A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools

by Chuck Siska
Rockwell Semiconductor Systems, Inc.
email: chuck.siska@rss.rockwell.com

Abstract

Many ISA-level machine description languages have been introduced to support the automated development and retargeting of digital signal processor (DSP) software development tools. These languages have yet to move below the ISA-level and adequately address DSP pipeline issues. ISA-level bit-accurate models may be reasonable for small micro-controllers, but are inadequate when applied to complex high-performance DSPs. We introduce a new machine description language, RADL, which supports the automated generation of DSP programming tools. From RADL, we can generate production-quality tools including cycle- and phase-accurate simulators. RADL has explicit support for pipeline modeling, including delay slots, interrupts, hardware loops, hazards, and multiple interacting pipelines in a natural and intuitive way. RADL can represent both SIMD and MIMD instruction styles. We have coupled our language to an in-house tool-chain generator which is used to create production assemblers, simulators and compilers.

1 Introduction

Time-to-market pressures in telecommunications and consumer electronics are in direct conflict with the increasing complexity of today's embedded hardware designs. These designs are increasingly turning to programmable DSP core processors employed in conjunction with custom circuitry. DSP core designs are themselves evolving. To meet these pressures it is necessary to be able to reuse application and hardware designs as well as to be able to generate programming tools for those hardware designs.

Early availability of assemblers, instruction-set simulators, and compilers are required to meet market pressures in developing new cost-effective applications. Manual development of such production-quality tools is simply too slow; hence, the need for retargetable programming tools. As pointed out by Fauth et al [1], such tools can even be used as aids to architecture exploration, if they can be generated fast enough. However, we do not wish to sacrifice tool quality. ISA-level retargeting leading to bit-accurate tools may be reasonable for micro-controllers, but is inadequate when applied to complex high-performance embedded DSP designs.

Making the retargeting of programming tools flexible and fast requires a machine description language which both can capture the ISA- and pipeline-level aspects of a design and can be easily understood and manipulated. Attempting to achieve these goals has led us both to build upon the work of others and to extend that work.

2 Previous work

We have designed a new machine description language because no existing system includes an ISA-level description with a simple to use yet detailed pipeline model suitable for embedded DSPs.

The nML processor description language [1-5] is the closest to our work. It includes the concepts of an operation hierarchy, separate specification of assembly language syntax, machine language encoding and simulator/compiler behavior. All these appear in RADL (Retargetable Architecture Description Language), albeit, in a somewhat different syntactic form. However, nML handles a pipeline only awkwardly [5]. It also includes an implicit program counter, which is inconvenient in some circumstances such as hardware or zero-overhead loops and interrupts, and the handling of delay slots is very limited. Resources (e.g., registers, etc.) are global, only, which complicates descriptions.

Fauth, Van Praet and Freericks [1], briefly describe an extension to nML to both define "transitory" registers and then to "synchronize" their behavior. Their transitory nature means that their values will become undefined after some finite delay. It is unclear how a collection of such synchronized registers are supplied with values "at the same time" without extreme care being given to the checking of "undefined" values. It is also unclear what happens when an operation attempts to store to a transitory register which has already received a value but is still awaiting the delay from a previously stored value to complete.

The LISA processor description language [6] was developed to support high performance simulators. LISA has a more detailed ability to describe certain types of pipelines with its Schedule construct. However, it is limited in this regard by its ASAP Gantt chart scheduling. Also, the pipeline stage names are defined implicitly throughout its description. It has no notion of a hierarchy of behavior through which description sharing can be achieved -- everything is at the instruction-level. LISA is limited in how it describes assembly language. As with nML, LISA also has an implicit instruction fetch mechanism and program counter, and resources are global, only. LISA doesn't support delay slots, zero-overhead loops, or interrupts.

The MIMOLA language [7, 8] typically uses a graph model of a restricted set of processors (although MIMOLA is not always linked to the graph model described by

Nowak [7]). The instruction set is implicitly defined in this model. MIMOLA appears too low-level for our purposes.

The Maril language [9] is specifically tailored to RISC processors and is designed to generate a compiler backend, including code instruction selection and scheduling and register allocation. It is limited to load-store architectures and, hence, cannot describe many typical DSP architectures.

