Directions in ISA Specification

Anthony Fox

Computer Laboratory, University of Cambridge, UK

Abstract. This rough diamond presents a new domain-specific language (DSL) for producing detailed models of Instruction Set Architectures, such as ARM and x86. The language's design and methodology is discussed and we propose future plans for this work. Feedback is sought from the wider theorem proving community in helping establish future directions for this project. A parser and interpreter for the DSL has been developed in Standard ML, with an ARMv7 model used as a case study.

This paper describes recent work on developing a domain-specific language (DSL) for Instruction Set Architecture (ISA) specification. Various theorem proving projects require ISA models; for example, for formalizing microprocessors, operating systems, compilers and machine code. As such, (often partial) ISA models exist for a number of architectures (e.g. x86, ARM and PowerPC) in a number of theorem provers (e.g. ACL2, PVS, HOL-Light, Isabelle/HOL, Coq and HOL4). These models differ in their presentation style, precise abstraction level (fidelity) and degrees of completeness. In part this reflects the nature of the projects for which the models have been originally developed, e.g. compiler verification [4] and machine code verification [7]. There are also differences based on the expressiveness and features of the theorem provers that are used. The ACL2 theorem prover has been used very successfully in this field for many years, where it has the advantage of providing very fast model evaluation. Recently, Warren Hunt has developed an ACL2-based specification of the Y86 processor, which implements a subset of the x86 architecture; see [3].

The main objective of the DSL is to make the task of modelling ISAs simpler, more reliable and less tedious. In particular, it should be possible for people who are not experts in HOL4 to readily read, develop and create ISA specifications for use in HOL4. Furthermore, it is also hoped that this work will help facilitate the dissemination of ISA models — enabling various concrete ISA models to be derived for different settings, tools and use cases.

Although various ISA DSLs currently exist, often these have been developed for writing compiler backends and binary code analysis tools, e.g. λ-RTL [9] and TSL [5]. The most closely related work is Lyrebird [1], which was developed as part of the seL4 project at NICTA. This tool supports fast simulation but it has not been successfully used in a theorem proving setting. Also of interest is the RockSalt project [6], which modelled x86 in Coq through the use of embedded DSLs. The aim of this work is to produce high-fidelity specifications that are inherently formal and yet prover/tool agnostic. The DSL and generated native prover specifications should be acceptable to both the engineering (computer architecture) and formal methods communities.

1 Language Design and Methodology

The design of the DSL has been influenced by our experiences in specifying the ARM architecture in HOL4, which is described in [2]. In particular, the DSL has been developed and tested through the production of a completely new version of the ARMv7 specification.[1] However, it is believed that the DSL is flexible enough to produce good models of other ISAs, such as the x86 architecture.

Methodology. The requirements for the DSL are based on our current specification approach, where we define:

- A *state space*. This represents all of the programmer visible registers, flags and memory. It may also include components that are not directly visible, such as static system configuration information (e.g. describing the architecture version and extension support) as well as any helpful shadow state components (e.g. the bit width of the current instruction).
- An *instruction datatype*. This provides an interface between instruction decoders and the instruction set semantics.
- A collection of definitions, specifying the semantics of each instruction class. This provides an operational (next state) semantics for each element of the instruction datatype.
- A *decoder*. This maps machine code values to the instruction datatype.
- An *encoder* (optional). This maps elements of the instruction datatype to concrete machine code values.
- A top-level next state function. This fetches an instruction from memory, decodes it and then applies the appropriate semantics definition for that instruction.

This approach has been implemented directly in HOL4 for the ARM, x86 and PowerPC architectures. However, there are areas where producing and maintaining ISA specifications in HOL4 is unduly tedious and potentially error prone.

The DSL improves upon native HOL4-based specifications in a few key areas:

- In HOL4 the state space is declared as a type, which means that all state components must be introduced early on and all in one go. It is also necessary to manually introduce collections of functions for accessing and updating state components and sub-components (e.g. named bit-fields).

  It is more natural to introduce state components in context, as and when they are needed. For example, within separate sections for the specification of machine registers and main memory. In the DSL state components are treated as global variables that may be declared anywhere at the top-level.
- The instruction datatype is also declared as a HOL4 type, which is somewhat tedious to specify and to maintain.

  The instruction datatype can be built incrementally, using the type signatures of the functions that define the instruction semantics.

[1] The new model actually covers the very latest incarnation of ARMv7, which adds support for a new "hypervisor" mode.

– Writing a good decoder is particularly challenging in HOL4. This is primarily because HOL4 does not provide direct support for matching over bit patterns. There is also the challenge of making the decoder evaluate efficiently, which currently requires some degree of HOL4 expertise.

