Specification and Simulation of Microprocessor Operations and Parallel Instructions

Loe Feijs
Eindhoven University of Technology
L.M.G.Feijs@Tue.nl

Paul Gorissen and Joachim Trescher
Philips Research Laboratories Eindhoven
{Paul.Gorissen,Joachim.Trescher}@Philips.com

Abstract

In this paper, we report on the development of a language which is especially tailored to the specification and simulation of microprocessor operations and parallel instructions. The approach is rigorous, and it combines the naturalness and readability of the traditional pseudocode with the formality and rigour of instruction specifications in the programming language C (but without the disadvantages of the latter). The underlying semantic model has been formalized by the equations of an appropriate denotational semantic model. The specifications can be used for a variety of purposes, such as the generation of a data book and other on-line documentation, the generation of a simulator that allows functional testing of programs even before the hardware has been designed and implemented, and the generation of a test suite to perform functional tests of a given design or real chip.

1. Introduction and motivation

There is a tendency to replace application specific integrated circuitry (ASICs) for digital signal processing by programmable digital signal processors (DSPs). This tendency is motivated by the reduced debugging cost and effort, and the reduced architectural complexity of the chip, resulting in a decreased time to market. In an endeavour to meet the stringent performance and cost requirements of many application domains, DSP architectures have tended to become involved, and, as a consequence their instruction sets are irregular or even application specific (ASIPs). As a consequence, software development tools are not readily available, and software development can only start later in the system development process. When that software development constitutes a large part of the overall system development, or is even part of the critical path, the unavailability of software development tools could cancel out the time to market advantage that DSP solutions have over ASICs.

The specification of the instruction set architecture (ISA) must be available as one of the initial documents in a classic DSP based system development process, since it provides the necessary interface definition that allows concurrent software and hardware development. An ISA specification to be used for this purpose must be explicit and unambiguous. A single ambiguity or mis-interpretation will inevitably lead to programming errors, incompatibilities between different designs of the same processor, and erroneous compiler back-ends. Thus there is an urgent need for explicit and unambiguous specifications. The typical subtleties to be addressed by ISA specifications include the calculation of addresses, the embedding of flagging bits in registers, and the interference of parallel calculations.

As soon as an ISA specification exposes sufficient detail of the operational semantics of the specified DSP, it is possible to generate software development tools such as (dis)assemblers, simulators, debuggers, and non-optimized compilers. The problem in deploying application specific processors described above is then eased.

The traditional specification techniques used for this purpose are: (1) pseudo code, enriched by diagrams and natural language texts, and (2) models in programming languages such as C. The second technique is more formal than the first one; the specifications are machine-readable, which is an important advantage for simulator and test generation. The C programs tend to be intricate and hard to read, however. This is partly due to the low abstraction level, and partly because C offers no built-in mechanisms for dealing with the parallelism in an elegant way.

Since a new language is being proposed, it should be justified why it has to be a new language and in particular why the language is more advantageous than VHDL for the task at hand. The proposed language is tuned to the specific task of specification and simulation of micro processor operations and parrallel instructions. Simulation means that the application programmers of a not-yet-existing processor can start testing their programs as soon as the instruction set is defined (it does not mean simulating its implementation at gatelevel). The advantages are twofold:

- structure: users of the language are helped by the built-

in two-level structions; inside the operations and parrallel instructions; inside the operations it is useful to have separate sections for definitions and statements;
- compactness: users of the language are helped by the special operations(e.g.#) which allows for descriptions that are as short as possible.

In fact, almost any general purpose specification language or programming language could be used, for example VDM, VHDL, C++, Java, but each of these fails with respect to the abovementioned advantages for this specific application domain.

In this paper, we describe a rigorous approach, introducing a language which is especially tailored to the task of specifying the ISA of processors that display characteristics of RISC, CISC, DSP, and VLIW architectures. The syntax of the language is formalized and we have built a parser and other tools for automated processing of the specifications.

