Use of Architectural Simulation Tools in Education

Pradip Bose
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
Tel: (914) 945–3478, E–mail: bose@watson.ibm.com

ABSTRACT

This paper presents the author's experience in using architectural simulation tools in the instruction of computer architecture courses. In particular, we develop the notion of incrementally building a programmable, trace–driven "timer" tool, for use as a learning vehicle. We show how the cycle–by–cycle simulation output of such timers can be used to illustrate performance bottlenecks, and how this and other output statistics can be interpreted to convey key design tuning issues. As part of the overall simulation toolkit, we also use available cache simulators, trace generators and other utilities in illustrating key performance determinants and architectural trade–off issues.

I. Introduction:

Undergraduate or beginning graduate courses in computer architecture, such as those based on the well–known texts by Hennessey and Patterson [1, 2] often use a simple processor, e.g. DLX [1] as a running example to develop and illustrate key machine design concepts. Projects and assignments centered around the example processor, are crafted to enable the student to grasp alternate design and optimization strategies. These assignments are often paper exercises; thus, for example, a hardwired or microprogrammed control unit may be designed on paper, which is manually checked for correctness by the instructor and his aides. Even if software simulation aids are adopted, most of the class examples and home assignments stress logic design and minimization skills. Higher–level machine organization design and trade–off analysis methods are often not stressed adequately. Advanced computer architecture (graduate) courses, on the other hand, deal mostly with highly technical research papers, which usually examine specific, design issues and often employ complex analytical model–based reasoning. Consequently, students are often left with a gap in their understanding of

fundamental pipeline–stage level design trade–off issues. Until very recently, use of architectural simulation tools (e.g. SPIM [2]) was not prevalent.

This author has found it extremely useful to use such simulation tools in generating creative enlightenment among students. In this paper, we first describe (in section II), TRISC, a simple load/store instruction set architecture, implemented using a basic super scalar machine organization [3]. We have used this simple machine and its associated trace–driven "timer" quite successfully as an instructional aid. In fact, for some courses, we have encouraged a few students to develop the timer itself from scratch, as their course project. Other students have used a timer provided by the author, to study fundamental design trade–offs. In section III, we explain the basic software structure of the simple, parametrized timer used to study cycle–by–cycle pipelined execution states of the TRISC machine. In section IV, we illustrate the use of timer and other related simulation tools in explaining fundamental design trade–off issues. We conclude, in section V, by summarizing our experience in timer toolkit based instruction, and speculate on future trends.

II. The TRISC Architecture and Machine

The Instruction Set:

The TRISC architecture [3] is a simple, but extendable load/store instruction set, with 32 integer and floating point registers. It has fixed fields for opcode, register specifiers and immediate or displacement operands. The *core* ISA (instruction set architecture) consists of the following opcodes:

(A). Load instructions:
l, lu: integer LOAD, w/o update and w/ update.
lfd, lfd(u): floating point LOAD, w/o update and w/update
Only the integer loads are described below; the floating point loads are the same, except that they load a target *floating point* register, FRT.

Syntax:
l(u) RT, D(RA)

Semantics:
The l(u) instruction loads a word (32 bits) in storage from a specified location in memory addressed by the effective address (EA) into the target fixed point register, RT. For lu, the EA is stored into RA, if RA $\neq$ 0. EA = sum of the contents of RA and D, if RA $\neq$ 0. If RA = 0, EA = D. D is a 16–bit signed two's complement integer, sign extended to 32 bits.

Notation:
EA = c(RA|0) + D;
RT <— M(EA);
RA <— EA; /* if lu and if RA $\neq$ 0 */

(B) Store instructions:

st, stu: integer STORE, w/o update and w/update.

stf, stfu: floating point STORE, w/o update and w/update.

Only the integer stores are described below; the floating point stores are the same, except that they store a source floating point register, FRS.

Syntax

st(u) RS, D(RA)

Semantics:

The st(u) instruction stores the contents of fixed point register RS into the word of storage addressed by the EA. For stu, if RA $\neq$ 0, the EA is placed in RA.

Notation:

EA = c(RA|0) + D;

M(EA) <— c(RS);

RA <— EA; /* if stu and RA $\neq$ 0 */

(C) Functional instructions:

add, sub, mul, div: integer functional instructions, register–to–register.

fa, fs, fm, fd: floating point functional instructions, register–to–register.

Only the integer add instruction is described below; other integer functional instructions are the same except for the particular arithmetic operation. The floating point functional instructions are the same, except that they use floating point register operands, and imply floating point arithmetic operations..

