# Fast Specification of Cycle-Accurate Processor Models*

Felix Sheng-Ho Chang and Alan J. Hu
Department of Computer Science, University of British Columbia
(fchang,ajh)@cs.ubc.ca

## Abstract

*This paper introduces a new specification style for processor microarchitectures. Our goal is to produce very simple, compact, but cycle-accurate descriptions, in order to enable early exploration of different microarchitectures and their performance. The key idea behind our approach is that we can derive the difficult-to-design forwarding and stall logic completely automatically. We have implemented a specification language for pipelined processors, along with an automatic translator that creates cycle-accurate software simulators from the specifications. We have specified a pipelined MIPS integer core in our language. The entire specification is less than 300 lines long and implements all user-mode instructions except for coprocessor support. The resulting, automatically-generated, cycle-accurate simulator achieves over 240,000 instructions per second simulating MIPS machine code. This performance is within an order of magnitude of large, hand-crafted, cycle-accurate simulators, but our specification is far easier to create, read, and modify.*

## 1. Introduction

Cycle-accurate processor simulators are crucial for making intelligent microarchitectural trade-offs early in the design cycle. Unfortunately, writing a cycle-accurate simulator for a given microarchitecture is time-consuming and error-prone, limiting the amount of design exploration possible. In this paper, we introduce a novel specification style tailored expressly for describing processor microarchitectures. The specification style allows the designer to think in terms of pipeline structure, functional unit latency, and whether or not a bypass path is provided. An automated tool can then infer all of the bypass and stall control logic, which is normally one of the most time-consuming parts of a cycle-accurate processor model to design.

To test the usefulness of our approach, we have embodied our specification style in a simple language to describe pipelined processors. We have also written an automatic translator that reads descriptions in this language and generates a cycle-accurate simulator for the processor. The resulting simulator executes machine code for the processor being simulated. The key questions to answer are (1) whether the specification style enables simpler, more compact descriptions, and (2) whether the generated simulator is fast enough to be used for meaningful performance evaluation. If the answer to both questions is affirmative, then we have a useful tool for early-stage microarchitectural design exploration.

### 1.1. Related Work

We emphasize the distinction between an instruction set simulator, a cycle-accurate simulator, and a logic simulator. An instruction set simulator provides the highest simulation speed (currently, many millions of instructions per second), but does not model at all the processor's implementation or performance. A logic simulator provides the slowest simulation (a few thousand instructions per second), but simulates the actual hardware. Cycle-accurate simulators fall somewhere in between, providing intermediate simulation speed and an accurate cycle count to estimate performance.

Cycle-accurate simulators are traditionally hand-coded for a particular microarchitecture. Many exist, but are usually proprietary. Some widely used, freely available cycle-accurate simulators include Cycle-Level SPIM, based on the the SPIM instruction set simulator [7], which simulates a MIPS R2000, and the SimpleScalar toolkit [2], which simulates a generic, fictitious superscalar processor. Minor microarchitectural variations are often simulated by modifying a simulator like SimpleScalar, because the effort of writing a new simulator is enormous. Such an approach, however, applies only to microarchitectures very similar to the existing simulator. In contrast, we demonstrate an easy way to create cycle-accurate models from scratch, allowing greater flexibility and simpler descriptions. Our higher-level descriptions, however, do sacrifice some performance versus a highly-tuned hand-crafted simulator.

Recently, several research groups have proposed tools for generating processor simulators from higher-level descriptions, e.g., UPFAST [10], LISA [11], and Sleipnir [5]. All of these systems propose domain-specific description languages tailored for processors, and then generate simulators automatically from the descriptions. The domain-specificity allows the descriptions to be much shorter (a few thousand lines for a cycle-accurate model of a small processor) than a simulator written in a general-purpose language. The automatically generated simulators achieve performance approaching that of the highest performance, hand-coded simulators. Our work can be viewed as taking this research direction one step higher: we free the user from having to describe stall and bypass conditions, creating models an order of magnitude smaller, but our generated simulators run roughly an order of magnitude slower. We have invested little research on our simulation performance, concentrating instead on the specification style. We are hopeful that techniques others have developed for high-performance simulation (such as the compiled simulation

*This work was supported in part by the National Science and Engineering Research Council of Canada.

used by LISA) can be applied to our very high-level specifications. We won't completely close the performance gap, however, because our simulators need to do more work dynamically to make up for less design information specified in the description, so we envision our approach being used for early-stage design exploration, with the additional coding effort to use the lower-level simulator generators expended only on the most promising designs.

