Automatic Generation of Microarchitecture Simulators*

Soner Önder   Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh PA 15260
{soner,gupta}@cs.pitt.edu

Abstract

In this paper we describe the UPFAST system that automatically generates a cycle level simulator, an assembler and a disassembler from a microarchitecture specification written in a domain specific language called the Architecture Description Language (ADL). Using the UPFAST system it is easy to retarget a simulator for an existing architecture to a modified architecture since one has to simply modify the input specification and the new simulator is generated automatically. UPFAST also allows porting of simulators to different platforms with minimal effort. We have been able to develop three simulators ranging from simple pipelined processors to complicated out-of-order issue processors over a short period of three months. While the specifications of the architectures varied from 5000 to 6000 lines of ADL code, the sizes of automatically generated software varied from 20,000 to 30,000 lines of C++ code. The automatically generated simulators are less than 2 times slower than hand coded simulators for similar architectures.

1 Introduction

The realization of processor architectures in silicon is an expensive endeavor. Thus before a new processor is actually built, extensive simulation studies are carried out to estimate the expected performance of the microarchitecture on a variety of benchmark programs. The evaluation process requires collection of cycle level statistics. Typically the simulations must be carried out for millions of machine cycles so that general conclusions can be drawn regarding the performance of the microarchitecture with confidence. Thus to support the development of new processors, tools are required to enable rapid development of cycle level simulators that are fast enough to carry out extensive simulation studies.

A commonly used approach for developing simulators is their hand coding in a general purpose language such as C. Examples of some popular simulators which were developed using this approach include the SPIM simulator for the MIPS architecture [6], the SimpleScalar simulator [3], and the SuperDLX simulator [8]. The hand coding of simulators is a substantial task which typically takes between 12 to 24 man months. Once developed, such simulators are difficult to *retarget* to a modified microarchitecture or an instruction set architecture without a significant amount of effort. Another problem that one encounters is the difficulty in *porting* these simulators to different platforms. The portability issue arises due to the need for handling of external system calls that are made by the benchmarks being run. Solutions that either disallow such calls or allow external calls but sacrifice portability by allowing the simulator to run only on a specific platform (e.g., SPIM) are undesirable.

An alternative to hand coding a simulator is to generate it automatically from a machine specification written in a domain specific language. Automatic generation not only significantly shortens the development cycle, it also allows retargeting since modifications in the architecture can be made at the specification level and the new simulator can then be automatically generated. Although a number of hardware description languages [1, 7, 9, 13] are available, these languages are not suitable for developing cycle level simulators. These languages are capable of defining the hardware to the smallest detail and result in simulators that are orders of magnitude slower than cycle level simulators. The retargeting of simulators requires significant effort and no solution to the portability problem is offered by these languages.

In order to allow rapid prototyping of simulators we have designed a domain specific language for specifying processor microarchitectures called the *Architecture Description Language* (ADL) and implemented this language in the *University of Pittsburgh Flexible*

*This work is supported by NSF PYI Award CCR-9157371, NSF grant CCR-9402226, and grants from Intel Corporation and HP Labs to the University of Pittsburgh.

Architecture Simulation Tool (UPFAST). We support both *retargeting* and *portability* of simulators. The key contributions of our work are:

1. The ADL language has been designed to support an execution model that is suitable for expressing a broad class of processor architectures. It provides constructs for specifying the following:

(a) the microarchitecture including pipelines, control, and the memory hierarchy including instruction and data caches;

(b) the instruction set architecture (ISA), the assembly language syntax and the binary representation;

(c) a mapping between the calling convention of the simulated architecture and the machine that hosts the simulator. This specification enables the simulator to make external calls and achieve portability;

(d) commands for the collection of statistics that may be desired by the user to understand the performance of the architecture; and

(e) invocations of the debugger when error conditions are encountered and monitoring commands to identify information for display during debugging.

2. The UPFAST system has been developed to allow automatic generation of the cycle level simulators and other support tools. The system provides:

(a) an implementation of a compiler for ADL through which cycle level simulators can be generated automatically from an ADL processor description;

(b) automatic generation of support software tools including the assembler, disassembler and the loader/linker for the architecture;

(c) a cycle level assembly language debugger that assists in tracing of program behavior; and

(d) support for displaying statistics and monitored information.

3. We have obtained some experience with the system by developing three different simulators based upon the MIPS ISA [10, 5]. These simulators range from a simple pipelined design to a complicated superscalar architecture design. The three simulators were developed in a short time period of three months demonstrating the ability for rapid prototyping. The specifications of the architectures varied from 5000 to 6000 lines of ADL code while the sizes of automatically generated software varied from 20,000 to 30,000 lines of C++ code. Our automatically generated simulators are less than 2 times slower than hand coded ones.

In section 2, we present the ADL language. In section 3, we describe an implementation of the language, the UPFAST system. We conclude by reporting our experience with UPFAST in section 4.

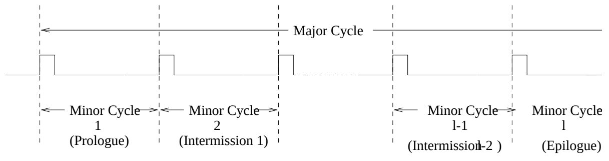

Major Cycle
Minor Cycle 1 (Prologue)
Minor Cycle 2 (Intermission 1)
Minor Cycle l-1 (Intermission l-2)
Minor Cycle l (Epilogue)

Figure 1: ADL Clock Labeling

2 Architecture Description Language

An ADL program primarily consists of the description of a processor architecture which includes the specification of the instruction set architecture as well as the organization of the components of the microarchitecture. Before we describe ADL in detail, let us first consider the model of execution used by ADL to express the operation of an architecture and highlight some of its design characteristics:

Explicit Instruction Flow and Instruction Context: In ADL the flow of instructions through the architectural components is explicit. The data associated with an instruction under execution is called the *instruction context*. The context is passed from one component to the next and is operated upon by the components till the execution of the instruction is complete. The context is allocated when the instruction enters the pipeline and is deallocated when the instruction retires.