  Matching over bit patterns is built into the DSL. We hope to support efficient evaluation for the generated HOL4 model.

These and other language features make it much easier to write ISA specifications in a natural style; making it possible to automatic generate HOL4 specifications that are otherwise hard or tedious to write manually.

Language Overview. The DSL is a first-order language with a fairly basic type system.[2] The intention is to keep the design of the DSL reasonably simple; shunning features that are not directly focussed on the ISA domain. This reduces the effort required to implement the language and it should help simplify the task of targeting models to different settings. Although the DSL is not particularly sophisticated, there were no problems in concisely specifying ARMv7.

Types. The primitive types of the language are: `unit`, `bool`, `string`, `nat`, `int`, `bitstring` and `bits(n)`, where `n` is either fixed or is constrained to a (possibly infinite) set of positive integers at the point of a function definition.[3] Type checking is implemented using Hindley-Milner inference, with some additional light-weight support for bit-vectors. Users can declare type synonyms, records, enumerations and non-recursive sum types. Constructors for product, map and set types are provided. For example, the following are valid declarations:

```
type reg  = bits(4)              -- type synonym (this is a comment)
type mem  = bits(32) → bits(8)   -- map

construct SRType -- enumerated type
{ SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX }

construct offset -- sum type
{  register_form  :: reg * SRType * nat  -- product
   immediate_form :: bits(32)  }
```

Syntax and Constructs. The DSL syntactically distinguishes between statements and expressions. Mutable values can be declared and updated in statements but not in expressions. There are *if-then-else*, *when-do*, *match-case* and *for-do* constructs. Function calls are strictly call-by-value but side-effects are possible, i.e. the global state can be updated. Exceptions can be declared and called, but not handled. A wide selection of primitive data operations are provided.

Users can define their own operations but cannot give these symbolic or infix/mixfix syntax. The following declaration defines n-byte word alignment:

```
bits(N) Align (w::bits(N), n::nat) = return [n * ([w] div n)]
```

[2] Recursive types, type polymorphism and dependent types are not supported.
[3] Floating-point support will be added in the future.

The operation '[·]' is used as a general casting map for primitive types. All types are inferred above using the function's arguments, which must be annotated.

Registers. The DSL supports declarations of register types with named bit-fields. The following declares a type for ARM's Programme Status Registers:

```
register PSR :: word
{  31: N  30: Z
   29: C  28: V      -- Negative, Zero, Carry, oVerflow flags
   27:    Q          -- Cumulative saturation flag
   15-10, 26-25: IT  -- If-Then
   24:    J          -- Jazelle bit
   23-20: RAZ!       -- reserved
   19-16: GE         -- Greater-equal flags (SIMD)
   9:     E          -- Endian bit (T: Big, F: Little)
   8:     A          -- Asynchronous abort disable
   7:     I          -- Interrupt disable
   6:     F          -- Fast interrupt disable
   5:     T          -- Thumb mode
   4-0:   M          -- Mode field  }
```

This introduces a new type `PSR` that corresponds with a 32-bit word. The named bit-field `M` is a 5-bit word, `N` is a Boolean flag and the special value `RAZ!` (read-as-zero) signifies an anonymous field. The expression `CPSR.IT` is equivalent to `&CPSR<15:10> : &CPSR<26:25>`; where the overloaded operator '`&`' maps registers to their bit-vector values, '·<·:·>' is bit-field extraction and '`:`' is bit-vector concatenation. In the HOL4 model [2], PSRs are defined using a record type and encoding/decoding functions are manually defined. It is now relatively easy to automatically generate these types and functions for each of ARM's system registers, saving users time and effort.

State. Global state components are declared as follows:

```
declare CPSR :: PSR
declare MEM  :: bits(32) → bits(8)
```

These components can be updated with various assignment forms, for example:

```
CPSR.N ← true;       &CPSR<31> ← true;         CPST.M ← '11010';
&CPSR ← 0x11;        &CPSR<31:28> ← '1101';    CPSR ← PSR(0x11);
MEM(4) ← &CPSR<15:8>
```

The dot syntax also applies to conventional record types. Users can define their own update operations; for example, consider the following declaration:

```
component NZCV :: bits(4)
{ value = &CPSR<31:28>
  assign v = &CPSR<31:28> ← v  }
```

This declaration makes it easy to access and modify the NZCV status flags. The component construct is particularly useful for declaring operations that access registers and memory. For example, one can specify:

```
NZCV ← NZCV && '0101';
R(12)<15:0> ← MemU(address, 2);
MemU(address + 2, 2) ← R(12)<31:16>
```

where the operations `R` and `MemU` provide an interface to the general-purpose registers and memory. Note that the physical register corresponding with the argument 12 actually depends on the current processor mode, given by `CPSR.M`; and memory accesses are affected by the endian bit `CPSR.E`.