Philosophically, our approach shares some commonality with research aimed at developing more powerful, higher-level hardware description languages. For example, the Hawk project builds a hardware description language out of the lazy functional language Haskell and has developed several microarchitecture-level processor models [9]. Such an approach offers the potential advantage of making the full abstractive and expressive power of functional programming available to the hardware designer. Our approach is much less ambitious, being restricted to modeling processors rather than general hardware. As a result of this domain-specificity, however, our descriptions are much higher-level, because our tool understands program order, forwarding, and stalling. Furthermore, our generated simulators run orders of magnitude faster. Similarly, Hoe and Arvind propose a hardware description style based on guarded commands and synthesize a subset of a MIPS R2000-like processor (ISA- but not cycle-compatible) [4]. Their approach is also a general-purpose hardware description language, so by comparison, our approach sacrifices generality in exchange for easier-to-write, higher-level, shorter descriptions.

## 2. Theory

Our entire approach hinges on two assertions:

**Assertion 1 (Sequential Execution Semantics)** *Processors are designed to provide the same programmer-visible behavior as if individual instructions were executed completely, one-at-a-time, one-after-another.*

**Assertion 2 (Producer Knows Best)** *The functional unit that produces (or fails to produce) a result is the clearest place in a description to control access to that result.*

The Sequential Execution Semantics assertion means that a description style, by default, should enforce the normal execution semantics that software programmers expect and that processors typically provide. For example, if an instruction depends on the result of the preceding instruction, a simulator generator should determine automatically that a stall or bypass is necessary. In contrast, typical processor descriptions languages require the model to describe explicitly all stall and bypass conditions needed to enforce sequential execution semantics. Describing these conditions is error-prone and time-consuming, and needlessly complicates the description. Note that our specification style can handle small violations of this assertion, — for example, our MIPS R2000 model in Section 4 accurately models delayed branches — but if a processor completely disregards sequential execution semantics, e.g., a completely non-interlocked VLIW processor, our specification style provides no advantage over existing methods.

The Producer Knows Best assertion means that the description code to determine which instructions have access to which results should be localized to the portion of the description that produces the result, rather than scattered throughout the description among all possible places a result might be needed. Other description styles typically require an explicit check of whether a result is available in any code that accesses that result. In our style, those checks are implicit and can be automatically inferred by a simulator generator. For example, consider the code sequence:

```
addi $t0, $t1, 1
add  $t0, $t0, $t2
sw   $t0, 0($t3)
```

(MIPS assembly convention has the destination register as the first operand, except for store instructions.) In our description style, the add-immediate and add instructions would simply announce whenever their results were ready, and the Sequential Execution Semantics assertion would ensure that the add and store-word instructions get the proper data values, via forwarding or stalling, without any explicit checks for data readiness in the description.

More concretely, our description style requires specifying the functional units of the processor, what each functional unit does, and how instructions flow from one functional unit to the next (or are retired). Instructions carry with them their *context*, i.e., the instruction's view of the values of all machine state. The use of contexts is similar to other formal models of pipelined execution (e.g., [1, 9]) and greatly simplifies the semantics of the description. Functional units will rely on the newest available value for operands, automatically forwarding or stalling as necessary, unless otherwise specified. Results are written into the instruction's context, where they are available to subsequent instructions automatically. Before an instruction retires, its results are committed to a global context, unless the instruction was killed due to exceptions, misprediction, etc.

To support the Sequential Execution Semantics, we introduce the concept of *containers* and *access operators*. Containers are like variables and correspond to all programmer-visible machine state, such as registers, the program counter, condition codes, etc., as well as any additional bookkeeping information needed to process the instruction, such as the instruction register, the opcode, etc. An instruction's context is simply an assignment of values to containers. In addition to their normal values, containers can be assigned the special values UNAVAILABLE, which indicates that the given value is not yet available, and TRANSPARENT, which indicates that the value of the container should be taken from the preceding context. Typically, an instruction's context will initialize with all containers being UNAVAILABLE, and then all containers that the instruction won't modify will be changed to TRANSPARENT. When the instruction's result is available for forwarding to subsequent instructions, the result container will be changed from UNAVAILABLE to the result value. When the instruc-