Syntax:

add RT, RA, RB

Semantics:

The integer add instruction adds the integer operands in register specifiers RA and RB, and places the sum in the target register, specified by RT.

Notation:

RT <— c(RA) + c(RB) /* integer addition */

(D) Branch instructions:

b, bc: unconditional and conditional branch instructions.

Syntax:

b D

bc RC, D

Semantics:

The unconditional branch instruction (b) causes a change in program sequence by unconditionally jumping to the target address (TA). The conditional branch (bc) tests the value of the fixed point register specified by RC, which is used as a count register; the branch occurs if the value is non–zero, after pre–decrementing RC. The target address is computed by adding the branch displacement D to the program counter, i.e., the address of this branch instruction.

Notation for unconditional branch, b:
TA <— PC + D
PC <— TA /* PC is the program counter */

Notation for conditional branch, bc:
TA <— PC + D
RC <— RC – 1
if (RC $\neq$ 0) PC <— TA

Other instructions, like logical operations, and branches based on condition registers, are not described here. The ISA depicted here was used to study floating–point intensive, loop–oriented applications, where conditional branches are primarily loop–ending branches. The core ISA can easily be extended, if desired. However, we found this to be a very adequate core for beginning undergraduate courses, which did not go into elaborate branch prediction and resolution schemes, for example.

The Machine Organization:

We usually use a simple, super scalar processor model. In its simplest form (see Figure 1), intended for a beginning course in computer architecture, we assume a centralized, in–order instruction fetch/dispatch process, with three functional units: a branch unit (BRU), a fixed point unit (FXU), which processes integer arithmetic, as well as all load/store operations and a floating point unit (FPU). Initially, a perfect (infinite) cache model is assumed. (Later in the course, time permitting, finite cache models may be used). Re–order (completion) buffer mechanisms to enforce in–order completions (for precise interrupt support) is frequently omitted in an introductory course. Similarly, the concept of register renaming to eliminate certain kinds of data dependencies at run–time, is not introduced initially. Wherever possible, a particular hardware resource is parametrized in the simulation model (see next section) to enable trade–off and bottleneck analysis, which is the main intention in using or developing such a tool for class use.

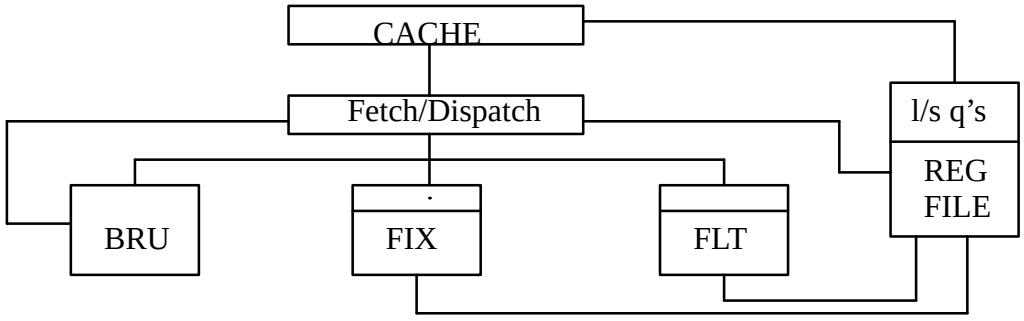

Figure 1. Simple, high–level super scalar machine organization

III. The TRISC Parametrized Timer

We usually use a classical, trace–driven, cycle–by–cycle simulation approach [5, 7] in implementing a class tool, or in using one for analysis purposes. We shall not go into the details of such timer implementation methods in this paper. The key steps and elements will be iluustrated in the actual talk. Here, we simply show (Figure 2) a sample cycle–by–cycle output for an input trace to illustrate the benefit of using such a tool for understanding pipeline stage level cycle–by–cycle behavior of such machines. For example, the degree of "slip" [6] between loads and their consuming operations in floating point loop kernels, and its sensitivity to overall performance, can be understood quite clearly from such "timeline" output.