The Machine Clock: The notion of machine clock is built into the language and the operation of the architectural components is described with respect to this clock. The machine clock is viewed as a series of *pulses*. Each discrete pulse is called a *minor cycle*, and a number of minor cycles are grouped together to form a *machine cycle*. The minor cycles in ADL are represented by a series of labels. The first and the last minor cycles of a machine cycle are labeled as the *prologue* and the *epilogue* and those in between are labeled as *intermissions*. The actions of each component in the system during a machine cycle are divided into the operations that it performs in each of the minor cycles. During the prologue a component receives an instruction context from another component for processing, during the intermissions it operates upon the instruction context, and during the epilogue it sends the modified context to another component. Fig. 1 shows the clock of a machine in which the major cycle is composed of λ minor cycles.

Artifacts and Processing Stages: The architectural components are divided into two categories: *artifacts* are components with standard well known semantics that are directly supported by the language and *stages* are components whose semantics must be explicitly specified as part of the ADL program.

Examples of artifacts include caches, memory units, and register files. Since they are directly supported by ADL as built-in *types*, the programmer can use them by simply declaring objects of these types in an ADL program. Access to artifacts takes the form of assignments to and from the artifact variables. Different implementations of these components can be used by specifying different attribute values for the artifacts. The interaction of an artifact with the machine clock is also specified as a list of attributes.

Processing stages are architectural components that exhibit a significant functional variety. Their operation is dependent on the microarchitecture as well as the current instruction being processed. Furthermore, the function such an element performs is tightly coupled with the system clock and the status of other components in the system. Thus, it is not feasible to follow a declarative approach for stages but instead the user must explicitly specify their semantics using Register Transfer Level (RTL) statements.

Separation of Instruction Set Architecture and Microarchitecture Specification: The ISA specification is separated from the microarchitecture specification to facilitate the development of different microarchitecture implementations for the same ISA or extend an ISA by adding new instructions without altering the microarchitecture. The above separation has the following consequence on the specification of stage semantics. The RTL statements describing the semantics of stages are divided into two components: the *general component* that is common to all instructions and the *ISA-component* which depends upon the specific instruction being processed. The former is specified in the microarchitecture description while the latter is included as part of the ISA specification.

Time Annotated Actions and Parallelism in the Microarchitecture: The specification of the actions associated with the execution of specific instructions as well as the actions associated with various architectural components are annotated with timing information so that it can be determined when they are to be performed.

The procedures that implement the *general component* of actions associated with a processing stage carry the name of the stage and the label of the minor clock cycle during which they are to be executed. Such procedures are referred to as *time annotated procedures* (TAPs). Since there are λ minor cycles, there may be up to λ TAPs for a given stage. The *ISA-component* associated with an instruction is labeled with the name of the processing stage and optionally with the label of the minor cycle during which it must be executed. These statements are referred to as *labeled register transfer level* (LRTL) segments.

Parallelism at the architecture level is achieved by executing in each machine cycle the actions associated with each component during that cycle as well as actions associated with an instruction that are annotated with the current cycle. The machine execution is realized by invoking each TAP corresponding to a minor cycle as the clock generates the corresponding label and the parallel operation of individual components is modeled by concurrently executing all TAPs which have the same annotation. During this process, LRTL segments corresponding to the currently processed instruction are *fused* together with the corresponding TAP. The operation of a machine can be described as follows:

```
do forever
  for clock.label := $prologue$, $intermission\ 1$, ...
          ... $intermission\ (\lambda - 2)$, $epilogue$ do
    $\forall$ TAP, TAP.annotation = clock.label do
          { process {TAP; TAP.instruction.LRTL } }
end
```

2.1 Microarchitecture Specification

The specification of the microarchitecture consists of describing the artifacts of the architecture, declaring pipelines involved and their stages, specifying instruction contexts, and finally defining TAPs for each of the stages. In the following sections, we will use a simple pipelined architecture shown in Fig. 2 to discuss each of these steps. In this architecture, the instruction fetch stage (IF) fetches instructions from the instruction cache and ships them to the instruction decode (ID) stage. ID stage decodes the instructions it receives, fetches their operands from the register file, and sends them to the execution unit (EX). The memory access (MEM) stage performs a data memory access for the load and the store instructions, but other instructions pass through this stage unchanged. Finally, the write back (WB) stage writes the results back to the register file. In order to eliminate pipeline stalls that would otherwise result, data values are forwarded through forwarding paths to the earlier stages.

Artifacts: Artifacts are hardware objects with well-established operational semantics and they are supported as built-in *types* by the language. A declaration of an artifact supplies the values of the *attributes* of the artifact to derive a specific implementation of the artifact. For an artifact, we also specify how long does it take to process a single request in terms of clock cycles (i.e., the *latency*), the rate at which new requests can be issued to the artifact (i.e., the *repeat rate*), and the maximum number of requests that can be outstanding in a clock cycle (i.e., the number of

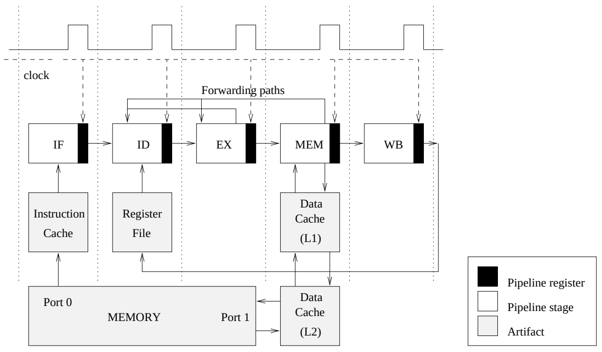


Figure 2: A Simple Pipelined Processor

ports). The list of the different types of artifacts supported by the language is given below.

artifact-declaration ⇒ *register-declaration*
| *register-file-declaration*
| *memory-port-declaration*
| *cache-declaration*
| *buffer-declaration*
| *token-declaration*

