

GEORGIA INSTITUTE OF TECHNOLOGY

College of Computing

CS6290/CS4290 — High-Performance Computer Architecture

Fall 2001

CS6290/CS4290
Homework 4

Issued: September 30, 2001
Due: October 19, 2001

- Purpose:** This homework covers hardware scheduling for instruction-level parallel architectures.
- Reading:** H&P Chapter 4, particularly Sections 4.2, 4.3 and 4.4. [Smith88]
- Problems:**
1. Extracting ILP in Hardware.
 2. Precise Exceptions [Smith88].
- Appendix:** State table for Tomasulo's algorithm (also on-line)

Problem 1: Extracting ILP in Hardware

Use the following program fragment in DLX assembly language for the questions in the problem. The program computes the dot product of two vectors.

```
; R1 and R2 point to the vectors, F2 is initialized to zero.
;-----
loop:  LD      F0, 0(R1)
      LD      F4, 0(R2)
      MULTD   F0, F0, F4
      ADDD   F2, F0, F2
      SUBI   R1, R1, #8
      SUBI   R2, R2, #8
      BNEZ   R1, loop
;-----
; result in F2
```

For this problem assume the following delay characteristics. This table is similar to Figure 4.2 in the text with the exception that branches have *no* delay slots or other stalls associated with them. Instead, branches are predicted taken.

| Operations | Stall Cycles (for plain DLX) | Pipeline Stages (for Tomasulo) |
|------------|---------------------------------|-----------------------------------|
| ----- | ----- | ----- |
| FOP -> FOP | 3 | 4 |
| LD -> FOP | 1 | 2 |
| IOP -> IOP | 0 | 1 |
| branch | 0 | --- (branches are predicted) |

For reference, the code has written has a few stalls:

```
; R1 and R2 point to the vectors, F2 is initialized to zero.
;-----
loop:  LD      F0, 0(R1)
      LD      F4, 0(R2)
      <stall>                ; LD -> MULTD through F4
      MULTD   F0, F0, F4
      <stall>                ; MULTD -> ADDD through F0
      <stall>
      <stall>
      ADDD    F2, F0, F2
      SUBI    R1, R1, #8
      SUBI    R2, R2, #8
      BNEZ    R1, loop
;-----
; result in F2
```

A: For the rest of the problem you'll see how dynamic scheduling hardware improves performance. Let's start by computing the baseline performance with the stalls above.

- i. How many cycles are required to execute *one* iteration of the original loop? Assume the vectors are very long.
- ii. In scientific codes (like this one), one often takes the view that only floating point instructions are doing any real work and the rest of the instructions represent evil necessary only to get the job done. How many Floating Point Operations Per Cycle (FOPPCs) are achieved by the original loop?

Note that the rate of execution is limited by the number of instructions and by stalls due to data dependencies.

Note, as in Homework 4, it is possible to eliminate the stalls through clever scheduling. Here's one solution. Note the slight difference between this and the solutions to Homework 4 because here there is no delay slot after the branch.

```

; R1 and R2 point to the vectors, F2 is initialized to zero.
    MOVED    F0, 0.0          ; prologue: first ADDD is a dummy
;-----
loop:  LD     F4, 0(R2)
      ADDD   F2, F0, F2      ; ADDD for *previous* iteration
      LD     F0, 0(R1)
      SUBI   R1, R1, #8      ; was a stall
      MULTD  F0, F0, F4
      SUBI   R2, R2, #8      ; was a stall
      BNEZ   R1, loop        ; was a stall
;-----
      <stall>                ; MULTD -> ADDD, last iteration only
      ADDD   F2, F0, F2      ; epilogue: do last ADDD
; result in F2

```

B:

- i. How many cycles per iteration are required by the rescheduled loop above?
- ii. How many FOPPCs are achieved by the the rescheduled loop? Note that the rate of execution is limited by the time to issue instructions only – we've eliminated all stalls.

Next, consider a pipelined machine controlled by Tomasulo's algorithm. See the attached worksheet for the bookkeeping data structures implied.

Assume:

- Only floating point instructions are controlled by Tomasulo's algorithm, i.e., only floating-point numbers pass over the Common Data Bus (CDB). Integer instructions consume issue slots but do not appear in the rest of the instruction-state table.
- One whole cycle is required to send a result of the CDB. In other words, Tomasulo's algorithm introduces an extra cycle of latency over a static pipeline. For instance, note in the example instruction-state table below that the MULTD begins executing in cycle 6, an extra cycle after the *LD* on which it is waiting.
- Assume that structural stalls due to the single CDB are resolved *after* the execution unit.
- If you find that you have to make other assumptions, list 'em.

Example instruction-state table

```

-----
instr.  issue  exec   write
-----
LD      1      2-3    4
LD      2      3-4    5
MULTD   3      6-9    10
ADDD    4      ...     ...
ADDI    5                                ; exec/write unused
...

```

C: Simulate the execution of the original code in a pipeline controlled by Tomasulo's algorithm.

- Use the attached worksheet to show the detailed state of execution at the time the **BNEQ** instruction issues for the *second* time.
- How many cycles are required to execute *one* iteration of the loop in a long vector? To figure out this question, you need to simulate through the second iteration of the loop (or better yet the third to really convince yourself). Show the instruction-state table you use to answer this part.
- How many FOPPCs are achieved by the hardware?
- What limits the rate of execution of the loop?

