligned}$$

If ρ is the state function then $\rho(x)$ is the value stored at location x. A new state can be built from an old state ρ with the state update function $\rho[x \mapsto d]$ which means "the new state with x updated to d". The update function is straightforward and we omit its definition. Given this definition of state we can begin to describe the semantics of individual instructions. For example, the MIPS `Add` instruction is defined by

$$\mathcal{I}[\![\texttt{Add R}_i\texttt{, R}_j\texttt{, R}_k]\!]\rho = \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_j) + \rho(\texttt{R}_k)]$$

The notation $\mathcal{I}[\![\texttt{-}]\!]$ represents the valuation function. The metabrackets "$[\![\texttt{-}]\!]$" surround the syntax of an instruction and separate the language being defined from the defining language. The state function ρ is an argument to the valuation function and an updated valuation function is returned. This coincides with our definition of instructions being functions from state to state.

The advantage of a denotational semantics is that it provides an ability to reason about specifications and to mathematically prove properties about the specification. For example, consider the definition of an instruction that increments by one the value stored in the register

$$\mathcal{I}[\![\texttt{Inc R}_i]\!]\rho = \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1]$$

Given the definitions of `Add` and `Inc` it is now an easy matter to prove that

$$\texttt{Inc R}_i \equiv \texttt{Add3 R}_i\texttt{, R}_i\texttt{, 1}$$

by checking that

$$\mathcal{I}[\![\texttt{Inc R}_i]\!] \equiv \mathcal{I}[\![\texttt{Add3 R}_i\texttt{, R}_i\texttt{, 1}]\!]$$

By substituting each side of the "$\equiv$" with its definition we see that

$$\rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1] \equiv \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1]$$

This property of being able to substitute equals for equals is known as *referential transparency* and is a property of functional specifications not shared by traditional imperative programming languages and some hardware specification languages such as VHDL or Verilog [BS89].

1.2 Semantic Algebras

We make our denotational specifications more readable and easier to work with by creating a library of architectural primitives. This library is essentially an abstract data type and is sometimes called a *semantic algebra* [Sch86]. The operations defined by the semantic algebra are called *semantic actions*. Rather than specify the architecture directly as a denotational semantics we first translate it into semantic actions defined by the semantic algebra. The semantic algebra is a *register transfer language*, or RTL, which is then specified denotationally.

1.3 Organization of the Paper

Section 2 describes how we specify an ISA semantics where semantic actions are used to represent basic architectural operations. Section 3 outlines the semantics of the actions using denotational semantics. Using our specification technique:

- Specifying the semantic actions occurs once. When this is done, many architectures can be specified using the predefined actions.
- The semantics of the actions need not be specified denotationally, but could be specified using any formal semantic method (*e.g.* operational or algebraic semantics). This gives our semantic specification an abstractness and modularity that is absent from other semantic specification methods. This style of semantics is called a *separated semantics* [Lee89].

Section 4 concludes.

2 The Semantics of ISA's

The denotational semantics of an ISA will comprise of the normal three parts: a syntactic domain, a semantic domain, and a valuation function.

- **Syntactic domain** — The syntactic domain is the instruction formats of the architecture being specified.
- **Semantic domain** — The semantic domain is a set of *semantic actions*. In this paper we use actions that implement a *register transfer language* (RTL).
- **Valuation function** — A valuation function specifies how an item in the syntactic domain maps to an item in the semantic domain. In this case, the valuation function formally describes an instruction in terms of the semantic actions.

To illustrate the process we use the PDP-11 as an example ISA.

2.1 Notation

It is common to use a modern functional language to specify a denotational semantics [Lee89], and in our case we will use Standard ML. While SML is not completely referentially transparent, due to its imperative features, its semantics is formally specified. Also, if we restrict ourselves to the purely functional subset we keep referential transparency.

Traditionally, the syntactic domain is specified with a context free grammar. Since the syntax of instructions is simple, SML's algebraic data type constructor (`datatype`) is used to specify the abstract syntax.

The semantic domain is a set of semantic actions that represent basic architectural operations and, as was mentioned before, is RTL. The semantic actions constitute a language with a syntax and a semantics and, to be complete, both must be specified.

The valuation function in a traditional denotational description will be described using SML functions. Since our semantics is directly coded into SML, a simulator for the architecture is immediately available.

2.2 The Syntactic Domain

The syntactic domain consists of the ISA's instruction formats. Figure 1 shows the SML definition of the syntax of the PDP-11 instruction formats. For example, the PDP-11 instruction `Add R0,(R1)+` is represented by the SML construct,

```
TwoOp1(ADD, RegDirect(0), AutoInc(1))
```

and the instruction `Add 10(R2), #10` is represented by,

```
TwoOp1(ADD, Indexed(10,2), Immediate(10))
```

Here, `TwoOp1`, `RegDirect`, `AutoInc`, `Indexed`, and `Immediate` are type constructors (or tags).