A *register* declaration declares an artifact of type simple register while a *register file* declaration declares an array of registers. Registers and register files may be given the attribute *shadow* which makes them invisible to the instruction set. ADL allows definition of one or more *aliases* for the individual register file entries. A *memory* declaration defines a memory port with a given access latency in units of machine cycles and a data path width in units of bits. For the *cache* artifact, attribute values include degree of set associativity, the kind of replacement strategy, and whether it is a write-back or write-through cache. Memories, caches and buffers have an important property of being *stackable*. This property is required for building memory hierarchies. When an artifact is declared, the name of the artifact immediately lower in the hierarchy is mentioned using the *of clause*, effectively placing the new artifact higher in the hierarchy.

```
shadow register temp 16;  # A 16 bit temporary register.
register file gpr [32,32] # 32 registers,32 bits each.
   $zero    0,            # $zero is another alias for gpr[0]
   $at      1,            # $at is another alias for gpr[1]
   $v0      2,
   .....
   $sp      29,
   $fp      30,
   $ra      31;
memory mport0 latency 12 width 64, # 64 bit path to memory.
       mport1 latency 12 width 64; # 64 bit path to memory.
instruction cache icache of mport0 directmapped 64 kb 4 wpl;
data cache  l2 of mport1 directmapped 64 kb 4 wpl,
            l1 of l2     4 way 8 kb 4 wpl;
```

Figure 3: Example artifact declarations

A sequence of artifact declarations for the example pipelined architecture of Fig. 2 is shown in Fig. 3. The first declaration declares a temporary register invisible from the instruction set. Next a register file `gpr` is declared and individual registers in the file are assigned aliases. The names `$0`, `$sp`, etc, are ISA visible since the register file itself is ISA visible. RTL statements may use either form of access (i.e., `gpr[31]` or `$ra`). The declaration specifies two memory ports with 12 cycles of access latency and 64 bit data paths. The memory port `mport0` hosts a direct mapped instruction cache of 64 kilobytes with 4 words per cache line. Memory port `mport1` hosts a direct mapped cache of similar attributes and this direct mapped cache in turn services a four way set associative cache of size 8 kilobytes. Thus, the cache L1 is at the highest level in the hierarchy and the memory ports are at the lowest level.

Once declared, artifacts are accessed just like variables by the RTL statements in the specification. For complicated structures, such as data caches, passing of additional parameters may be required. For example, in order to store a single byte to the L1 cache, and retrieve a halfword, the following sequence of RTL statements could be used:

```
l1.(_BYTE) [addr] = data_value;
data_value = l1.(_HALFWORD) [addr];
```

When an artifact is accessed, the status of the result is queried using the `access-complete` statement. This statement returns a true value if the operation has been completed successfully, and a false value otherwise. A false value may be returned because the artifact is slow, such as in the case of memory-ports, or because there is a structural hazard. In these cases the request must be repeated. Further details of why the operation was not successful may be queried using additional statements.

Processing Stages and Instruction Context Declarations: The primary means of declaring stages of the microarchitecture is the *pipeline declaration*. A pipeline declaration specifies an ordering among pipeline stages such that each stage receives an instruction context from the preceding stage and sends the processed context to a later stage. There may be more than one pipeline declaration in an ADL program but the stage names must be unique. Once a stage is declared using a pipeline construct, TAPs may be specified for each of the stages and semantic sections of instruction declarations may utilize the stage names as LRTL labels. The following declaration defines the pipeline for the example architecture:

```
pipeline ipipe (IF, ID, EX, MEM, WB);
```

In ADL, the set of data values carried along with pipeline stages are grouped together in a structure called `controldata`. There is only one such declaration, which means all stages have the same type of context, and the *instruction context* is the *union* of the data required by all the pipeline stages in the system. While in a hardware implementation pipeline stages may carry different types of contexts, definition of instruction context in this way simplifies the transfer and handling of instruction contexts in the simulator. Since there is a uniform single instruction context for all pipeline stages, each pipeline stage name is also an object of type controldata. The following is a simple controldata declaration for a pipelined machine:

```
controldata register
  my_pc 32, # Instruction pointer for the instruction.
  simm  32, # Sign extended immediate.
  ....
  dest  32, # dest holds the value to be written.
  lop   32, # lop holds the left operand value.
  rop   32; # rop holds the right operand value.
```

Elements of the controldata structure may be accessed from TAPs and by the semantic parts of instruction declarations (i.e., LRTLs). Access to the elements of the structure may be qualified or unqualified. When they are not qualified, the pipeline stage is the stage of the TAP that performs the reference or the label associated with the LRTL segment that performs the reference. In its qualified form, the syntax controldata-element[stage-name] is used to access the instruction context of another stage. This form is primarily used to implement internal data forwarding by either the source stage writing into the context of the sink stage or the sink stage reading the data from the context of the source stage.

Specifying Control and TAPs: The machine control is responsible for checking the conditions for moving the pipeline forward, forwarding the instruction context from one stage to the next, controlling the flow of data to and from the artifacts, and introducing stalls for resolving data, control, and structural hazards. In ADL, the semantics of the control part of the architecture is specified in a distributed fashion as parts of TAPs by indicating how and when instruction contexts are transferred from one stage to another.

The movement of an instruction context through the pipeline, from one stage to the next, is accomplished through the send statement. The send is successful if the destination stage is in the idle state or it is also executing a send statement in the same cycle. All pipeline stages execute the send statement during the epilogue minor cycle. In the normal pipeline operation, an instruction context is allocated by the first pipeline stage using the ADL statement new-context. This context is then filled in with an instruction loaded to the instruction register. When this stage finishes its processing, it executes the send statement to send the context to the downstream pipeline stage. When a context reaches the last pipeline stage it is deallocated using the ADL statement retire. If any of the pipeline stages does not execute a send, send operations of the preceding stages fail. In this case, they repeat their send operations at the end of next cycle. For decoding the instructions, ADL provides a decode statement. The decode statement does not take any arguments and establishes a mapping from the current context to an instruction name. This mapping is fully computable from the binary section of instruction declarations. Once decoded, all the attributes of the instruction become read-only controldata variables and are accessed accordingly.

The conditions for internal data forwarding can be easily checked by the stage that needs the data. For example, the TAP for the ID stage in the example pipelined machine may check if any of the stages EX and MEM has computed a value that is needed by the current instruction by comparing their destination registers with the source registers of the instruction currently in the ID stage. If that is the case, the stage reads the data from the respective stages instead of the register file.