CodeSyn [10] is a retargetable compiler backend system. It has no explicit machine description language but, rather, relies on describing the processor in the C programming language.

The RECORD processor description language [11] describes the behavior of machine instructions as register-memory transfer assignments and explicit no-operation statements. Its main objective is to support retargetable code generation. RECORD has no explicit notion of a processor pipeline, although pipeline conflicts can be represented. It also has no notion of assembly language.

The first element in the above list is the signal controlling whether the strategy is applicable. The second element, "ID:", indicates the stage involved in what we call a pipeline tactic. The third element, "stall(NOP)", indicates what to do. In the case of a "stall" tactic, the indicated stage and stages upstream are frozen in place -- they will repeat their stage behavior with the same inputs (and hence their instructions will not move forward). Also, the "NOP" argument to the stall tactic indicates that a NOP instruction will be inserted into the stage just after the ID stage. The rest of the stages (MEM and WB) will continue to flow smoothly. Other RADL pipeline stage tactics include, for example, "kill" to replace an instruction without stalling upstream stages.

If more than one strategy is applicable at a particular machine cycle, then the strategy with the higher priority is chosen. Priority is determined by the order of strategies in the pipeline description. For the simple DLX, we only need enter the one "load_raw" strategy, above. The default strategy, which is chosen if no other strategy is applicable, simply fetches the next instruction using an explicitly described memory and program counter.

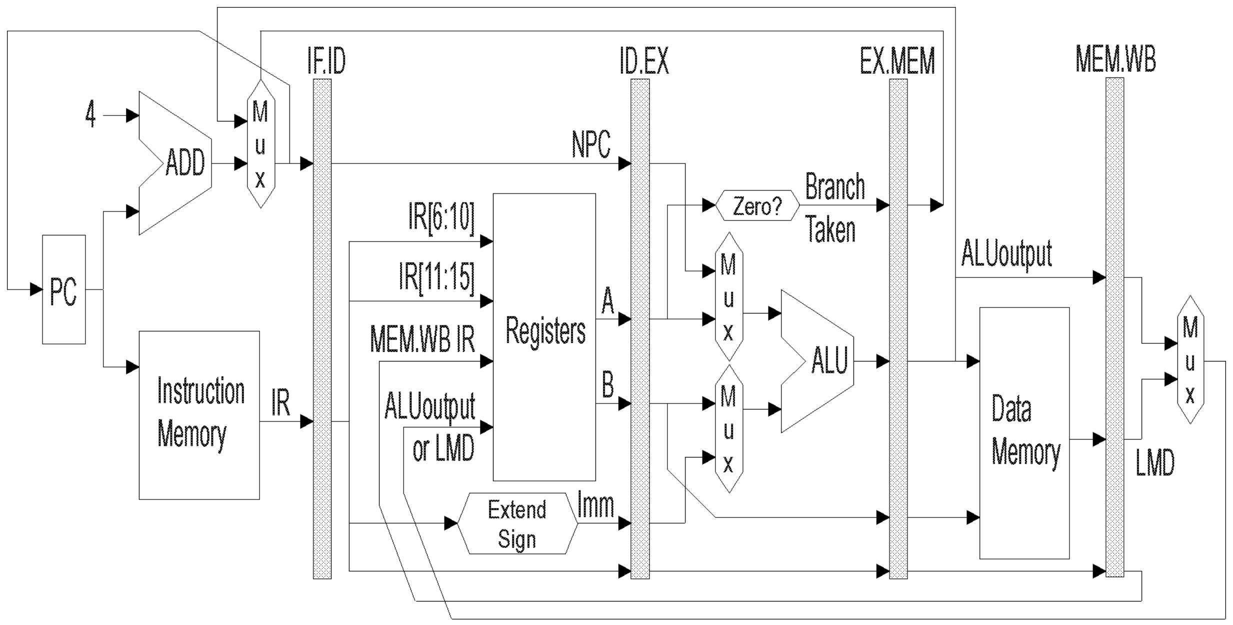

Figure 1. The simple DLX pipeline as shown in H&P figure 3.4.