2. Related work

The need for fast and accurate simulation, as motivated and explained in Bedicheck [2], has created a wealth of literature on this subject. For didactic purposes we split the existing literature on simulation into three categories according to the objective of the described approach: (1) The first category is mainly concerned with the speed of simulation. The published approaches draw from the rich body of compiler construction techniques, such as partial evaluation and dead code elimination, to speed up the execution of a simulator; (2) the main objective of the second category is to create a simulation that is sufficiently accurate to serve as a platform for design space exploration. These simulators typically co-simulate parts of the environment of a microprocessor, such as the memory subsystem or external devices, to achieve their goal; (3) the creation of basic software development tools for a new programmable device is labour intensive, error prone, and time consuming. The focus of the third category is to automatically generate these tools from an abstract description of the programmable device.

Speed: In [1], Bedichek describes a simulator for the Motorola 88000 at the instruction set architecture level. He introduces the technique of translating instructions into a quick-to-execute form. Decoded instructions are cached and this has the effect of greatly improving the speed of simulations. This is one of the techniques which we also adopt in our own work.

May [5] combines flow analysis and block-wise translation to develop a fast simulator for the System/370 processor.

Sykes and Malloy [6] introduce the technique to separate the updating of the program counter from updating the state vector. In their terminology, the *decode* step of processor simulation is incorporated into the state of each instruction object. As a consequence, each instruction is decoded only once during simulation. We have also included this speed improving technique in our work.

Cmelik and Keppel [8] describe a simulator called Shade, which is used for simulating SPARC and MIPS instruction sets. The system can be used to generate traces in the same way that we do. The level of detail of the tracing can be adjusted in various ways in order to control the overhead involved. To improve speed, code is expanded and cached for reuse. There is also a flexible scheme of saving and reusing condition flags: if Shade can determine that the next application code will always set the condition flags before using them, it will not save them in the translation epilogue.

Živojnović et al. [4] describe further advancements with respect to simulation speed, amongst other techniques, achieved essentially by running the simulation as compiled C code. Magnusson [20] introduces partial translation, a hybrid approach that combines the advantages of block translation and interpretive translation.

It should be noted that the approaches described above are dedicated simulators, whereas in our approach, we generate the simulators from high-level descriptions in an automated way.

Accuracy: Witchel and Rosenblum [3] present a simulator for the MIPS R3000/R4000 machine, which is faithful enough to run a commercial operating system. They introduce the technique to allow simulation models for hardware devices which can be called during execution of a code simulation.

Herrod [7] uses the Embra [3] approach; he is amongst the first who are able to simulate a task as big as the booting of an entire operating system. He also proposes techniques to make calls to real peripheral hardware from the simulator.

Rosenblum et al. [17] discuss SimOS, which is particularly flexible in the trade-off between the speed and detail of the simulation. SimOS also simulates the memory, the memory management unit (MMU), and the related protection mechanisms.

Anderson [19] presents the design of a simulator for Motorola's PowerPC. He shows that there is a need for a family of simulators, including tools such as an architectural simulator, a timing simulator, and a functional simulator.

Magnusson and Werner [21] present an approach called SIMICS, which is concerned with simulations of all aspects of memory, including multiprocessors and memory hierarchies. The power of this approach is demonstrated by the SimICS/sun4m virtual workstation [22], which runs large benchmarks and applications. It is argued that this type of simulator should be viewed as an instrument for computer architecture studies and performance debugging tasks. The

cycle rate	debug hours	debug time	technology
1	1	1 hour	silicon reference design
10^{-1}	10	1 day	FPGA
10^{-2}	10^2	4 days	HW emulator
10^{-3}	10^3	1.4 months	throughput model
10^{-4}	10^4	1.2 years	bit-true simulation
10^{-5}	10^5	12 years	cycle-true simulation
10^{-6}	10^6	$>$ lifetime	RTL model
10^{-7}	10^7	millenium	gate level model

Table 1. Time required to obtain the equivalent of one hour of testing experience on different platforms[18]

same point is made in [23].

In [16], Teich and Weper introduce a tool set for the automatic generation of efficient yet cycle-accurate and bit-true simulators from a graphical entry of the processor building blocks. This approach uses Gem-Mex [15] to automatically generate a debugging and simulation environment. While the ambitions of this project are intriguing, it will be interesting to see which techniques the authors employ to achieve the claimed high simulation speeds.