TRISC timer cycle–by–cycle results

CY–CLE	I D	PROGRAM COUNTER	INSTRUCTION	DSP	FIX D	FIX A	FIX E	FIX P	FIX R	FLT D	FLT E/P
0	A	00001000	L 01 00 0004								
	B	00001004	LF 02 01 5000	A B							
1	C	00001008	LF 03 00 0008	B C	A						
2	D	00001012	FA 04 01 03	C D	B	A					
3				C D	B				A		
4	E	00001016	L 05 00 0004								
	F	00001020	STF 04 05 7000	E F	C	B				D	
5	G	00001024	L 06 00 0004	F G	E	C			B	D	
6	H	00001028	LF 07 06 5000	G H	F	E			C	D	
7				G H	F				E		D
8				G H	F						
9	I	00001032	LF 08 00 0008	H I	G	F					
10	J	00001036	FA 09 06 08	I J	H	G			F		
11				I J	H				G		
12	K	00001040	L 01 00 0004								
	L	00001044	STF 09 01 7000	K L	I	H				J	
13				L	K	I			H	J	
14					L	K			I	J	
15					L				K		J
16					L						
17						L					
18									L		

STATISTICS FOR THE TIMING RESULTS SHOWN

CPI = 1.462, CPF = 9.500, FIX utilization = 1.000, FLT utilization = 0.421

Figure 2. TRISC cycle–by–cycle output

In the above timer output, 'L' stands for an integer load; LF stands for a floating point load; 'FA' stands for a floating point add; 'STF' stands for a floating point store. The timeline output illustrates the pipeline flow of instructions (indicated by alpahebetic labels) through the shown functional units (dispatcher, fixed point unit (also the load–store unit), and floating point unit. For the parameter settings used for this example run, we show a 2–issue machine, with two shown positions in the dispatch buffer; a 3–stage load/store–cache pipe (D: decode, A: address generate; R: cache request); and a 2–stage floating point operation execution pipe (D: decode, E/P: execute/putaway). For the case shown we see that the effect of data dependencies in the context of the hardware parameters chosen, results in stalling of the floating point pipe.

The TRISC timer is invoked in association with the following inputs: (a) a parameter file which specifies settings of modifiable hardware parameters; (b) a program trace, generated by an instruction set simulator or some other trace generation mechanism (e.g. hand–tracing for simple loop kernels). It generates at least one output file, with cycle–by–cycle listing of the program execution timeline and various processor statistics. The main timer loop looks as follows (using a Pascal notation):

```
BEGIN  /* main program */
   init_system;   /* set up files and initialize variables */
   REPEAT
      print_summary;
      cur_cycle:= cur_cycle + 1;
      cache_access;
      fix;
      flt;
      dsp;
      compute_system_idle; /* sets Boolean variable system_idle */
  UNTIL end_of_trace AND system_idle;
  total_cycles := cur_cycle – 1;
  printstats;
  close_files;
END.  /* main program */
```

Initially, the parameter file is read in, and the system is initialized; The main timer loop then services each functional pipeline unit once every cycle, and also updates the timeline output file. The loop terminates after the input trace has been consumed and all instructions have been emptied from all the unit pipelines.

IV. Fundamental Issues: Tools–based Analysis

The primary use of such simulation tools in class, is clearly in studying the fundamental trade–offs which exist between machine organization parameters, in optimizing instructions–per–cycle (IPC) performance. In this section we mention a simple example to illustrate such use. In our experience, it is useful to encourage the students to first use analytical reasoning or "intuition" [5, 6] in answering a "what–if" question posed in class. They are then told to validate their reasoning via detailed simulation–based analysis.

Test kernel for testing overlapped (decoupled) access–execute: The additional test case [7].

```
do i = 1, n
  c(i) = a(i) + b(i)
enddo
```

The corresponding TRISC machine code sequence is:

```
LFU  0,  6, 0x8       /* floating load of ftlt reg 0, with update of the base int reg 6 */
LFU  1, 5,  0x8
FA    0, 0, 1         /* floating add of target flt reg 0; operands: flt reg 0 and 1 */
STFU 0, 4, 0x8        /* store flt reg 0, with update of the base int reg 4 */
BC                    /* conditionally branch back to top of loop */
```

Let us consider a case, where we assume presence of full register renaming. Let us also consider a 3–issue execution model, with an n–stage floating point execute pipe. In pipelined mode, the throughput of completed adds should be determined solely by the three load/store instructions. Since we are dealing with a single–ported (infinite) cache, the number of steady–state cycles per iteration should be 3; hence, with adequate number of queue resources, cycles–per–instruction (CPI) expected is: 3/5 = 0.6 and cycles–per–flop (CPF) is 3.0. Unrolling the loop will not further improve the CPF because of the limitation imposed by the single cache port. In a detailed timer model, the various resources, such as reservation station sizes, completion (reorder) buffer size, rename buffers, etc can be varied to measure variation of cpi and cpf. If, under large extensions of buffer size ranges, the student is unable to achieve the expected CPI bound of 0.6, he may have either exposed a timer model bug, or a limitation of some piece of the design logic, which is causing an unexpected stall in some unit. The cycle–by–cycle output can then be used to isolate the cause of the bottleneck. Table 1, shows an example summary of experimental timer–aided results for the above loop (with and without unrolling), which would be indicative of a performance bug in the machine. In the table, CBUF size refers to the size of the completion (reorder) buffer and FRBF size refers to the number of floating point rename buffers.