3 Pipelines in RADL

When an inter-instruction hazard is detected, it is necessary to stall the pipeline. In RADL, a strategy table within the pipeline describes how to stall the pipeline, and a signals section describes when.

A strategy consists of a pipeline control signal together with a description of what to do to the pipeline stages to initiate the next machine cycle. As a simple example, consider the DLX pipeline described in Hennessy and Patterson (H&P) [12], see Figures 1, 2 and 3. The pipeline stages are IF, ID, EX, MEM and WB. If the detection of a read after write conflict between a load instruction in the EX phase and an instruction in the ID phase was signaled with, say, a 'load_raw' signal then we could write the strategy to perform the stall as follows:

```
        load_raw, ID: stall(NOP);
```

The pipeline strategy table is represented syntactically, as follows (ellipses indicate code not shown):

```
pipeline pipe1 { ...
  strategies {
    signal, IF(PC), ID, EX, MEM, WB;
    load_raw, ID: stall(NOP);
  } ... }
```

This is the strategy table for the pipeline, pipe1. The first list, headed by "signal" is not a strategy but, rather, it is just the title line for the strategy "table" and serves to name the pipeline stages. (The syntax shown is a stage keyword style -- the "title line" is more clear using a less compact alternative positional style of strategy description.) The "(PC)" following the "IF" stage name indicates that for the default smoothly flowing pipeline that the next instruction is fetched using the PC register resource. The memory to be used is indicated in a resources section not shown.

The signals, themselves, must also be declared in RADL. This declaration has two forms: a simple signal, and a composite signal. The composite signal is built up from a boolean expression of previously defined signals. For example, we might describe the signals needed for the above pipeline strategy table as follows (see Figure 3):

```
signals {
  cond; // Controls use of branch address.
 // The EX stage's load rd field == ID stage's rs1 field.
 //   "ex_rd" is an alias of "pipe1[ ID.EX ].IR[11:15]"
 //   similarly "id_rs1" is aliased to "pipe1[ IF.ID ].IR[6:10]"
 //   and "id_rs2" to "pipe1[ IF.ID ].IR[11:15]".
  ex_ld_rd_eq_id_rs1 == (ex_rd == id_rs1);
 // The EX stage's load rd field == ID stage's rs2 field.
  ex_ld_rd_eq_id_rs2 == (ex_rd == id_rs2);
  ex_load == load_instr.EX; // Load instruction in EX stage.
  id_branch == branch_instr.ID; // Branch instruction in ID stage.
  id_imm_alu == imm_alu_instr.ID; // Immediate alu instr in ID stg.
  id_mem == mem_instr.ID; // Load or store instruction in ID stage.
  id_reg_alu == reg_alu_instr.ID; // Reg-reg alu instr in ID stage.
 // Load interlock detection logic, from Fig. 3, rows 1-3, resp.
 // Detect a RAW hazard between ID and an EX load instr.
  load_raw1 == (ex_load && id_reg_alu && ex_ld_rd_eq_id_rs1);
  load_raw2 == (ex_load && id_reg_alu && ex_ld_rd_eq_id_rs2);
  load_raw3 == (ex_load && (id_mem || id_imm_alu || id_branch)
                            && ex_ld_rd_eq_id_rs1);
  load_raw == load_raw1 || load_raw2 || load_raw3;
}
```

Here, all but the first, the cond, signal are composite signals. They are composed of boolean expressions of most of the other signals. (The cond control signal is shown as a latch field in Figure 1 as "Branch Taken" but is represented as a RADL signal in our DLX example). Simple signals, such as cond, are set by behavioral code during a machine cycle. The ex_load signal and the four composite signals following it are set by the activation of the behavior of an operation (in this case an instruction) in a particular stage of the pipeline.

Signals do not need to be part of a pipeline. They can be used in non-pipelined descriptions as well. In a simulator, the composite pipeline signals are evaluated prior to the beginning of each machine cycle. Then the highest priority of the applicable strategies is chosen and the indicated adjustments to the pipeline stages are performed (e.g., whether to allow a smooth flow from one stage to the next, to stall a stage, or to inject an instruction into the stage). Following this strategy selection, the simple pipeline signals are reset to prepare for the new machine cycle.