Tool generation: Larsson [25] introduces the concept of a generic simulator. As in our own work, he uses a specification file describing the instruction set architecture. There is a pattern, a syntax and semantics for each instruction.

The hardware description language nML [10] has spawned two approaches that offer a retargetable software development environment, namely the CBC/SIGH/SIM framework [11] and the CHESS/CHECKERS environment [9]. In contrast to Larsson's work, nML provides a concise and formal notation to describe the behaviour of fixed point DSP processors. However, both specification mechanisms are awkward to use when not all pipeline effects are hidden to the programmer by an interlocking pipeline.

3. Architecture and levels

In this section, we discuss the objectives and requirements that influenced our design decisions. Our main goal is to create a description formalism that allows us to specify the ISA of microprocessors. This description formalism must be expressive enough to describe the ISAs of a broad spectrum of microprocessor architectures, ranging from RISCs, DSPs, and ASIPs to VLIWs. Furthermore, the descriptions must have sufficient detail so that it is possible to generate efficient simulators and other basic software development tools.

Table 1 details the time required to obtain the equivalent of one hour of testing experience for a wide range of prototype platforms. We note that a simulation environment used for the early development and testing of software is only feasible if the efficiency of the simulation environment approaches that of a hardware emulator. The decision to simulate processors on the level of the ISA is mainly motivated by this observation.

It is important to understand the architectural concepts that influence the programmer's view of a core, in order to render the specification of ISAs for a broad range of microprocessors feasible. Today, almost all microprocessors use pipelining to increase their efficiency. Furthermore, in the domain of DSPs and ASIPs, we also observe that the performance requirements demand that multiple operations are executed in parallel, which leads to the design of multiple issue processors.

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution [24]. Pipelining increases the CPU instruction throughput (i.e. the number of instructions completed per unit of time) but it does not reduce the execution time of an individual instruction. In pipelined architectures, there are situations called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. We distinguish three classes of hazards: (1) *structural hazards* arise from resource conflicts when the hardware can not support all possible combinations of instructions in simultaneous overlapped execution; (2) *data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline; and (3) *control hazards* arise from the pipelining of branches and other instructions that change the program counter.

Eliminating a hazard often requires some instructions in the pipeline being allowed to proceed while others are delayed. When an instruction is *stalled*, all instructions issued later than the stalled instruction are also stalled. Instructions issued earlier than the stalled instruction must continue, since otherwise the hazard will never clear.

Processor architects have the choice of either hiding all hazards from the programmer by an interlocking pipeline, or leaving those pipeline effects visible to the programmer. For example, the designers of the Pentium [12] and R.E.A.L. [13] chose an interlocking pipeline, whereas the MIPS [26] architects have chosen a hybrid approach, where data hazards are hidden from, and control hazards are visible to, the programmer. In contrast, the designers of the TriMedia [14] prefered a solution where all pipeline effects are visible to the programmer. As a consequence, a description formalism for ISAs must provide the means to describe a temporal ordering on operations performed during the execution of a single instruction. It is for this reason that we

introduce the ',' operator in Sect. 6, and define its semantics in Sect. 7.

The goal of multiple-issue processors is to increase the performance of a processor by issuing several operations every clock cycle. These operations are then executed in parallel by different functional units. Ideally, these operations should execute independently and do not have any side effects. In practice, however, we observe different approaches. For instance, the design of the R.E.A.L. allows the execution of up to eight operations in parallel, which all read operands from and write results to shared resources. All of those operations might have side effects that could change the state of the global status and control register. We introduce the $\|$ operator to allow an unambiguous specification of instruction parallelism in an ISA. Its semantics are defined in Sect. 7.

4. Specification language

We consider a language of *operations* and *parallel instructions*. The operations are essentially simultaneous register assignments, except for a few details that are needed for handling shared sub-expressions and aliases. The 'parallel instructions' are essentially patterns of sequential and parallel composition in a clocked timing grid. We are aiming at a formalization of the syntax and semantics of operations and parallel instructions. This will have the advantage that we can perform transformations on (and perhaps even equational reasoning about) operations and parallel instructions. This is useful, as we sometimes want to do more computation work in parallel or conversely, to remove parallelism.