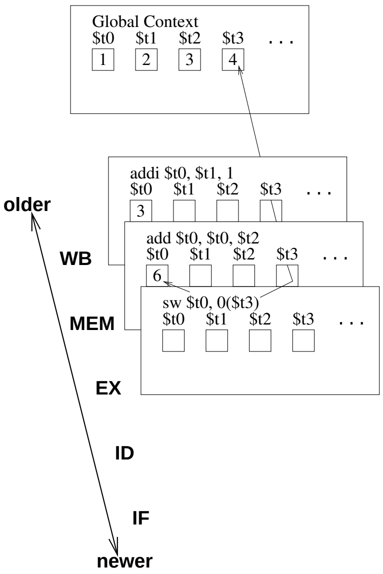


**Figure 1.** The store-word instruction wants to execute and needs to access the values of the registers. It gets the values from the most recent context with a non-TRANSPARENT value, so it gets $t0 from the preceding instruction, but $t3 from the global context, because the add instructions assigned TRANSPARENT to their $t3 containers (because they knew they didn't modify the value of $t3). IF, ID, EX, MEM, and WB refer to MIPS R2000 pipeline stages.

tion is ready to retire, the result value will be written to the global context. (See Figure 1.) We call assignments to the current context *announcements*; we call assignments to the global context *commitments*.

When a functional unit accesses the value of a container, the default semantics is to return the most recent non-TRANSPARENT value for that container. The value might be forwarded from a recent instruction or taken from the global context. If the most recent value is UNAVAILABLE, the functional unit will stall. This default behavior automatically enforces sequential execution semantics. Each instruction always gets the most current value for its operands. Stalling occurs automatically if a needed result has not yet been announced. Forwarding occurs automatically if a needed result has been announced, but not yet committed. Precise exceptions are easy to implement because uncommitted values disappear when instructions are killed/retired. No description effort is spent implementing any of these effects.

In unusual circumstances, we would like to bypass the sequential execution semantics, so we provide alternative access operators. To date, we have needed only four: Given a container $c$, the default access $c$ denotes the most recent non-TRANSPARENT value of the container, starting from the present context; $c'$ denotes the most recent non-TRANSPARENT value of the container, starting from the *previous* context; $c$\$ denotes the most recent non-TRANSPARENT value of the container from the previous context *after all functional units have completed the current clock cycle*; and $c$# denotes the value of the container in the global context. For example, the MIPS instruction set architecture specifies delayed branches — the instruction following a branch always executes, regardless of whether the branch is taken or not. Our specification style most naturally produces non-delayed branches in which the Fetch unit is stalled until the branch is resolved. To get delayed branches, we simply add a branch target container and a branch taken flag. The fetch unit fetches instructions normally, unless the branch taken flag signals a branch. The branch taken flag is accessed using the # access operator, so the fetch unit doesn't stall even though the branch decision isn't available yet. (The functional unit that resolves the branch commits the decision to the global context.)

Note that existing description styles make the delayed branch case easy to describe, as it is a natural artifact of the pipelining, but makes normal execution hard to describe. Our description style makes the common case (sequential execution semantics) simple, at the cost making the uncommon cases somewhat harder.

## 3. Implementation

To test our specification approach, we built a simple proof-of-concept language for specifying pipelined processors, along with an automatic tool to translate the specifications into cycle-accurate simulators. The language allows declaring containers and pipeline stages. The pipeline stages consist of guarded assignments, which can be announcements (denoted by `<-`) and commitments (denoted by `:=`). The right-hand-side of the assignments can be arithmetic expressions, using the four access operators defined above. All assignments whose guards evaluate to true are executed in parallel.

The translator is also quite straightforward. The generated simulator creates a global context and a context for each pipeline stage. Contexts are stored in a sparse, linked-list representation, since most containers will be transparent in most contexts. For each clock cycle, the simulator evaluates the access operators by searching backward through the contexts, evaluates the guards, and executes the assignments. We have not used any sophisticated techniques for accelerating processor simulation, such as predecoding instructions, cross compilation, or strength reduction. Given that our current simulators are traversing linked-lists for every container access, we believe substantial speed improvements are possible with a more efficient implementation. Given our higher-level semantics, however, it is not clear if we can match the performance of simulators generated from closer-to-the-machine specifications.

The most challenging aspect of the implementation turned out to be the system interface. Our language was designed to specify processor cores, but testing the resulting simulators requires loading object files, performing I/O, handling system calls, etc. We have implemented a minimal, ad hoc system call interface, similar to that provided by SPIM [7]. The processor description can invoke a machine-dependent `syscall` function to access functions from the host machine.

