CS152
Computer Architecture and Engineering

Complex Pipelines
Assigned 10/11/2016 — Problem Set #3 — *Due October 18*

http://inst.eecs.berkeley.edu/~cs152/fa16

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in his own solution to the problems.

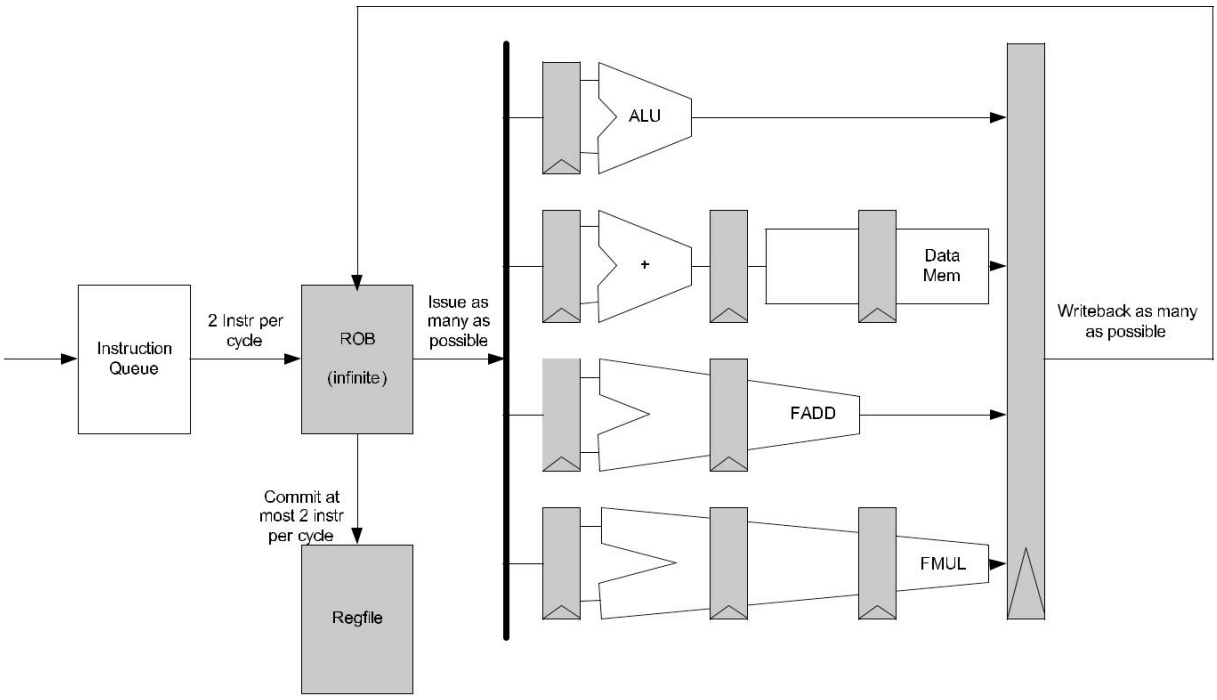
The problem sets also provide essential background material for the quizzes. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the quizzes! Homework assignments are due at the beginning of class on the due date. Late homework will not be accepted.

Problem 1: Superscalar Processor

Consider the out-of-order, superscalar CPU shown in the diagram. It has the following features:

- Four fully-pipelined functional units: ALU, MEM, FADD, FMUL
- Instruction Fetch and Decode Unit that renames and sends 2 instructions per cycle to the ROB (assume perfect branch prediction and no cache misses)
- An unbounded length Reorder Buffer that can perform the following operations on every cycle:
  - Accept two instructions from the Instruction Fetch and Decode Unit
  - Dispatch an instruction to each functional unit including Data Memory
  - Let Writeback update an unlimited number of entries
  - Commit up to 2 instructions in-order
- There is no bypassing or short circuiting. For example, data entering the ROB cannot be passed on to the functional units or committed in the same cycle.

.



Now consider the execution of the following program on this machine using:

```
I1      loop:   LD F2, 0(x2)
I2              LD F3, 0(x3)
I3              FMUL F4, F2, F3
I4              LD F2, 4(x2)
I5              LD F3, 4(x3)
I6              FMUL F5, F2, F3
I7              FMUL F6, F4, F5
I8              FADD F4, F4, F5
I9              FMUL F6, F4, F5
I10             FADD F1, F1, F6
I11             ADD x2, x2, 8
I12             ADD x3, x3, 8
I13             ADD x4, x4, -1
I14             BNEZ x4, loop
```

(x refers to general purpose registers. F is floating point registers)

Problem 1.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10. You shouldn't have to use more than 10 renaming registers (T1 to T10). You can leave the two pre-filled entries in the renaming table intact.

Tags should not be reused.