Syntactically, an instruction of a microprocessor is described by a list of definitions, followed by a list of assignments. This reflects the temporal ordering of actions performed during the execution of an instruction: first, values are read from designated locations (registers, memory), then the results of specific operations are written back to designated locations potentially overwriting values read in the first phase. An example of a definition is $NewPC = PC + 2$, which introduces the name NewPC and defines its value as the result of the expression $PC + 2$, where PC is a name defined in the context. An example of an assignment is SP := SP + 1, where SP is a register, in this case the stack pointer register.

Some care is needed due to the phenomenon of aliasing as we will explain below. We could define an instruction that performs the following operations, for example:

IEB := 0 # SSR := 0xFFFFFFFF

where IEB is the Interrupt Enable Bit and SSR is the Source and Status Register. Now let us assume that IEB is the same as bit three of the SSR. Then the order in which the assignments are done really matters. In this case, we want the assignment IEB := 0 to take priority. So we should think of the # operator as a kind of parallel assignment which guarantees that the result of the textually first assignment takes priority over the second assignment. The effect is that, after execution of the parallel assigment, SSR contains FFFFFFF7, which is what we want.

5. Syntax of operations

We consider registers with typical elements r, r_1, r_2, etc., variables with typical elements v, v_1, v_2, etc., expressions with typical elements e, e_1, e_2, etc., operations with typical elements o, o_1, o_2, etc., and statements with typical elements s, s_1, s_2, etc. We assume constants c, unary operations $\mathcal{O}_1$ and binary operations $\mathcal{O}_2$. The set of *operations* is then generated by the following rules:

$$
\begin{array}{rl}
o ::= & \text{nop} \\
 \mid & v_1 = e_1 \text{ ; ... } v_m = e_m \text{ ; (} s_1 \text{ \# ... \# } s_n \text{)} \\
s ::= & \text{nop} \\
 \mid & r := e \\
 \mid & \text{if } e_1 \text{ then } r := e_2 \text{ fi} \\
e ::= & c \\
 \mid & v \\
 \mid & r \\
 \mid & \mathcal{O}_1\ e_1 \\
 \mid & e_1\ \mathcal{O}_2\ e_2
\end{array}
$$

The language elements generated by o are called operations, the language elements generated by s are called statements, and the language elements generated by e are called expressions. The operator # denotes priority inside the operation. We adopt the following additional well-formedness requirements:

- the variables v_1 ... v_m must all be different variables,
- the variables may be used only after introduction, e.g. v_2 can occur in $e_3 \ldots e_m$ and also in the expressions inside the statements, but not in e_1 or in e_2.

The example operation below works on a set of registers P = { PC, X, Y, C, N, Z, O, LOF }. The set of variables is L = { sx, sy, sr, r, o1, new_pc }. We assume constants 0, false, etc., and certain hexadecimal constants (e.g. 0xFFFFFF). We assume unary operators signed(n,_) for natural numbers in addition to binary operators _+_, _\/_, _&_, _<_, _>_ and _==_.

new_pc = PC + 2
; sx = signed(24,X)
; sy = signed(8,Y)
; sr = sx + sy

```
; r = sr&0xFFFFFF
; o1 = sr>0x7FFFFF ∨ sr<0xFF80000
; ( PC := new_pc
  # X := r
  # C := sr>0x7FFFFF ∨ sr<0xFF80000
  # N := sr<0
  # Z := r==0
  # O := o1
  # if o1 then LOF := false fi
  )
```

The operation as a whole could be the detailed description of an addition operation with the intended rough interpretation that X ← X+Y. Here, C is a *carry bit*, N is a *negative bit*, Z is a *zero bit*, O is an *overflow bit*, and LOF is a so-called *line overflow bit*. Of course we could employ some additional syntactical sugar, such as making the definition of o1 local to the last two assignments.

We can say that the language of operations gives us all the expressive power needed in practice to specify the functional behaviour of each machine instruction of a typical digital signal processor or a media processor. The parallel instructions of the next section are only needed if we want to describe what happens at the hardware level more faithfully.