In order to handle phase-accuracy, the number of machine-phases in a machine-cycle (i.e., in a pipeline stage) is specified for a pipeline, as follows:

```
pipeline pipe1 {
  phases_per_stage = 2; ...
}
```

Instruction, or instruction-part, behavior can be partitioned into different phases for the same stage so as to be able to read a register file in one phase and write to it in another phase. Another reason for this description element is to

Stage	**Any instruction**		

IF	IF/ID.IR ← Mem [PC];
	IF/ID.NPC,PC ← (if EX/MEM.cond {EX/MEM.ALUoutput} else {PC+4});		

ID	ID/EX.A ← Regs [IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs [IF/ID.IR$_{11..15}$];
	ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;
	ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;		

	ALU instruction	**Load or store instruction**	**Branch instruction**

EX	EX/MEM.IR ← ID/EX.IR;
	EX/MEM.ALUoutput ← ID/EX.A *op* ID/EX.B;
	or
	EX/MEM.ALUoutput ← ID/EX.A *op* ID/EX.Imm;
	EX/MEM.cond ← 0;
		EX/MEM.IR ← ID/EX.IR;
		EX/MEM.ALUoutput ← ID/EX.A + ID/EX.Imm;
		EX/MEM.cond ← 0;
		EX/MEM.B ← ID/EX.B;
			EX/MEM.ALUoutput ← ID/EX.NPC + ID/EX.Imm;
			EX/MEM.cond ← (ID/EX.A *op* 0);

MEM	MEM/WB.IR ← EX/MEM.IR;
	MEM/WB.ALUoutput ← EX/MEM.ALUoutput;
		MEM/WB.IR ← EX/MEM.IR;
		MEM/WB.LMD ← Mem[EX/MEM.ALUoutput];
		or
		Mem[EX/MEM.ALUoutput] ← EX/MEM.B;	

WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUoutput;
	or
	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUoutput;
		Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD;	

Figure 2. Events on every pipe stage of the DLX pipeline as shown in H&P Figure 3.5 with corrections.

provide for the possibility of other pipelines which are running at different (but synchronous) speeds. That is, they have a different number of phases per stage than that of a "main" pipeline.

In RADL, there is a latch register between each consecutive pair of stages. (As we will see, below, we can also have latch registers before or after all the stages). An instruction's typical stage behavior is to read from the latch before that stage and modify the data of the latch following that stage. A latch is named by the preceding and following stage names. For example, "ID.EX" is the name of the latch between the ID and EX stages, and the latch would be accessed in behavioral code by "pipe1[ID.EX]" in the case of our DLX pipeline example.

The RADL processor designer can specify the types of latch data for a pipeline as C data declarations. The syntax of the latch section's body is somewhat like a C language switch statement with sequences of C struct field declarations optionally prefixed by a colon-terminated sequence of latch names, as follows. The latch names indicate which fields belong to which latches. The default is for a latch field to belong to all latches. Here is the simple DLX pipeline latch definition, see Figure 2.

```
 pipeline pipe1 { ...
  latch {   // IR is used in all pipeline latch registers.
      feed unsigned int IR; // the latch field to feed from mem[PC].
    IF.ID, ID.EX: // applies only to IF.ID and ID.EX latches.
      unsigned int NPC; // the new PC value.
    ID.EX:: // applies only to ID.EX latch.
      // We need most latch fields to be both signed and unsigned:
      // we use "union nonsigned_int { int s; unsigned int u; };"
      nonsigned_int A; // the rs1 field's register value.
      nonsigned_int Imm; // an immediate value.
    ID.EX, EX.MEM:
      nonsigned_int B; // the rd or rs2 field's register value.
    EX.MEM, MEM.WB:
      nonsigned_int ALUoutput; // the ALU output value.
    MEM.WB:
      nonsigned_int LMD; // the loaded memory data value.
   } ... }
```

In the above example, the nonsigned_int is a C union (equivalent to a Pascal variant record) which allows us to use the bits either as a signed or unsigned integer, as needed. Note that the cond latch field of Figure 2 is represented not as a latch field but, instead, as a pipeline control signal. Also, the PC is represented as a register resource rather than as a latch field.