For the handling of artifact data-flow and the handling of various hazards, ADL provides the stall statement through which a stage may stall itself. The stall statement terminates the processing of the current TAP and the remaining TAPs that handle the rest of the machine cycle. The net effect of the stall statement is that no send statement is executed by the stage executing the stall in that machine cycle.

In addition to the stall statement, ADL also provides statements to reserve a stage, release a stage, and freeze/unfreeze the whole pipeline. When a stage is reserved, only the *instruction* that reserved it may perform a send operation to that stage, and only this instruction can release it regardless of where in the pipeline the instruction is at. When a stage executes a freeze, all stages except the stage that executed the freeze statement will stall and only the stage that executed the freeze statement may later execute an unfreeze statement.

Examples of hazard handling using these statements are shown in Fig. 4. Fig. 4(a) indicates the case where the result of a load instruction may be used immediately by the next instruction. Such data hazards cannot be overcome by forwarding alone and therefore require insertion of *pipeline bubbles*. The stage in this case checks for the condition by examining the context of the EX stage and its destination register and stalls appropriately. Because of the stall, the ID stage

```
instruction register ir;
stall category mem_ic,ld_d_dep,pool_full;

(a) procedure ID epilogue
    begin if i_type[EX]== load_type &
            (dest_r[EX]==lop_r | dest_r[EX]==rop_r) then
            stall ld_d_dep;
    end ID;
(b) procedure IF prologue
    begin ir=icache[pc];
          if access_complete then
             begin unfreeze; pc=pc+4 end
          else
             begin freeze; stall mem_icl end;
    end IF;
(c) pipeline RSPOOL(RSTA[64]);
    procedure ID epilogue
    begin reserve_unit RSTA my_pc;
          if ! access_complete then stall pool_full;
    end ID;
```

Figure 4: Handling of Hazards.

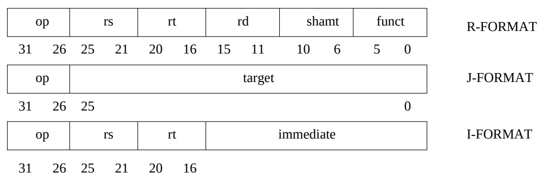
does not execute a `send` in this cycle. Since the send operations of following stages are not effected by the `stall` of prior stages, the EX stage enters the next cycle in an idle state which is equivalent to introducing a pipeline bubble. An instruction cache miss in a pipelined architecture is usually handled by freezing the machine state. In Fig. 4(b), the instruction fetch stage executes a `freeze` statement whenever there is a cache miss. A `stall` is also executed so that the epilogue will not attempt to execute the `send` statement. Note that an `unfreeze` is always executed whenever the cache access is successful. Executing an `unfreeze` on a pipeline which is not frozen is a null operation. In this way, the stage code does not have to be history sensitive. Finally in Fig. 4(c), a structural hazard and its handling is illustrated. The example shows one possible way to implement a unified pool of 64 reservation stations using an array of stages for the Tomasulo's algorithm [14]. The ID stage attempts to reserve a unit from the pool of reservation stations. If the `reserve` statement is unsuccessful, the stage executes the `stall` statement.

2.2 ISA Specification

The ISA is specified by means of instruction declarations which describe the syntax and semantics of both the machine instructions and the macro instructions using a uniform syntax given below:

instruction-declaration	⇒ *machine-instruction-declaration* | *macro-instruction-declaration*
machine-instruction-declaration	⇒ *syntax-part* `emit` *binary-part semantic-part*
macro-instruction-declaration	⇒ *syntax-part* `macro` *semantic-part*

There are three major components of the instruction specification. These are the *syntax-part*, the *binary-part* and the *semantic part*. The syntax part and the binary part together define how the assembler should parse instructions and generate the appropriate binary encoding of them. The binary part is also used to automatically generate the decoder for the implementation of the `decode` statement discussed earlier. The semantic part of a machine instruction description is a list of LRTL segments describing what each stage should compute when the instruction is processed by the stage, whereas the semantic part of a macro instruction description specifies how the assembler should generate machine instructions from the macro specification.



(a) MIPS formats

declare	op	constant	field	31	6,
	rs	register	field	25	5,
	rt	register	field	20	5,
	rd	register	field	15	5,
	shamt	integer	field	10	5,
	funct	integer	field	5	6,
	target	integer	field	25	26,
	immediate	integer	field	15	16,

(b) MIPS formats

Figure 5: Instruction format specification

Generation of a binary encoding of an assembly instruction involves three steps. These are the parsing of the assembly instruction, extracting the values of any instruction fields which are derived from the assembly instruction, and packing these values in an instruction format. The instruction format for an instruction is a sequence of fields making up the instruction word. Some of the instruction formats for the MIPS architecture are shown in Fig. 2.2.

ADL defines instruction fields by associating a *start bit* and *field width* pair with a name. The same pair may be defined multiple times using different names since the same pair may have a different purpose in a different instruction format. If a field has a constant value for all the instructions in the instruction set, it is declared to be a *constant* field. Otherwise, it is declared to be one of the ADL types *register*, *integer* or *signed integer*. Such fields are considered to be *variable* fields. Variable fields typically get their values from the assembly instruction when such an instruction is parsed by the assembler. We specify the instruction fields using the *declare* construct.

declare-construct	⇒ `declare` *declarations*
declarations	⇒ *field-declaration* | *variable-declaration* | *temporary-declaration*
field-declaration ⇒	*field-name* (`constant` | `integer` | `register` | `signed`) `field` *start-bit field-width*