Next, consider a 4-way superscalar machine using Tomasulo’s algorithm (something like the Alpha 21264 described in [Gwennap96]) and executing the original loop.

We haven’t properly discussed superscalars other than showing a few real-world examples. However, for the purposes of this problem, the idea is really simple: a 4-way superscalar with dynamic scheduling follows the same rules as the Tomasulo machine in the part (C), except that it is allowed to issue *four* instructions per cycle. The instructions may be of any type.

Assume unlimited resources (e.g. multiple CDBs), other than the restriction that only four instruction may issue per cycle.

Note: the quick way to simulate the superscalar is to fill out only the “instruction status” table from the Tomasulo worksheet. In the worksheet, assume that four instructions are allowed to issue per cycle, e.g.:

| instr. | issue | exec | write | |
|--------|-------|-------|-------|---------------------|
| ----- | ----- | ----- | ----- | |
| LD | 1 | 2-3 | 4 | |
| LD | 1 | 2-3 | 4 | |
| MULTD | 1 | 5-8 | 9 | |
| ADDD | 1 | ... | ... | |
| ADDI | 2 | | | ; exec/write unused |
| ... | | | | |

D: Given your simulation of the 4-way superscalar on the original code:

- i. How many cycles are required to execute *one* iteration of the loop (one out of many, as usual)? Show the instruction-state table you use to figure out the answer.
- ii. How many FOPPCs does your code achieve? Assume a very long vector.
- iii. What limits the speed of execution of the loop?

Dynamic scheduling hardware is commonplace but some designs are still using static, in-order scheduling. The reasoning is that the dynamic scheduling hardware is so complex that it many slow down the clock rate of the processor. The last two questions will look at the consequences of sticking with an in-order design.

The last machine to consider is a 4-way superscalar that supports only in-order instruction execution (something like the UltraSPARC-III described in [Song97]). The machine can execute up to four sequential instructions in a cycle *as long as there are no dependencies between them*. For instance, the original loop on such a machine would start out executing as follows (this is the *execution trace*):

```
loop:  LD      F2, 0(R2)      |      LD      F4, 0(R4)
      <stall>
      MULTD   F2, F2, F4
      ...
```

The machine attempts to start four instruction on the first cycle but can only start two before running into a dependence. Since instructions must execute in order, no other instructions may execute until the dependence is resolved.

Assume:

- All four units are identical.
- Only one branch instruction is allowed per four.
- Pipeline delays and resulting stalls are the same as for the vanilla DLX, I.e. the original code would suffer stalls as indicated in the trace above. There is a minor benefit here over an out-of-order machine: the in-order machine does not add an extra cycle of latency for a CDB.
- Note that on a stall, *no* new instructions issue (zero out of a possible four!). Thus it is hugely important to eliminate stalls. This is the major drawback of an in-order design.

E: For the static 4-way superscalar hardware:

- i. Show the execution trace for executing the original loop.
- ii. How many cycles are required to execute *one* iteration of the loop? I.e. one loop out of many, once you get started.
- iii. How many FOPPCs does your code achieve in a long vector?

Finally, rewrite the dot-product loop to target the static 4-way superscalar hardware. Feel free to use unrolling, software pipelining and anything else you can think of. You may transform the linear sequence of additions into a “tree” as suggested on one of the slides in Lecture, if you like.*

- F:** For your rewritten loop:
- i. Show your new code. You need only show the body of the loop in detail.
 - ii. How many cycles are required to execute *one* iteration of your new loop? I.e. one loop out of many, once you get started.
 - iii. How many FOPPCs does your code achieve? Assume a very long vector.
 - iv. **OPTIONAL:** loop scheduling generally introduces prologue and epilogue code, sometimes gobs of it. This extra code reduces the performance of the machine for short vectors. Considering the extra code (and stalls therein), what length of vector is required to achieve *half* the FOPPCs you found in part iii? This vector length is called “ $N_{\frac{1}{2}}$ ” and is widely used to characterize the “agility” of vector machines. Current supercomputers often exhibit $N_{\frac{1}{2}}$ of 100 or more.

* The tree trick isn’t entirely fair with floating point addition because floating point addition is not associative due to rounding. We’ll ignore the rounding problem.

Problem 2: Precise Exceptions

[Smith88] J. E. Smith and A. R. Pleszkun, “Implementing Precise Interrupts in Pipelined Processors”, *IEEE Transactions on Computers*, 37(5), pages 562-573, May, 1988.

The [Smith88] paper catalogs several ways to achieve precise exceptions, including “re-order buffers”, “history buffers” and a “future file”. Last year, when the ISCA conference was in Atlanta, Prof. Smith was awarded the Eckert-Mauchly Award* by the computer architecture community in part for his early work on precise exception handling (this paper). One or another of the ideas here appears in every superscalar design. Amusingly, in his talk, Smith displayed part of a four-page negative review he received for this paper which explained in lengthy detail why the problem should be considered *irrelevant* based on alternate solutions existing at the time.

A: What are the principle advantages of a “future file” over a “history buffer”.

* Eckert and Mauchly developed the ENIAC and then started a company to develop the first commercial computer, UNIVAC. The Eckert-Mauchly award has been presented yearly since 1979. See the whole list at <http://computer.org/awards/awdem.htm>