A pipeline strategy can feed an instruction into the latch field which is designated as the "feed" field. The data type of such a feed latch field must match the data type of the instruction memory addressed by the default strategy's program counter, PC in the example, and there can be only one such feed latch field associated with a given pipeline. The feed(<alt-pc>, <alt-memory>) tactic allows the use of multiple program counters (e.g.., in the example, not just PC) within a single pipeline — to support multiple threads, for example. To support instruction decompression, specialized memory resource read/write behavior can be supplied to override the default simple array access behavior.

RADL includes as a default the copying, or propagation, of data of common latch fields from the latch register before a stage to the latch register following. Thus, behavior code to indicate such copying need not be written. This default latch field propagation is overridden by actual assignment to a latch field. Here is an example of this from the simple DLX load-store instruction's behavior. Here, adder33 is a functional part of the DLX ALU. (The adder33's "1" arguments merely indicate that the arguments following are signed rather than unsigned.)

```
operation mem_instr { ...
  behavior.EX
   { // see Fig. 2, EX row, Load or store instruction section.
     // EX/MEM.IR <- ID/EX.IR is implicit in latch propagation.
     // do EX/MEM.ALUoutput <- ID/EX.A+ID/EX.Imm;
     pipe1[ EX.MEM ].ALUoutput
       = adder33( 1, pipe1[ ID.EX ].A,
                  1, pipe1[ ID.EX ].Imm );
     // EX/MEM.cond = 0 is implicit in machine cycle signal reset.
     // EX/MEM.B <- ID/EX.B is implicit in latch field propagation.
   } ... }
```

The "behavior.EX" section indicates behavioral code for the EX stage of the mem_instr operation (an hierarchically combined load and store instruction). If we had needed to specify a particular phase, say phase 2, of the EX stage, we could have named the section "behavior.EX.2", for example. Note that of the four assignment statements in figure 2's EX row, Load or store instruction section, only one assignment actually required representation in this RADL fragment.

Opcode field of ID/EX($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	$ID/EX.IR_{11..15} = IF/ID.IR_{6..10}$
Load	Register-register ALU	$ID/EX.IR_{11..15} = IF/ID.IR_{11..15}$
Load	Load, store, ALU immediate, or branch	$ID/EX.IR_{11..15} = IF/ID.IR_{6..10}$

Figure 3. The logic to detect the need for load interlocks during the ID stage of an instruction as shown in H&P Figure 3.18 with corrections.

3.1 Operation Hierarchy

In order to understand how to introduce multiple pipelines we need to make a brief excursion into hierarchical operations. A processor is described as a nested hierarchy of operations and sub-operations, as is done in nML [2]. The root of the hierarchy is marked as a "main" operation. The hierarchical relationships of operations are specified in the composition sections of those operations. This corresponds to nML's And- and Or-rules, except that in RADL we can combine both rules into a single composition statement. A simplified DLX example follows:

```
operation.main DLX_simple { // pipeline from Figs 2 & 3.
  composition {
     instr && adder33 && adderPC;
   }
  resources {
      unsigned int PC; // the DLX program counter.
      nonsigned_int Reg[ MAX_REG ]; // DLX registers.
      feed nonsigned_int Mem[ MAX_MEM ]; // DLX memory.
    }
  pipeline pipe1 { ... }
}
operation instr {
  composition {
      alu_instr || mem_instr || branch_instr;
    } ... }
operation alu_instr {
  composition {
      Imm_alu_instr || Reg_alu_instr;
    } ... }
```

The "||" indicates alternative specializations of the operation. That is, an instr can be either an alu_instr, a mem_instr or a branch_instr sub-operation. The "&&" indicates aggregation. The DLX_simple main operation includes four sub-operations representing instruction behavior, the instr operation, the two adders (the adder33 being part of the ALU). We can also nest aggregation and alternation if necessary as well as provide local name bindings to the sub-operations in case, for example, they are used more than once in the same aggregate.