Examples of field declarations for the MIPS instruction formats are given in Fig. 2.2. Field declarations alone are not sufficient to describe the binary encoding of an instruction. We also need to define which fields make up the instruction (i.e., the instruction format) as well as how their values are computed. Instead of defining separate instruction formats and then mapping instructions to these formats [4], ADL chooses to specify the instruction format as part of the instruction's binary part. The binary part of each instruction is represented as a sequence of *field expressions*. A field expression is the assignment of a value to a field of the instruction. The value assigned to a field may be a constant, a constant expression, or, it may reference a value to be derived from the assembly instruction by the syntax-part. The ADL syntax for the syntax-part and the binary-parts of an instruction declaration are given below:

```
syntax-part        ⇒ instruction-mnemonics argument-list
argument-list      ⇒ argument | argument argument-list
argument           ⇒ label-variable | field

binary-part        ⇒ field-expression
                   | field-expression binary-part
field-expression   ⇒ field | field = constant | field = <fog-list>
fog-list           ⇒ fog-predeclared | fog-list.pure-function
fog-predeclared    ⇒ label-variable.base
                   | label-variable.offset
                   | label-variable.absolute
                   | label-variable.delta
                   | label-variable.segoffset
```

The syntax part of an instruction declaration is a list of *arguments* defined to be either *label variables* or fields. A field in the argument list means that the assembler should expect to find an object of the corresponding type such as a register or an integer constant at the corresponding position of the assembly instruction. A label variable represents an address primary. Examples of address primaries include labels, base/offset pairs, and any constant arithmetic on labels. Field expressions given in the binary-part may query the values of the arguments of the instruction using pre-declared functions such as *base*, *offset*, *absolute*, or *delta*, or substitute them directly. These values may also be transformed by using *pure functions* which are functions which have a single parameter and return a single transformation of this parameter.

Let us now see how the assembler could parse an instruction using the specification shown in Fig. 6 and generate the appropriate binary. In the example, the argument part consists of a register field (`rt`), and a label variable (`address`). Therefore, the assembler expects to find a register name followed by a sequence of tokens which can be reduced to an address primary when a `lw` mnemonic is detected in the input stream. The field expressions in the binary part indicate that the opcode field must be set to the constant value of 35, `rs` field must be given the base register number representing the address, and the immediate portion must be given the offset representing the address. Since the `rt` field appears in the argument list, it gets a register number from the parsed instruction. ADL representation of binary encoding is a concise representation and is more natural than the SLED approach [11, 12] since there is no need for separate opcode tables and *constructors*.

Specifying Instruction Semantics: The semantic-part of an instruction specification serves two purposes. These are the specification of what each stage computes when such an instruction is received and instruction classification so that stages may apply operations specific to a class of instructions. For example, branch instructions may be handled by a specific stage which requires that the type of an instruction be known so that proper instruction steering can be performed.

The instruction specific operations of stages are specified using LRTL segments. A LRTL segment is a program segment that consists of register transfer level statements where each block of such statements are labeled using a stage name. The syntax of the LRTL segment is depicted below.

```
LRTL-segment       ⇒ begin labeled-RTL-list end
labeled-RTL-list   ⇒ labeled-RTL | ; labeled-RTL-list
labeled-RTL        ⇒ case stage-name RTL-statement-list end
```

The classification of instructions is achieved using an optional *instruction attributes* section where the attributes of the instruction are specified. These attributes can be queried by pipeline stages upon receiving the instruction. An instruction attribute is a member of the global enumeration defined by the *attribute declaration* given below:

```
attribute-declaration   ⇒ identifier : attribute-list
attribute-list          ⇒ name-list | integer
name-list               ⇒ identifier | identifier , name-list
```

Since an attribute of an instruction classifies an instruction, values of attributes must be specified for all the instructions. An example attribute declaration section that classifies instructions according to their operation types is shown below:

```
attributes
  i_type : alu_type, branch_type_0,
           branch_type_1,load_type,store_type;
end;
```

Let us examine the semantic part of the lw instruction declaration shown in Fig. 6. This instruction has the `i_type` attribute `load_type`, and LRTL segments `ID`, `EX`, and `MEM` define the operations each of the corresponding stages. The LRTL segment ID performs a sign extension using powerful ADL bit operations. The sign extension is achieved by repeating the bit 15

```
declare  rt         register  field
         rs         register  field
         immediate  signed    field
         address    label     variable
Instruction
    lw rt address
       emit opcode=35 rs=<address.base>  rt immediate=<address.offset>
       attributes (i_type: load_type, dest_r: rt, lop_r: rs)
       begin
           case ID   simm=immediate.[15:1] |<  16 || immediate;   end;
           case EX   lmar=lop + simm;   end;
           case MEM    dest=dcache[lmar];   end;
       end;
```

Figure 6: MIPS Load Word Instruction

of the immediate field (`|<` operator) for 16 bits and then concatenating (`||` operator) it with the field itself. The result is then stored into the variable `simm`. The LRTL segment EX performs an address computation by adding the contents of the variable *lop* with the sign extended value computed by the ID stage. Similarly, the LRTL segment MEM performs a data cache access using the value computed in the EX stage and stores the returned value into the variable *dest*. Since writing back the results of instructions into the register file is common for all instructions, this task is handled by TAPs.

The address space of a TAP consists of the global address space implemented by the artifacts and the local address space defined by the instruction being currently processed. In Fig. 6, the variables *simm, dest_r, lop* are part of the local address space or the *instruction context*. When the execution of a TAP is completed, the local address space is *transferred* to another TAP instead of being deallocated. Typically, the next TAP that executes in the same context is the TAP belonging to the same stage that has the next clock label. When the TAP that has the label epilogue is executed, the context is either transferred to the prologue TAP of the same stage or to the prologue TAP of another stage.

Macro Instructions: Most compilers available today (e.g., gcc) make use of macro instructions in code generation. The task of converting these instructions into actual machine instructions is left up to the assembler. ADL handles macro instructions in a manner similar to machine instructions. The syntax part of the instruction has the same syntax, but no field variables are allowed in the argument part. Therefore, all of the instruction arguments are variables. Since macro instructions themselves do not directly lead to a binary representation, there is no binary generation part. The macro specification can be visualized as a procedure where the procedure arguments correspond to the instruction arguments and the semantic part corresponds to the body of the procedure. The procedure defines what instruction(s) should be generated given a particular instance of arguments. Instructions to be generated are specified using an *instruction call* statement that generates a machine instruction by passing the values of the fields of the instruction as parameters. The syntax for the instruction call statement is shown below.

instruction-call $\Rightarrow$ *instruction-mnemonics* : *field-assignment-list*

An example macro declaration for the MIPS load immediate instruction is shown in Fig. 7. This macro generates either a single instruction (`ori`) or a pair of instructions (`lui, ori`) depending on the size of the immediate field.