We do not currently model cache behavior, as our emphasis was on the processor core. It would not be difficult to add an external cache model to the simulator, that would stall accesses to a designated memory container based on cache effects.

## 4. Example: MIPS R2000

We have implemented the integer core of the MIPS R2000 [6], including all user-mode instructions except coprocessor support and non-aligned loads/stores. The model we present here does not implement exceptions, although they could be added easily.

The R2000 is a classic, 5-stage RISC pipelined processor. Extensive documentation is available, and the microarchitecture has been thoroughly studied, so it makes an excellent proof-of-concept example. Although such a design is no longer leading edge, it is similar to low-end application-specific cores designed and used today. We intend to extend our language to handle more modern microarchitectures (e.g., superscalarity) in the future.

### 4.1. Processor Specification

The most difficult and tedious part of the processor specification was the instruction decoder. Decoding instructions involves nothing more than extracting bit fields and pattern matching, but the decode unit encompasses over half of the entire description. Our research is not directed at instruction decoding, so this part of the model is simply a necessary evil. Fortunately, other researchers have developed high-level techniques for specifying instruction decoders [12], so a real, high-level processor specification language should combine that work with ours.

Our entire processor specification is 290 lines long, of which 150 lines are devoted to the instruction decoder. The description starts by declaring containers for memory, the 32 general-purpose registers, branch target and branch taken flag, program counter, instruction register, and a handful of instruction fields and temporary results. The R2000 has five pipeline stages, so the description also declares five pipeline stages.

The first pipeline stage is the instruction fetch unit:

```
constructor FetchStage
 {
  PF# == 0:{INS <- Mem#[((PC') >>L 2) + 1];
               PC <- PC'+4;}
  PF# == 1:{INS <- Mem#[((PN#) >>L 2)];
               PC<-PN#; PF:=0;}

  true: { Reg[0]<-tr; goto DecodeStage; }
 }
```

The container PF is a flag indicating a taken branch. We use the PF# access operator in order to get delayed branches as specified in the MIPS instruction set architecture. The two cases fetch either the next sequential instruction, or branch to the branch target in PN. The last assignment is always executed. It sets register 0 to be transparent (because register 0 always has the value 0) and directs the instruction to the decode unit.

As mentioned already, the decode unit is the messiest part of the description. The decode unit description consists of numerous cases such as this one:

```
// ADD[U],SUB[U],AND,OR,XOR,NOR,SLT[U]
  (INS & 0xFC000030) == 0b100000:
   {
    Mem[..] <- Reg[..INS[15..11]-1]
            <- Reg[INS[15..11]+1..]
            <- Hi <- Lo<- tr;
    T2 <- 0;
    V1 <- Reg$[INS[25..21]]; Op <- INS[ 5.. 0];
    V2 <- Reg$[INS[20..16]]; T  <- INS[15..11];
   }
```

The guard checks the opcode of the instruction register INS to see if it is one of the specified instructions. The first assignment sets all of memory to be transparent (the reserved word tr), as well as all registers except for the destination register (specified by bits 15 to 11 in the instruction register) and the temporary containers Hi and Lo. These containers are all transparent because we know at this point that the instruction will not modify any of them. The remaining assignments set up the opcode, source registers, and destination. The T2 container is used for instructions like MULT and DIV that have two destination registers. Branches are also resolved in the decode unit, committing the branch decision to the PF container. The decode unit directs instructions to proceed to the ALU unit.

The ALU unit is the second longest unit, simply because there are many possible ALU operations. The ALU unit description consists of numerous cases such as this one:

```
Op=="ALU-SLTU": Reg[T ? T : 32]<-(V1 <U V2);
```

(Reg[32] denotes a don't-care register.) The ALU stage computes the desired operation and directs instructions to the memory unit next. Note that all stalling and forwarding is implicit in the language and need not be spelled out.

The memory unit accesses memory if necessary, and consists of various cases such as this one:

```
Op=="MEM-LW": Reg[T?T:32] <- Mem[Offset >>L 2];
```

We do not model cache effects and non-aligned accesses.

The last stage is the writeback stage.

```
stage WriteBackStage
 {
  true: { retire; }

  T: { Reg[T] := Reg[T]; }

  T2: { Hi:=HiTemp; Hi<-HiTemp;
        Lo:=LoTemp; Lo<-LoTemp; }

  Op=="SYSCALL": { Reg[2] := syscall(V1,V2,V3); }

  (Op & 520)==520:
```

```
    { Mem[Offset >>L 2] := Mem[Offset >>L 2]; }
}
```

Note the use of commitment using the `:=` operator so that committed values affect future instructions, even after the current instruction is retired.