6. Syntax of parallel instructions

We consider operations with typical elements o, o_1, o_2, etc., in addition to instructions with typical elements i, i_1, i_2, etc., and parallel instructions with typical elements p, p_1, p_2, etc. The set of *parallel instructions* is then generated by the following rules

$$
\begin{array}{lll}
i ::= & \{\ o_1, \ldots, o_k\ \} & \\
p ::= & i & \\
\mid & i_1 \parallel i_2 & \\
\mid & i_1 \parallel i_2 \parallel i_3 & \text{(etc.)} \\
\mid & i_1 \parallel_{\neg} i_2 & \\
\mid & i_1 \parallel_{\neg} i_2 \parallel_{\neg} i_3 & \text{(etc.)}
\end{array}
$$

Note, that we have chosen this rather awkward language definition instead of a recursive one with the aim of easing the definition of its denotational semantics. In the following, we will use the convenient vector notation $\vec{i_1}$, $\vec{i_2}$, etc., where each vector is a syntactic sequence of operations of the form $i ::= \{\ o_1, \ldots, o_k\ \}$. This avoids clumsy notations based on double indices.

The operator ',' denotes sequential composition of operations. Each such sequence is called an *instruction*. The operator $\parallel$ denotes parallel composition of instructions (the intuitive interpretation is that the instructions are executed in a step-wise parallel way). The operator $\parallel_{\neg}$ denotes pipelined parallel composition of instructions (the intuitive interpretation of this operator is that both instructions are executed, again in a step-wise parallel way, but such that an instruction is always one phase ahead of the next instruction).

If $i = \{\ o_1, \ldots, o_k\ \}$, then we say that k is the *length* of instruction i. We adopt the following additional well-formedness requirements:

- if p is obtained as the parallel composition of i_1, i_2, etc., then the length of all instructions i_1, i_2, etc., must be the same,
- the registers affected by the first operations of i_1, i_2, etc., must be mutually disjoint; this is also true for the registers of the second operations and so on until the registers of the last operations.

As an example of the second requirement, { X:=0 , Y:=1 } ∥ { Y:=0 , X:=1 } is correct, { nop , X:=0 , Y:=0 } ∥ { X:=1 , Y:=1 , nop } is correct but { X:=0 , Y:=0 } ∥ { X:=1 , Y:=1 } is not correct. Omitting this requirement would give rise to resource conflicts (the kind of conflicts which now lead to programmer exceptions). Our task is to define tools to analyze parallel instructions in order to detect such resource conflicts.

We consider $i \parallel_{\neg} p$ to be a derived operator in the sense that we define its well-formedness and its semantics via $\parallel$. For example, if $\vec{i_1}$ and $\vec{i_2}$ are two sequences of operations, such that the two sequences have equal length, then we put:

$$\left\{\ \vec{i_1}\ \right\} \parallel_{\neg} \left\{\ \vec{i_2}\ \right\} = \left\{\ \begin{array}{c} \vec{i_1} \\ \text{nop} \end{array}\ \right\} \parallel \left\{\ \begin{array}{c} \text{nop} \\ \vec{i_2} \end{array}\ \right\}$$

The idea is that we should apply this procedure recursively. We write nop^{n-1} for a sequence of $n-1$ nop operations.

$$\left\{\vec{i_1}\right\} \parallel_{\neg} \left\{\vec{i_2}\right\} \parallel_{\neg} \cdots \parallel_{\neg} \left\{\vec{i_n}\right\} = \left\{\ \begin{array}{c} \vec{i_1} \\ \text{nop}^{n-1} \end{array}\ \right\} \parallel \left\{\ \begin{array}{c} \text{nop} \\ \vec{i_2} \\ \text{nop}^{n-2} \end{array}\ \right\} \parallel \ldots \parallel \left\{\ \begin{array}{c} \text{nop}^{n-1} \\ \vec{i_n} \end{array}\ \right\}$$

So if we have n instructions of the same length, say l, we can compose them via $n-1$ applications of $\parallel_{\neg}$ and then find that the result is a parallel instruction all of whose instructions have length $l + (n-1)$.