2.3 Calling Convention Specification

The purpose of the calling convention specification is to enable the simulator to perform external system calls on behalf of the simulated program so that operating system services can be provided through the operating system of the host machine. For this purpose ADL provides a calling convention section where the calling convention of the simulated architecture and the prototypes of external references are specified. From this specification, we are able to automatically generate an engine that can execute an external procedure by passing the values of the parameters from the simulated architecture and returning the results back into the simulator. This approach allows the language user to specify external references of a program and treat them as if they are single instructions.

The calling convention specification is based on the formal model and specification language for procedure calling conventions by Bailey and Davidson [2]. Their language has been modified so that it fits the general structure of the ADL language. The specification provides a mapping to a register or a memory location, given an argument's position and type in the procedure call. Since an argument's value may not have been written to the memory cell or to the register file at the time of the call, we modified the mapping so that each register identifier that may be used to pass arguments to the callee and each stack alignment are

```
declare rdest register variable,
        src2  integer  variable,
        tx    integer  temporary,
        ty    integer  temporary;
instruction li rdest src2 macro
begin tx=src2.[31:16];
      ty=src2.[15:16];
      if (src2.[31:17] == 0x1ffff) | (src2.[31:17] == 0) then
        ori:rt=rdest rs=0 immediate=ty
      else
        begin lui:rt=rdest immediate=tx;
              ori:rt=rdest rs=rdest immediate=ty;
        end;
end;
```

Figure 7: Macro Instruction Example.

```
calling_convention begin
  argument $4:int_p1, $5:int_p2, $6:int_p3, $7:int_p3;
             $f12:flt_p1,$f13:flt_p2,$f14:flt_p3,$f15:flt_p4;
  unbounded stk4: stk_p4, stk8: stk_p8;
  set intregs($4,$5,$6,$7,stk4),
      intfpregs(<$4,$5>,<$6,$7>,<stk8,st4>),
      fpfpregs (<$f12,$f13>,<$f14,$f15>,<stk8,stk4>);
  equivalence ($4,$f12), ($5,$f12), ($6,$f14), ($7,$f14);
  typeset singleword(int, void *, ...), doubleword(double, ...);
  map argument.type begin
    singleword : intregs;
    doubleword : map argument[1].type begin
                     singleword: intfpregs;
                     doubleword: fpfpregs;
                 end map;
  end map;
  prototypes begin
      reference errno, sys_errlist ...
      double cosh(double); int printf(int,...);
  end;
end calling_convention;
procedure int_p1()
begin
    int_p1=gpr[4];
    access_complete=( has_context EX |
             has_context MEM | has_context WB)==0;
end int_p1;
```

Figure 8: MIPS Calling Convention Specification

associated with a *supplier procedure*. Supplier procedures are microarchitecture specific procedures that return the value of the argument at the time of the call. In a pipelined architecture, the supplier procedure may return the value from an artifact if there are no instructions in the pipeline that are computing the value, or the value may be returned from a stage if the value has been computed, but did not yet reach the write-back phase. If the value is available and is being returned, the procedure sets the built-in variable `access-complete` to true. In the case that more cycles are necessary before the value becomes available, the `access-complete` variable is set to false. An example calling convention specification for the MIPS architecture is given in Fig. 8.

The calling convention specification consists of two sections, namely a *data transfer* section which describes how arguments are allocated into the registers and the stack locations, and a *prototypes* section, where prototypes of external procedures and names of external data addresses are supplied. The data transfer section consists of *argument declarations*, *set declarations* and a *map declaration*. Argument declarations associate either a register name with a supplier procedure name, or a stack alignment name with a supplier procedure. For example, in Fig. 8, argument register `$4` is associated with the supplier procedure `int_p1`. Stack alignment names are declared using the `unbounded` keyword and correspond to an unlimited pool of argument values starting at a given alignment of the frame pointer for the architecture. Supplier procedures for stack alignment names do the required alignment first and return the first word at the indicated location. The register names and stack alignment names given as part of the argument declarations are called *argument locations*.

`Set` declarations create ordered pools of arguments based on types. In our example, the set `intregs` creates a pool of argument values which consists of four integer registers and an unbounded pool of stack locations. Thus, a call site that requires six integer arguments would find the values of its first four arguments in the registers `$4, $5, $6, $7`, and the remaining two on the stack. In some architectures, if one register is used, some other registers can no longer be used for the following arguments. For example, in the MIPS architecture, if the floating point register `$12` is allocated, integer registers `$4` and `$5$` cannot be used to pass the following integer arguments. The specification handles this problem by creating *equivalence* sets given by the *equivalence* declaration. Register pairs listed in an equivalence declaration are removed together from the respective sets when one of them is allocated.

`Typeset` declarations group variable types that map to the same sized objects. Once the sets and typesets are defined, a `map` declaration creates a mapping from typesets to sets. For each argument type, first the typesets are consulted to find the corresponding typeset. Next the typeset is supplied to the map construct to find the set from which the argument value(s) should be obtained. These sets are consumed one by one for each argument value that is needed. The `map` declaration in the example in Fig. 8 specifies that any arguments which have a type listed in the singleword typeset will consume the set `intregs` while those which are members of the doubleword set select the set based on the type of the first argument.

The prototypes section is an ADL extension to the calling convention specification which is necessary to call external procedures. This section consists of a list of external procedure prototypes and data reference names used by the benchmark programs. Both procedures and data references can be renamed to match the names of the architecture so that greater portability is achieved.