Instruction Specification. The following DSL code specifies the semantics of the ARM instruction BLX (register):

```
define Branch > BranchLinkExchangeRegister ( m :: reg ) =
{  target = R(m);
   if CurrentInstrSet() == InstrSet_ARM then
   {  next_instr_addr = PC - 4;
      LR ← next_instr_addr
   } else
   {  next_instr_addr = PC - 2;
      LR ← next_instr_addr<31:1> : '1'
   };
   BXWritePC (target)
}
```

This declaration extends an *abstract syntax tree* (AST) datatype `instruction`. A primitive operation `Run :: instruction → unit` runs the code associated with the given AST. The `>` notation allows instructions to be grouped into a hierarchy of instruction categories.

Instruction Decoding. A decoder is any function that takes the output of an instruction fetch and returns values of type `instruction`. Users are free to define such functions in any way that they see fit. A natural choice for decoding ARM instructions is through pattern matching over bit patterns. The ARMv7 decoder is approximately four thousand lines of code (including comments), with 233 top-level cases. Missing and redundant patterns are reported, which is essential in this context. Below is a code snippet relating to the BLX instruction:

```
instruction DecodeARM (w::bits(32)) =
  match w
  { ...
    case 'cond : 00010010 : (111111111111) : 0011 : Rm' =>
    if Take (cond, ArchVersion() >= 5) then
    {  when Rm == 15 do DECODE_UNPREDICTABLE (mc, "BLX (register)");
       Branch (BranchLinkExchangeRegister (Rm))
    } else Skip ()
    ...
  }
```

Bit patterns are surrounded by apostrophes. Bracketed bit fields are "should-be" tokens — they match *any* field of the appropriate length. The bit-widths of variables can be annotated or given default values: `cond` and `Rm` were declared as 4-bit

values. When an op-code is not valid the user function `DECODE_UNPREDICTABLE` is called, which raises a suitable exception. The user defined functions `Take` and `Skip` take care of conditional (no-op) and undefined instructions.

2 Directions

At the time of writing the new DSL has a parser and an evaluator/interpreter. Good progress has been made on exporting to HOL4 — the initial objective is to generate a first-order functional specification for ARMv7. One of the advantages of having a custom DSL is that different models can be generated from the same source. For example, it should be possible to generate deeply embedded models or monadic specifications similar to those already in HOL4. One key objective is to generate "evaluation friendly" code, i.e. to support fast evaluation for machine code verification (see [2]).

The back-end model export should be readily configurable, facilitating translations into various other settings, such as different theorem provers (ACL2, Coq, HOL-Light and Isabelle/HOL), high-level languages (e.g. SML and Bluespec) or OpenTheory.[4] This would enable the same model to be used in wide range of projects, potentially avoiding duplicated effort. In this regard our objective is similar to that of Lem [8]. A key challenge will be to generate satisfactory models (meeting the requirements of end-users) under each setting. Our main priority is to improve our own, well-understood HOL4 workflow. In targeting other theorem provers, dialogue and collaboration with other end-users will be essential.

Acknowledgments. Many thanks to Magnus Myreen, Mike Gordon and Peter Sewell for providing motivation and stimulating discussions.

References

1. Cock, D.: Lyrebird: assigning meanings to machines. In: SSV'10. (2010)
2. Fox, A.C.J., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: ITP 2010. Number 6172 in LNCS, Springer (2010)
3. Hunt Jr., W.A.: X86 specification in ACL2. `http://www.cs.utexas.edu/~hunt/research/y86/`
4. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009)
5. Lim, J., Reps, T.: A system for generating static analyzers for machine instructions. In: Compiler Construction. (2008)
6. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: Better, faster, stronger SFI for the x86. In: PLDI'12. (2012)
7. Myreen, M.O.: Verified LISP implementations on ARM, x86 and PowerPC. In: TPHOLs 2009. Number 5674 in LNCS, Springer (2009)
8. Owens, S., Böhm, P., Nardelli, F.Z., Sewell, P.: Lem: A lightweight tool for heavyweight semantics. In: ITP 2011. Number 6898 in LNCS, Springer (2011)
9. Ramsey, N., Davidson, J.W.: Machine descriptions to build tools for embedded systems. In: LCTES'98. Number 1474 in LNCS, Springer (1998)

[4] `http://www.gilith.com/research/opentheory`