2.3 The Semantic Actions

This section describes the syntax of the language of semantic actions (RTL). We defer their implementation until section 3. The choice of the semantic actions is important: they must be capable of specifying the low-level semantics of a machine instruction and also be suitable as an intermediate representation for the front end of a compiler. Register transfer language (RTL) satisfies both criteria[Dav84]. Figure 2 shows the SML representation of the RTL syntax. The RTL semantic actions constitute an *abstract data type* that separates the RTL syntax from its semantics. Now, the syntax can be used effectively for pattern matching. This keeps the low-level implementation details of the RTL operators hidden.

The RTL actions are of two kinds — *values* and *imperatives*. Value actions produce values (*e.g.,* memory fetching, register access, addition, etc.) and imperative actions alter the state (*e.g.,* assignment and statement sequencing). A valid RTL program is a sequence of imperatives. The `Parallel` operator specifies that an instruction has multiple effects on the state that occur simultaneously. For example, an instruction `ADD` *Dest, Source* performs the operation $Dest = Dest + Source$ and, in parallel, assigns condition codes based on $Dest + Source$.

2.4 The Valuation Function

The valuation function maps syntactic objects (instructions) to semantic actions (RTL). We specify the valuation function in two parts, addressing modes and instructions. The valuation functions build *prefix operator terms* (or *abstract syntax trees*) that represent the effect of the operations. It is these terms that can either be given a semantics (as we will do in the next section), matched for code selection [AGT89, Dav84], or analyzed for code optimizer generation [FW88].

In the following discussion, for the sake of clarity, and adhering to common notational conventions, we have strayed a bit from using strict SML syntax.

Addressing Modes— A valuation function for an addressing mode returns a `Value` (figure 2) that describes how an operand is accessed. Some addressing modes (e.g., auto-increment/decrement modes) cause side effects on the processor state. Consequently, an addressing mode also returns an `Imperative` that specifies this side-effect. Where no side-effects take place, the imperative `Nop` is returned. The signature for the addressing mode valuation function `Op` is:

$$\texttt{Op:}\quad \texttt{Operand} \rightarrow \texttt{Mode} \rightarrow (\texttt{Value} \times \texttt{Imperative})$$

Where the type `Operand` (figure 1) represents an operand including its addressing mode and `Mode` (figure 2) is the size of the data being accessed. Function `Op` is higher-order: `Op` takes `Operand` and returns a function from `Mode` to a tuple.

```
(* Instruction formats *)
datatype Instruction =

    TwoOp1  of (TwoOpInstr1 * Operand * Operand)  |
    TwoOp2  of (TwoOpInstr2 * int  * Operand)  |
    OneOp   of (OneOpInstr  * Operand)            |
    Branch  of (BranchInstr * int)

(* Instructions allowed for each format *)
and  TwoOpInstr1 = ADD | MOV  | MOVB | SUB | CMP | CMPB
      ⋮

(* Operand formats and addressing modes *)
and  Operand = RegDirect        of int               |
               RegIndirect      of int               |
               AutoInc          of int               |
               Immediate        of int               |
               Indexed          of (int * int)       |
               AutoIncIndirect  of int               |
                    ⋮
```

Figure 1: Syntactic domain that specifies PDP-11 instruction formats.

For example, consider the auto-increment addressing mode. The valuation function `Op` is

```
Op⟦AutoInc(RegNum)⟧ Mode =
   let
    val r = Reg(Addr, RegNum)
   in
    (Mem(Mode, r),
     Assign(r,Add(Addr,r,Int(Int8,Sizeof(Mode)))))
   end
```

`Op` returns a tuple, the first item of which is how the operand is accessed and the second item is the side-effect of the auto-increment addressing mode. "`AutoInc(RegNum)`" is abstract syntax and `Reg`, `Mem`, `Assign`, `Add`, and `Integer` are semantic actions. The remainder of the PDP-11's addressing modes are similarly defined.

Instructions— A valuation function for an instruction constructs an `Imperative` by combining the actions of the operands with the action that represents the semantics of the instruction. The function `Instr` has type

`Instr` : `Instruction` → `Imperative`

For example, the PDP-11 `MOV` instruction is specified by the following valuation function, `Instr`.

```
Instr⟦(TwoOp1(MOV, DestMode, SrcMode))⟧ =
   let
     val (Dest, S') = Op⟦DestMode⟧ Int16
     and (Src, S'') = Op⟦SrcMode⟧ Int16
   in
    Sequence([
      Parallel[
        Assign(Dest,Src),
        Assign(Z,Eq(Int16,Src,Int(Int16,0))),
        Assign(N,Lt(Int16,Src,Int(Int16,0)))],
      Parallel[S', S'']])
   end
```

Two subtrees, S' and S", are constructed that represent side-effects from both operand accesses. These subtrees are combined with the subtree that represents the effect or meaning of the operation.

An instruction can alter the state in three ways.

- Through a side effect of the addressing mode.
- The primary instruction operation, in this case a register transfer representing the `Mov`.
- Setting condition codes.