The calling convention specification when complied, provides an interface that returns a list of supplier procedures given a call site. This interface is used by the simulator to assemble the argument values, perform the external call on behalf of the simulated program and return the values.

```
statistics "Total number of branches %d:",branch_count,
           "Empty slots %d:",empty_slots;
procedure EX epilogue
begin if i_type == branch_type1 | i_type == branch_type0 then
        begin branch_count=branch_count+1;
              if op[ID] == 0 then
                 empty_slots=empty_slots+1;
        end;
        .......
end;
```

Figure 9: Language Support for Gathering Statistics

2.4 Statistics Collection and Debugging

ADL provides support for assisting the user in collection of statistics that may be required to evaluate the specified architecture. An *instruction category* declaration is supported using which the user can classify instructions into different categories. The counts for the number of retired instructions in each of these categories are provided to the user by the generated simulator. The stall statement may be followed by an optional *stall category name*. In this form, the stall is registered under the mentioned category for the current instruction and the stall statistics for each of the categories are reported to the user. This can be helpful in identifying performance bottlenecks.

More advanced customized statistic collection is also possible. The ADL programmer can insert statements into the ADL program to collect special purpose statistics. For this purpose, ADL provides a *statistics declaration* which accepts a register name and a format specifier string. At the end of execution, the value of the register is printed using the supplied format. The example in Fig. 9 shows how one could count the number of branch-delay slots which are not filled with useful instructions by the compiler. In this example, the TAP for the EX stage checks if the instruction in EX is a branch instruction and the instruction in the ID stage is a null operation which has an opcode field of zero.

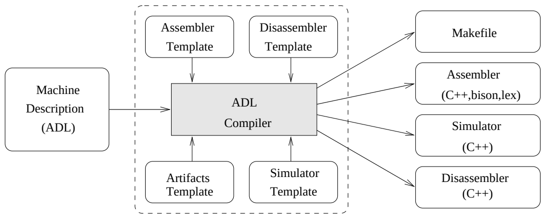
Interaction with the debugger can also be specified in an ADL program. The debugger can be entered through the ISA specification by using the ADL statement *pause*. In general, when an unexpected condition is detected, this statement may be used to enter the debugger. For example, a divide instruction may check for a zero operand and execute *pause* statement as part of an LRTL segment. The registers whose contents are desired by the user to be displayed when the debugger is entered can also be specified in the ADL program through the *monitor* declaration.

3 The UPFAST System

We have developed a compiler for the ADL language and additional support software, namely a linker/loader, C language libraries and a debugger in an integrated system called the *University of Pittsburgh Flexible Architecture Simulation Tool*, UPFAST. As shown in Fig. 3, the compiler reads in an architecture description in ADL and automatically generates a simulator, an assembler, and a disassembler from the given description. The generated simulator includes a built-in debugger which can be used to debug the architecture specification and monitor the simulated architecture. The resulting software is of good quality and can be used to compile and simulate large benchmark programs. For instance, we have extensively simulated SPEC95 integer and floating point benchmarks on a variety of architecture specifications.

The ADL Compiler

The ADL compiler uses prototype modules called *templates* to generate the desired software. A template is a prototype module of software that consists of only architecture independent components. For example, the assembler template contains a complete assembler with the exception of instruction set specific portions such as the mnemonics tables and specific rules to parse individual instructions and code that converts symbolic addresses to machine addresses. All these portions of an assembler are ISA specific and they are compiled in from the ADL program and filled in by the compiler. Similarly, artifacts have been implemented in another template file. For each instance of artifact declaration, the ADL compiler obtains the corresponding artifact declaration from the template file and generates the desired artifact implementation. The generation of the simulator system is accomplished by copying a template until a descriptor marker, indicating the position at which a component should be generated and placed, is encountered. The compiler generates a table, a procedure,



(a) Main Components

Software Component	Lines
ADL Compiler	15794
Artifacts	1141
Shared modules	499
Linker template	970
Disassembler template	531
Assembler template	3915
Simulator template	2677
Total	25527

(b) Number of lines of C++

Figure 10: The UPFAST System.

or a C++ class described by the marker. Once the required software element is generated, the scanning continues until another marker is found, or the end of file is reached.

The ADL compiler uses separate representations to describe the ISA and the microarchitecture. Imperative code such as TAPs, general procedures and LRTL segments are each represented by a separate syntax tree and these trees emanate from the internal representation of components of the architecture. In case of ISA, LRTL syntax trees emanate from instruction descriptions which represent the assembly syntax, binary representation, and macro implementations. In case of microarchitecture representation, syntax trees emanate from pipeline descriptions.

The Debugger

The debugger is entered through command line arguments or automatically upon detecting an error condition. Command line arguments may specify that the debugger must be entered after a specified number of cycles, or immediately. If a *deadlock* is suspected, that is, no instruction is retired for a large number of cycles, the simulator invokes the debugger. Upon an internal fault in the simulator the system's standard debugger along with the UPFAST debugger are fired. Finally the debugger may be invoked when the *pause* statement in an ADL program is encountered which is used when an unexpected condition occurs. The debugger when entered fires up two windows. The first window displays a disassembled memory image where the line number of the assembly language program, the memory location, binary encoding of the instruction, the machine instructions and the original assembly language program are shown in that order on each line. The second window displays the contents of registers specified in the *monitor* declarations and the contents of the pipeline stages of the machine architecture. In addition, the current number of machine cycles, the number of useful cycles and the number of stall cycles are also displayed.

Once in the debugger, the user can *single-step* the execution, continue the execution until a certain number of additional cycles are executed, or simply resume the execution. In case more powerful debugging is needed, the user may fire the regular system debugger, such as *gdb*, and perform further analysis. Since ADL compiler preserves ADL program names when generating the simulator, the user may inquire the values of variables using the ADL program names.