3.2 Initiating a sub-pipeline

One pipeline can initiate behavior in another pipeline. An example might be the initiation of a floating-point (FP) add instruction from the main instruction pipeline in an FPU sub-pipeline. The DLX FPU's Add pipeline has four stages, shown below. The FPU_Add operation will also need to be added to the DLX_simple operation's composition as another aggregated part for this to work, of course.

```
operation FPU_Add { ...
  pipeline fpa_pipe { ...
      strategies { // CF H&P Fig. 3.44.
          signal, A1, A2, A3, A4;
           ... } ... } ... }
```

To control the injection of the FP Add instruction, we will rely on a main pipeline signal, FP_add. Such signals can be used to control communication between pipelines which do not have a parent-child relationship.

The instr operation's ID stage behavior can detect the presence of an FP add instruction and raise the main pipeline's FP_add signal.

```
operation instr {
  behavior.ID { ...
      if ((0b000010 == pipe1[ IF.ID ].IR[ 0:5 ])
          && (0b000100 == pipe1[ IF.ID ].IR[ 26:31 ]))
        pipe1.FP_add = true;
    } ... }
```

This behavior could also be achieved in a more declarative fashion using a composite signal in the pipe1 pipeline as follows:

```
signals { ...
  FP_add == (0b000010 == pipe1[[ IF.ID ].IR[ 0:5 ])
          && (0b000100 == pipe1[ IF.ID ].IR[ 26:31 ]);
}
```

The fpa_pipe pipeline strategy table will have a strategy associated with this pipe1.FP_add signal. This strategy will inject, or gate, several of the main pipeline ID.EX latch fields into the FPU_Add's corresponding latch fields before the A1 stage. To do this, the FPU_Add latch section is annotated to indicate that there will be a "before.A1" latch register. We also supply an "A4.after" latch register. This facilitates data communication to and from the main pipeline. We also will need to add three extra floating point fields to the main pipeline's latch structure and to dereference the operands as floats (as well as integers) in the ID stage into FP latch fields: FP_A, FP_B and FP_Imm.

Below is the FPU_Add operation with its pipeline and the stall tactic which invokes the fpa_in operation behavior which performs the inter-pipeline latch mapping for getting data into the fpa_pipe. Care must be taken in the fpa_in injection behavior to perform operations simple enough for compiler resource analysis.

A similar technique in the main pipeline will get the data out of the fpa_pipe. The behavior performing the injection runs prior to the start of a machine cycle -- it sets up the stage's "input" latch. Note that the fpa_in operation's behavior has no stage annotation suffix -- is not associated with a particular stage.

The "(NOP)" after the "A1" stage name doesn't indicate a program counter resource (for there is no need to fetch from memory to feed the instruction pipeline, here), but rather a NOP instruction which is to be run for the A1 stage as the default strategy. The default strategy inserts a NOP for the fpa_pipe pipeline's A1 stage's input latch, “before.A1”.

```
operation FPU_Add {
  composition {
     fp_add_instr && fpa_in;
    }
  pipeline fpa_pipe { ...
      strategies {
          signal, A1(NOP), A2, A3, A4;
          pipe1.FP_add, A1: stall(fpa_in);
        }
      latch with before, after { // includes both before.A1 and A4.after.
            feed unsigned int IR; // the instruction register to feed.
            float FP_A; // the rs1 field's FP register value.
            float FP_Imm; // a signed immediate value as a float.
            float FP_B; // the rd or rs2 field's FP register value.
            float FP_Output; // the FP_Add output value.
            float FP_LMD; // the FP loaded memory data value.
        } ...  } ... }
operation.inject fpa_in {
  behavior {
      fpa_pipe[ before.A1 ].IR = pipe1[ ID.EX ].IR;
      fpa_pipe[ before.A1 ].FP_A = pipe1[ ID.EX ].FP_A;
      fpa_pipe[ before.A1 ].FP_B = pipe1[[ ID.EX ].FP_B;
      fpa_pipe[ before.A1 ].FP_Imm  = pipe1[ ID.EX ].FP_Imm;
      fpa_pipe[ before.A1 ].FP_Output = 0;
      fpa_pipe[ before.A1 ].FP_LMD = 0;
    } ... }
```

When a non-NOP FP_result is produced in the A4.after latch register, we will connect it to the main pipeline via a control signal set by an fp_add_instr operation behavior (not shown). This signal controls another main pipeline strategy (also not shown) to handle the injecting of FPU_Add pipeline results from the A4.after latch back into the main pipeline EX.MEM latch register. These control and data connections will use the same techniques as used to invoke the FPU_Add pipeline from the main pipeline.