In addition to trace–driven timer models, other related tools useful in instruction are: trace generation and analysis programs, and cache simulators (e.g. [4]). The former are useful in computing various dynamic workload (i.e. application) statistics, such as instruction frequency mix or average basic block size. These statistics can be used by the students to correlate timer–generated performance numbers against expectations or bounds computed from those statistics. Cache simulators are used to compute average miss ratios and miss rates (misses per instruction) for given applications and cache geometries. From the miss rate and average miss penalty (in processor cycles), the finite cache effect (or cpi penalty) can be estimated by taking their product. In a full–blown timer model, the detailed cache access pipeline is modeled as part of the timer model, along with the effects of reload time, leading and trailing edge effects, etc. Such a model is able to compute the overall CPI performance much more accurately than the “averaging” method referred to above.

Table 1: CPI/CPF Variations: Addition Test Case

	Vanilla Loop		Unrolled (once)		Unrolled (twice)		Unrolled (thrice)	
CBUF SIZE	CPI	CPF	CPI	CPF	CPI	CPF	CPI	CPF
2	2.60	13.0	2.78	12.5	2.77	12.0		
4	1.80	9.0	1.77	8.0	1.78	7.66	1.71	7.24
6	1.20	6.0	1.44	6.5	1.38	6.0		
8	1.10	5.5	1.11	5.0	1.31	5.66	1.17	5.0
10	1.00	5.0	1.11	5.0	1.08	4.66		
12	1.00	5.0	1.11	5.0	1.00	4.34	1.00	4.25
16	1.00	5.0	1.11	5.0	0.923	4.0	0.88	3.74
20	1.00	5.0	1.11	5.0			0.77	3.26
24	1.00	5.0	1.11	5.0	0.923	4.0	0.77	3.26
32	1.00	5.0	1.11	5.0			0.77	3.26
FRBF SIZE	CPI	CPF	CPI	CPF	CPI	CPF	CPI	CPF
4	1.20	6.0	1.44	6.5	1.54	6.66	1.47	6.22
8	1.00	5.0	1.11	5.0	1.04	4.50	1.12	4.75
12	1.00	5.0	1.11	5.0	0.923	4.0	0.88	3.74
16	1.00	5.0	1.11	5.0	0.923	4.0	0.88	3.74

V. Conclusion

We have described the use of simple architectural simulation models in class instruction. We have illustrated how such tools can be used to visualize pipeline–stage level concurrency and performance bottlenecks. Also, we have shown how such models may be used to validate or correct intutive student reasonings about fundamental organizational trade–offs. For advanced architecture courses, in which state–of–the–art vendor microprocessors may be studied, an available or constructed timer tool should preferably be instrumented to mark up the cycle–by–cycle timeline output by “stall codes”: symbols to classify various types of pipeline stall conditions. Such an instrumented tool minimizes diagnosis time for isolating a model defect or a performance bug in the design itself.

REFERENCES

1. David A. Patterson and John L. Hennessy, “Computer Architecture: A Quantitative Approach,” Morgan Kaufmann Publishers, Inc., 1990.

2. John L. Hennessy and David A. Patterson, Computer Organization and Design, Morgan Kaufmann Publishers, 1994.

3. P. Bose, “The TRISC architecture, super scalar organization and its timer,” IBM internal unpublished document, Nov. 1991.

4. K. So et al, “The KSIM hierarchical, multi–configuration cache simulatoe,” IBM internal software; also, Wen–Hann Wang and Jean–Loup Baer, “Efficient trace–driven simulation methods for cache performance analysis,” *ACM Trans. on Computer Systems,* vol. 9, no. 3, pp. 222–241, August 1991.

5. P. Bose, “Early performance estimation of super scalar machines,” *Proc. Int’l. Conf. on Computer Design (ICCD),* pp. 388–392, Oct. 1991.

6. W. Mangione–Smith, T.–P. Shieh, S. G. Abraham and E. S. Davidson, “Approaching a machine application–bound in delivered performance on scientific code,” *Proc. IEEE,* 81, 1166–1178, Aug. 1993.

7. P. Bose and S. Surya, “Architectural timing verification of CMOS RISC processors,” IBM Journ. of Res. and Develop., vol. 39, no. 1/2, pp. 113–129, January/March 1995.