In some cases, problems surface after large numbers of simulation cycles although the exact cause of the problem may actually be hundreds of cycles prior

Component	PIPE	%	TOM	%	FWD	%
ISA spec	4549	78.6	4549	73.8	4549	76.6
Artifacts	210	3.6	230	3.7	230	3.9
μ-arch	554	9.6	890	14.4	673	11.3
Other	459	8.2	497	8.1	485	8.2
Total	5782		6166		5937	

(a) ADL lines of code

Component	PIPE	%	TOM	%	FWD	%
Assembler	6775	30.7	6775	21.9	6775	35.8
Disassm.	1508	6.8	1508	4.9	1508	8.0
Simulator	10942	49.6	19834	64.1	7803	41.2
Linker-etc	2838	12.9	2842	9.1	2842	15.0
Total	22063		30959		18928	

(b) Generated C++ lines of code

Figure 11: ADL programs and generated software

to the point it is detected. Solving these kinds of problems requires the knowledge of how a specific point in the program execution is reached. For example, a label may be the destination of a number of branch instructions and it is virtually impossible to know which path had been taken to arrive at this point. In order to address these problems, the debugger provides a unique *reverse execution mode*. In order to use this mode, the user specifies a range of cycles during which the simulator saves register and the pipeline contents. When the debugger is entered upon the occurrence of the problem, the program can be traced in reverse using the *backstep* command. In this mode, it is possible to backstep then forward step, within the window of saved cycles. This mode is slow and saves significant amounts of data. However, it has proven to be very valuable in our experience.

4 Experience with UPFAST

The UPFAST system has been implemented by one programmer over a period of 18 months using C++. Using the system, we have developed three simulators during a course of an additional three months. All the simulators we have developed are based on the MIPS ISA consisting of 84 machine instructions and 53 macro instructions. Simulators we have developed include a standard five stage pipelined MIPS architecture (PIPE), an implementation of the Tomasulo's algorithm applied to MIPS ISA (TOM), and finally a simulator for our ongoing research that investigates a novel microarchitecture called the *data forwarding architecture* (FWD).

For each of the architectures we defined, relative percentages and the sizes of various sections of ADL descriptions are illustrated in Fig. 4. One immediate observation is the larger share of the ISA specification. This is a direct result of the ADL approach to the problem. ADL approach is an instruction oriented approach and in this respect, a significant portion of the semantics of the machine execution is defined as part of the ISA specification. Another important point is the small size of the artifacts section. Although artifacts make up a significant portion of the actual hardware, they can be specified with ease by means of powerful ADL abstractions in a few hundred lines. Finally, while the sizes of the architecture specifications are around 6000 lines of ADL code, the sizes of the simulators vary from approximately 20,000 to 30,000 lines of C++ code. This clearly shows the merit of automatic generation.

In our experience, developing the ISA portion was relatively straightforward. Few software bugs have been traced to the ISA section. Most of these errors resulted either because of typing errors or ambiguity in the architecture manuals we used. Although ISA section is fairly large and the microarchitecture section is relatively small, the development times for the ISA component and the microarchitecture sections were roughly equal. This is expected as the microarchitecture section involves a high degree of parallel operation. These results demonstrate that the separation of ISA from the microarchitecture is a powerful approach since developing three fully functional simulators in three months would not have been possible without this separation.

The size of the ADL generated software for each of the architectures are given in Fig. 4. When we compare the size of the ADL generated software to comparable hand coded simulators, we observe surprising similarities. For example, our pipelined MIPS architecture implements essentially the same architecture as SPIM. The automatically generated PIPE simulator consisting of 22,063 lines compares quite well with the version of the SPIM software that we have that consists of 20,441 lines of C code. Comparison of MIPS-Tomasulo (an out-of-order architecture) implementation with SimpleScalar yields similar results. SimpleScalar package contains a total of 26,500 lines (excluding the library and the provided gcc compiler) and includes three simulators. Considering only the out-of-order simulator would correspond roughly to 25,000 lines, as these simulators are relatively small and share enormous amount of code. Thus, the automatically generated TOM simulator consisting of 30,959 lines compares well the above SimpleScalar simulator. Finally, the data-forwarding architecture has an intermediate complexity, for which there is no hand coded simulator that we can compare with.

Simulation speeds are very reasonable and compare well with hand coded simulators. The pipelined version executes at an average speed of 200,000 simulator cycles/second on a 200 MHZ Pentium Pro and the Tomasulo's algorithm executes at an average speed of 100,000 cycles/second. The Tomasulo's algorithm is comparable in complexity to the out-of-order SimpleScalar simulator [3] which reports a simulation speed of 150,000 cycles/second on a 200 MHZ Pentium Pro. Comparing these figures with the SimpleScalar numbers we find that ADL generated simulators are less than 2 times slower than the hand coded counterparts. On the other hand, given that the development time for SimpleScalar simulator was 18 man-months [3], it is obvious that the ADL approach is a cost-effective approach.

References

[1] J.R. Armstrong and F.G. Gray. *Structured Logic Design with VHDL.* New Jersey: Prentice Hall, 1993.

[2] M.W. Bailey and J.W. Davidson. A formal model and specification language for procedure calling conventions. In *22nd ACM Symp. on Principles of Programming Languages*, pages 298–310, 1995.

[3] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, V. 2.0. Technical Report 97-1342, Computer Sci. Dept., Univ. of Wisconsin Madison, 1997.

[4] T.A. Cook and E.A. Harcourt. A functional specification language for instruction set architectures. In *Proc. 1994 International Conference on Computer Languages*, pages 11–19, 1994.

[5] L. Huffman and D. Graves. *MIPSpro Assembly Language Programmers Manual.* Silicon Graphics Corp., Doc. 007-2418-002, 1996.

[6] J.R. Larus. SPIM S20: A MIPS R2000 Simulator. Technical Report 90-966, Computer Sci. Dept., Univ. of Wisconsin Madison, 1990.

[7] J.D. Morison and A.S. Clarke. *ELLA2000 A language for Electronic System Design.* McGraw-Hill, 1993.

[8] C. Moura. SuperDLX a generic superscalar simulator. Technical Report 64, School of Computer Science, McGill University, 1993.

[9] D.L. Perry. *VHDL.* McGraw-Hill, 1991.

[10] C. Price. *MIPS IV Instruction Set Revision 3.2.* MIPS Technologies Inc., September 1995.

[11] N. Ramsey and M.F. Fernandez. The new jersey machine-code toolkit. In *Proc. 1995 USENIX Technical Conference*, January 1995.

[12] N. Ramsey and M.F. Fernandez. Specifying representations of machine instructions. *ACM Trans. on Programming Lang. and Systems*, 1997.

[13] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[14] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.