Except for the decode stage, we can see how our description style is vastly simpler than previous processor descriptions. The remaining question is whether we can generate efficient simulators from these descriptions.

### 4.2. Simulation Results

We selected the Dhrystone synthetic benchmark, Version 2.1 [13], to provide a non-trivial, integer workload to test the simulator. Dhrystone is too small for general-purpose processors, but is still widely used to benchmark embedded processors. Compiler limitations generating R2000 code precluded using a larger benchmark. Because of our limited system call interface, we omitted the initial log-file access of the Dhrystone benchmark, but included the rest of the benchmark. We used the `gcc` C compiler version 2.95.2 and had to substitute a few R2000 instructions for more modern instructions in the output code.

Automatic generation of the simulator took less than a second. The resulting simulator was compiled using `gcc` version egcs-2.91.66 with the `-O3` optimization option. We ran our experiments on a 733Mhz Pentium III with 256KB cache.

We consistently obtained over 240,000 simulated cycles/second. Direct comparison with other published results is difficult because of differences in modeling and host machine speed. As a very rough comparison, Önder and Gupta [10] reported 200K cycles/second for their UPFAST system simulating a cycle-accurate, 5-stage, complete MIPS on a 200Mhz Pentium Pro. The description was 5782 lines long. Jeremiassen [5] reported 5.1M instructions/second for his Sleipnir system, simulating an *instruction-level*, integer-only MIPS core on a 250Mhz MIPS R10000. The description was 700 lines long. Apparently, our performance is within an order of magnitude of more carefully tuned simulators, and our descriptions are much smaller and simpler.

## 5. Conclusion and Future Work

We have introduced a novel approach for specifying processor microarchitectures, and demonstrated the potential of our approach by implementing a simple description language with an automatic simulator generator, and then creating a MIPS R2000 model in our language. Our description is simpler and shorter than competing methods, and our simulation performance is within an order of magnitude of high-performance, purpose-built simulators. We believe our approach will be very useful for early-stage microarchitecture design exploration.

The most obvious future work is to improve the performance of our generated simulators. There is considerable literature on techniques to improve simulation performance, and we should be able to harness some of it.

More generally, we believe our ideas extend easily to superscalar processors and are working on formalizing the concepts and implementing a tool chain. The main difference is more dynamic constructs for directing instructions to functional units.

It may eventually be possible to synthesize efficient implementations from our descriptions, given promising work on automatic pipelining [3, 8]. Because the logic required to enforce sequential execution semantics is currently extremely challenging to design correctly, being able to synthesize from a small, obviously correct description is a very attractive long-term target.

## References


[1] M. Aagaard and M. Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design*, pages 13–32. Springer, 1994. Published in LNCS Vol. 901 in 1995.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 97-1342, University of Wisconsin, Department of Computer Science, 1997. The current version of SimpleScalar is available at `http://www.cs.wisc.edu/~mscalar/simplescalar.html`.

[3] S. Hassoun and C. Ebeling. Architectural retiming: Pipelining latency-constrained circuits. In *33rd Design Automation Conference*, pages 708–713. ACM/IEEE, 1996.

[4] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *International Conference on Computer-Aided Design*, pages 511–518. IEEE/ACM, 2000.

[5] T. E. Jeremiassen. Sleipnir — an instruction-level simulator generator. In *International Conference on Computer Design*, pages 23–31. IEEE, 2000.

[6] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[7] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report 90-966, University of Wisconsin, Department of Computer Science, 1990. The current version of SPIM is available at `http://www.cs.wisc.edu/~larus/spim.html`.

[8] M.-C. Marinescu and M. Rinard. High-level specification and efficient implementation of pipelined circuits. In *Asia and South Pacific Design Automation Conference*, 2001.

[9] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101. IEEE, May 1998.

[10] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *International Conference on Computer Languages*, pages 80–89. IEEE, May 1998.

[11] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In *36th Design Automation Conference*, pages 933–938. ACM/IEEE, 1999.

[12] N. Ramsey and M. Fernández. The new jersey machine-code toolkit. In *USENIX Technical Conference*, pages 289–302, January 1995. The current version is available at `http://www.eecs.harvard.edu/~nr/toolkit/`.

[13] R. Weicker. Dhrystone 2.1. *SIGPLAN Notices*, 23(8), August 1988. We downloaded from `ftp://ftp.nosc.mil/pub/aburto`.