4 Conclusions

This paper has introduced a flexible and effective approach to the task of modeling processors with multiple pipelines for the purpose of retargetable production-quality programming tools. The focus has been primarily on cycle- and phase-accurate simulators. The main contributions are in the ease and flexibility of capturing pipeline behavior and in inter-pipeline control and data communications. Examples were presented of a machine description of the familiar and accessible DLX pedagogical processor described in Hennessy and Patterson [12].

RADL demonstrates that the description of pipelines in support of tool retargetability can be achieved in a straightforward and intuitive manner. Typical pipeline aspects such as control signals, stall strategies and latch registers are made explicit in RADL. Many of the tedious issues such as copying unchanged latch fields to the next latch register and resetting control signals are handled automatically. The pipeline representation is sufficiently close to typical processor descriptions (e.g., H&P figures in Chapter 3) that it is very easy to write a RADL description from such hardware descriptions. This improves both visibility into a processor description and the ease with which it can be modified.

Acknowledgments

I would like to thank the reviewers and also my colleagues from the Rockwell Design Automation Review Team for their many helpful comments and corrections: Karl Andersson, Lisa Guerra, Dan Pettyjohn, and Suresh Sureshchandran. I would also like to thank Vojin Zivojnovic (Axys GmbH) for initiating the RADL language effort while a Rockwell employee (“Rockwell Architecture Description Language (proposal)”, 26 June 1997, Rockwell Semiconductor Systems, Advanced VLSI Architectures Department internal document). The figures and source code examples in this paper are taken from the RADL specification, copyright 1998, Rockwell Semiconductor Systems, Inc.

References

1. Fauth, A., J. Van Praet and M. Freericks, “Describing Instruction Set Processors Using nML,” Proceedings of the European Design and Test Conference (ED&TC), Paris, March 1995, pp.503-507.
2. M. Freericks, “The nML Machine Description Formalism,” TU Berlin Computer Science Technical Report — Updated & Revised Version 1.5 (Draft), June, 1993.
3. Fauth, A. and A. Knoll, “Automated Generation of DSP Program Development Tools,” in Proceedings of the IEEE ICASSP-93, May 1993.
4. Fauth, A., “Beyond Tool-Specific Machine Descriptions,” in *Code Generation for Embedded Processors*, Marwedel and Goosens (Eds.), Kluwer Academic Publishers, 1995.
5. Hartoog, M., J. Rowson, P. Reddy, S. Desai, D. Dunlop, E. Harcourt and N. Khullar, “Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign,” Design Automation Conference, 1997.
6. Zivojnovic, V., S. Pees, C. Schlager and H. Meyr, “LISA — Machine Description Language and Generic Machine Model,” ICSPAT, Boston, 1997.
7. Nowak, L., “Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, MICRO-20, 1987, pp. 126-132.
8. Bashford, S. et al, “The MIMOLA Language Version 4.1,” Technical Report, Lehrstuhl Informatik XII, Univ. Dortmund, Setp. 1994.
9. Bradlee, D., R. Henry and S. Eggers, “The Marion System for Retargetable Instruction Scheduling,” Proceedings of the ACM SIGPLAN ‘91 Conference on Programming Language Design and Implementation, Toronto, Canada, June, 1991.
10. Liem, C., T. May and P. Paulin, “Instruction-Set Matching and Selection for DSP and ASIP Code Generation,” European Design and Test Conference (ED&TC), 1994, pp. 31-37.
11. Leupers, R. and P. Marwedel, “Instruction-Set Modelling for ASIP Code Generation,” 9th International Conference on VLSI Design, Bangalore, India, Jan. 1996.
12. Hennessy, J. and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1996, 2nd Edition.