7. Formal semantics

An obvious idea would be to model the semantics of operations analoguous to the two phase operational semantics of a hardware implementation, where one phase takes a snapshot of the current state (i.e., read phase) and the second phase performs a state change (i.e., write phase). However, we were able to define the semantics in a more abstract and

compositional manner. We define a *configuration* to be a mapping from variables and registers to values. The meaning of an operation will be a special kind of mapping from configurations to configurations.

As a kind of auxiliary meaning function we need a mapping $[\![\ ]\!]$ from expressions and configurations to values; the standard mapping is $[\![v]\!](s) = s(v)$, $[\![r]\!](s) = s(r)$ and $[\![c]\!](s) = c$ and $[\![\mathcal{O}_1 e_1]\!](s) = \mathcal{O}_1([\![e_1]\!](s))$ and $[\![e_1 \mathcal{O}_2 e_2]\!](s) = \mathcal{O}_2([\![e_1]\!](s), [\![e_2]\!](s))$. Here we assume that there is a corresponding semantic constant (or operator) for each syntactic constant (or operator), here also denoted by the same symbol.

Next we ought to give the semantics for the operations. For s, a configuration whose domain is precisely the set of all registers, the meaning $[\![o]\!](s)$ will be defined as a certain result configuration with the same domain. We put $[\![v_1 = e_1; ...v_m = e_m; (r_1 := e_{m+1} \# \ldots \# r_n := e_{m+n})]\!](s) = s_m\{[\![e_{m+n}]\!](s_m)/r_n\}...\{[\![e_{m+1}]\!](s_m)/r_1\}$ where s_m is defined as the last element of the following sequence:

$$
\begin{aligned}
&s_0 = s,\\
&s_1 = s_0\{[\![e_1]\!](s_0)/v_1\},\\
&\vdots\\
&s_m = s_{m-1}\{[\![e_m]\!](s_{m-1})/v_m\}.
\end{aligned}
$$

We still have to deal with the if-then-fi construction but we do not expect this not to be a real problem (at least, if we know how to interpret values as Booleans).

The semantics of $\|$ are given by saying formally that the instructions involved are executed in a step-wise parallel way:

$$\{o_1, \ldots, o_k\} \parallel \{o'_1, \ldots, o'_k\} = \{o_1 \# o'_1\ ,\ \ldots\ ,\ o'_1 \# o'_k\}$$

and, analogously, for two or more applications of $\|$. In this way, we can reduce all applications of $\|$ and we are only left with the problem of defining semantics for a ‘,’-separated list of operations (recall that we know the meaning of $\#$ already, it is the priority operator, although the well-formedness requirements guarantee that the priority no longer matters). Defining semantics for a comma-separated list is easy. We put

$$[\![\{o_1, ..., o_k\}]\!](s) = [\![o_k]\!](...[\![o_2]\!]([\![o_1]\!](s))...)$$

which says that o_1 is applied to s first, followed by o_2 etc. Now that we have a meaning for $\|$, we also have a meaning for $\|_{\_}$ due to the way in which we defined the latter in terms of the former earlier in 6.

8. Implementation

The specification language based on the principles explained in this paper, as developed at Philips Research Laboratories, has been christened “ArchitEXt”. This name reflects the three most important aspects of the language: (1) it is about instruction set *archi*tecture; (2) it supports detailed *text*ual descriptions of instruction sets; (3) the specifications are *ex*ecutable.

The expressive power is sufficient to describe complex DSPs such as the first and second generations of the R.E.A.L. processors, which have been completely specified at a functional level. The generated simulator is used by software developers and compiler builders as a development platform and by hardware developers to test their designs.

We have built a generator that reads an ArchitEXt specification and translates it into an efficient Java program. In this way, we obtain a platform-independent simulator. It is not possible to give a full description of the generator due to space limitations. The following bullets we summarise the most important design and implementation decisions:

- the generator has been written in C using standard tools to build a parser for the ArchitEXt language;
- the expression language of ArchitEXt has been adopted from Java. This improves the readability and helps to avoid semantic problems in this part of the language;
- instructions can have two forms: undecoded and decoded. The first time an instruction is executed by the simulator it is decoded and then the decoded instruction replaces the original form, in situ. This technique has been adopted from Bedichek [1].