The remainder of the PDP-11 instructions are similarly defined.

```
datatype
  Mode = CC | Addr | Int8 | Int16 | Int32 | Int64 | Float | DoubleFloat
and
  Value =
     (* Arithmetic operators *)
     Mem          of (Mode * Value)                    |
     Reg          of (Mode * int)                      |
     Add          of (Mode * Value * Value)            |
     Uminus       of (Mode * Value)                    |
     Int          of (Mode * int)                      |
          ⋮
     (* Comparison operators *)
     Compare      of (Mode * Value * Value)            |
     Eq           of (Mode * Value * Value)            |
     Cond         of (Value * Value * Value)           |
          ⋮
     (* Program counter and condition codes. *)
     PC | N | V | Z | C
and
  Imperative =
     Sequence   of (Imperative list) |
     Parallel   of (Imperative list) |
     Assign     of (Value * Value) |
     Call       of  Value |
     Nop | Return
```

Figure 2: Syntax of semantic actions that specify RTL.

3 Action Implementation

This section outlines a denotational semantics of the RTL semantic actions used in the previous section. The implementation of the semantic actions that describe the RTL only needs to be done once, the point being that once the RTL is implemented we can easily describe a variety of architectures.

We do not have room here to present the semantics of the actions but we will briefly outline the type signatures of the actions and the kinds of values that they operate on. We first define an abstract notion of *state*.

The State— Our state consists of memory (byte addressable), registers (32-bit), and a status register (*i.e*, condition codes; C, N, V, Z) that operate on the data types given by `Mode`.

```
Mode = Addr + Int8 + Int16 +
       Int32 + Int64 + Float + CC
State = Memory × Registers × StatusReg
```

Each of `Memory`, `Registers`, and the `StatusReg` are stores. An abstract notion of a store is a function from locations (addresses, register numbers, condition codes) to data. This suggests the following definition for the three stores:

$$\begin{aligned} \texttt{Memory} &= \texttt{Address} \rightarrow \texttt{Int8} \\ \texttt{Registers} &= \texttt{RegNum} \rightarrow \texttt{Int32} \\ \texttt{StatusReg} &= \texttt{Flags} \rightarrow \texttt{Bit} \end{aligned}$$

Action Signatures— The `Mem` action takes two parameters: a mode (`Mode`) that specifies how many bytes to fetch, and a value producing action (`Value`) that yields an address. `Mem` itself is a value producing action. Its signature is given by the following equa-

tion.

```
Mem: Mode × Value → Value
```

An example of an imperative action (`Imperative`) is `Assign`. `Assign` takes two value actions, the first of which must be an L-value that will be assigned the data the second action produces (the R-value). The signature of `Assign` is:

```
Assign: Value × Value → Imperative
```

```
Denotation = CC of Bit + Address of int +
    Int8 of int + Int16 of int ···
```

A value action is consequently a function from a state to a denotation. An imperative action is a function from a state to a state.

```
Imperative = State → State
     Value = State → Denotation
```

To be complete, we must specify the semantics of individual actions. We only note that there is an SML function for each of the actions in figure 2. For example, there is an SML function `Assign` that has the following signature.

```
Assign : Value × Value → Imperative
```

Simulation — SML is, essentially, an implementation of the call-by-value typed λ-calculus. The simulator for the architecture is the SML interpreter. The simulation should be viewed as computational; that is, we model the computation of the individual instructions as functions described in the λ-calculus.

4 Conclusions

An instruction set architecture is, in a sense, a programming language and can be treated as such. Using denotational semantics we have specified, formally, the semantics of an instruction set architecture which allows for the design of modular, readable, and usable architectural specifications. We have implemented the semantics using the functional language SML which makes our specification executable and automatically providing a simulator for the instruction set architecture.

We are currently developing a *purely* architectural specification language called LISAS[CFHM93]. LISAS descriptions are translated into the formal semantics presented in this paper. One problem with using a functional semantics is their difficulty in coping with *explicit* temporal and concurrent properties. Many modern RISC architectures have temporal constraints on the instructions that effect their meaning. Also, in Superscalar architectures instructions can be issued in parallel. We are currently investigating specifying the high-level timing and concurrent properties of an instruction set [HMC93].

References

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[BS89] G. Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[CFHM93] Todd A. Cook, Paul D. Franzon, Ed A. Harcourt, and Thomas K. Miller. System-level specification of instruction sets. In *ICCD 93, Proceedings of the International Conference on Computer Design*, 1993.

[Dav84] Jack W. Davidson. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):506–526, October 1984.

[FW88] Christopher Fraser and Alan Wendt. Automatic generation of fast optimizing code generators. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[HMC93] Ed Harcourt, Jon Mauney, and Todd Cook. Specification of instruction-level parallelism. In *Proceedings of NAPAW'93, the North American Process Algebra Workshop*, 1993.

[Lee89] Peter Lee. *Realistic Compiler Generation*. MIT Press, 1989.

[Sch86] David A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn and Bacon, 1986.