A drawback of our approach is the large memory requirement of the generated simulator. The simulation of a moderately sized DSP program with 1 M-byte program memory and 360 K-byte data memory currently requires about 300 M-byte of memory on the host platform. This is because a number of speed improvements have been achieved at the cost of introducing efficient but space-intensive data structures.

With respect to execution speed, we measured 800 k-instructions per second on a 550 MHz Pentium III Windows NT platform. Comparing the simulation speed to the speed of a hardware implementation of 130 MHz we achieve a simulation ratio of 160 : 1, which we consider surprisingly good for a simulator implemented in Java. We also hope to improve the performance of the generated simulators by introducing just-in-time compilation.

9. Example

In this section we illustrate the language ArchitEXt with some examples which illustrate its expressive power. In an ArchitEXt description of an ISA each instruction is described by one or more entries. Since an ArchitEXt specification is developed for multiple purposes, each entry combines several types of information using different sections

similar to tags in HTML. We distinguish a textual section (useful for the generation of a databook), a format section (useful for the generation of a decoder), and a semantic section (useful for the generation of an interpreter).

Fig. 1 shows the ArchitEXt description of an ARM data processing instruction performing a arithmetic or logical operation (in our case an ADD). The first operand is always a register the second operand maybe a shifted register or a rotated 8-bit immediate value. The textual section consists of a book tag which is used to organise the data book and a description tag used to hold an informal description of the instruction. The format section contains information about the binary encoding of the instruction and its size. The semantics section consists of a binding tag that links the parts of the binary encoding of an instruction which specify registers, offsets, or immediate values to the definition of its semantics.

```
<instruction> 1
<book>
   <title> Arithmetic </title>
   <title> ADD </title>
</book>
<description>
   Adds the value of register Rn and imm.
   The result is stored in register Rd.
</description>
<size> 16
<format> 0001ddddnnnniiii
<binding>
   (int rd = dddd, int rn = nnnn,
   int imm = iiii)
</binding>
<semantics>
   reg(rd) = reg(rn) + imm
</semantics>
```

Figure 1. ADD instruction of the ARM 7TDMI

The example in Fig. 1 is a quite simple example that illustrates the basic language elements of ArchitEXt. To illustrate the strength of ArchitEXt we use one of the exceptions to the basic data processing instruction which is described by the ARM databook as follows:

"*When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. [...] If R15 (the PC) is used as an operand in a data processing instruction the register is used directly. The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.* "

The ArchitEXt entry for the ARM ADD instruction that uses the PC as a operand and destination register is illustrated in Fig. 2. Although, the informal description given in the ARM databook contains sufficient detail to discern the exact semantics of this particular instance of and ARM addition we feel that the ArchitEXt description is more concise and definite. Also the concept of atomicity is neatly embedded in the language model of ArchitEXt.

```
<instruction> 1
<book>
   <title> Arithmetic </title>
   <title> ADD </title>
   <title> Destination PC </title>
   <title> First operand PC </title>
   <title> Second operand immediate </title>
</book>
<description>
   Adds the value of register R15 and imm.
   The result is stored in register R15.
   SPSR is moved to CPSR as a side effect.
</description>
<size> 32
<format> ffff0010100111111111rrrriiiiiiii
<binding>
   (int ra=(rrrr << 1), int imm=iiiiiiii)
</binding>
<semantics>
   int op1=reg(rn) + 8;
   int op2=rot(uns(32,imm),uns(5,ra));
   (reg(rd)=op1 + op2;
   # cpsr=spsr
   )
</semantics>
```

Figure 2. ADD instruction of the ARM 7TDMI processor using the PC as operand and destination register

10. Concluding remarks

In this paper, we introduce a set of essential language elements for describing the various kinds of parallelism in ISAs. Naturally we also have the additional language elements for specifying the precise opcode and the layout of its operands. These have not been formally specified, but they are processed by the tools in the classical way. We also observed that it is not possible to describe processors whose implementations employ interlocked pipelines in a way that allows a cycle-true simulation to be derived. One option for future work is to introduce an operator that formalizes a stalling pipeline.

References

[1] R. Bedichek. Some efficient architecture simulation techniques, Proceedings of the Winter 1990 USENIX Confer-