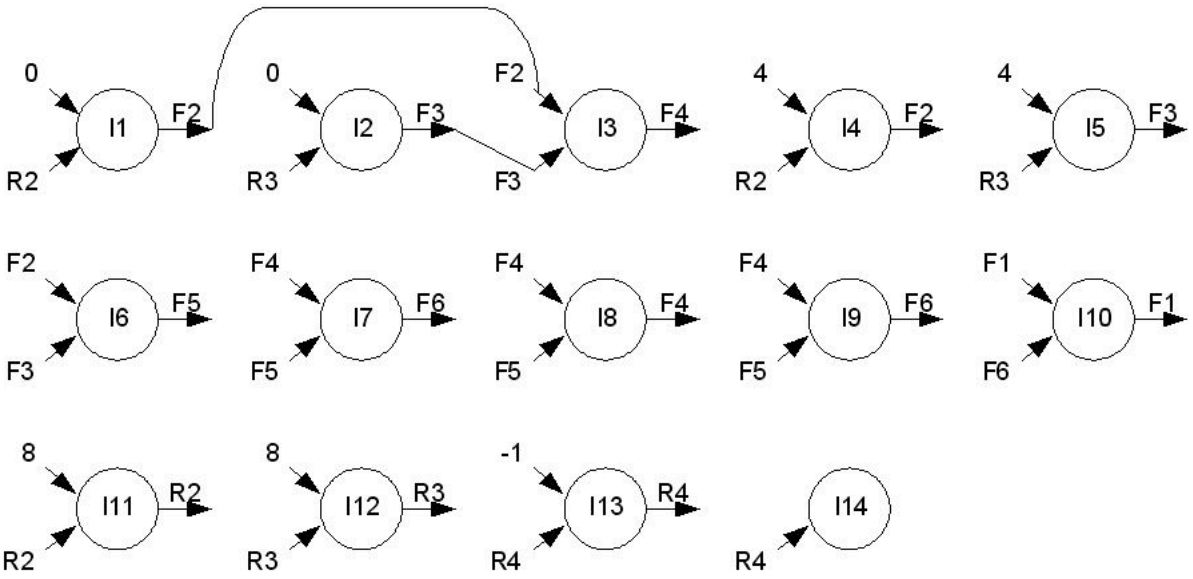
Instr #	**Instruction**	**Dest**	**Src1**	**Src2**
I1	LD F2, 0(x2)	T1	x2	0
I2	LD F3, 0(x3)	T2	x3	0
I3	FMUL F4, F2, F3			
I4	LD F2, 4(x2)		x2	4
I5	LD F3, 4(x3)		x3	4
I6	FMUL F5, F2, F3			
I7	FMUL F6, F4, F5			
I8	FADD F4, F4, F5			
I9	FMUL F6, F4, F5			
I10	FADD F1, F1, F6		F1	

Renaming table.

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
x2										
x3										
F1										
F2	T1									
F3		T2								
F4										
F5										
F6										

Problem 1.B

Consider the execution of ***one*** iteration of the loop (I1 to I14). In the following diagram draw the data dependencies between the instructions after register renaming ("R" and "x" denote the same registers).



Problem 1.C

The attached table is a data structure to record the times when some activity takes place in the ROB. For example, one column records the time when an instruction enters ROB, while the last two columns record, respectively, the time when an instruction is dispatched to the FU's and the time when results are written back to the ROB. This data structure has been designed to test your understanding of how a Superscalar machine functions.

Fill in the blanks (You may use the source columns for book keeping – no credit will be taken off for the wrong entries there).

Slot	Instruction	Cycle instruction entered ROB	Argument 1		Argument 2		dst	***C y c l e dispatched***	***Cycle written back to ROB***
			src1	cycle available	Src2	cycle available	dst reg		
T1	`LD F2, 0(x2)`	1	C	1	x2	1	F2	2	6
T2	`LD F3, 0(x3)`	1	C	1	x3	1	F3	3	7
T3	`FMUL F4, F2, F3`	2					F4		
T4	`LD F2, 4(x2)`	2	C	2	x2	2	F2		
T5	`LD F3, 4(x3)`	3	C	3	x3	3	F3		
T6	`FMUL F5, F2, F3`	3					F5		
T7	`FMUL F6, F4, F5`	4					F6		
T8	`FADD F4, F4, F5`	4					F4		
T9	`FMUL F6, F4, F5`	5					F6		
T10	`FADD F1, F1, F6`	5					F1		
T11	`ADD x2, x2, 8`	6	x2	6	C	6	x2		
T12	`ADD x3, x3, 8`	6	x3	6	C	6	x3		
T13	`ADD x4, x4, -1`	7	x4	7	C	7	x4		
T14	`BNE x4, x0, loop`	7			C	Loop			
T15	`LD F2, 0(x2)`	8	C	8	x2	9	F2		
T16	`LD F3, 0(x3)`	8	C	8	x3	10	F3		
T17	`FMUL F4, F2, F3`	9					F4		
T18	`LD F2, 4(x2)`	9	C	9	x2	9	F2		
T19	`LD F3, 4(x3)`	10	C	10	x3	10	F3		
T20	`FMUL F5, F2, F3`	10					F5		
T21	`FMUL F6, F4, F5`	11					F6		
T22	`FADD F4, F4, F5`	11					F4		
T23	`FMUL F6, F4, F5`	12					F6		
T24	`FADD F1, F1, F6`	12					F1		
T25	`ADD x2, x2, 8`	13			C	13	x2		
T26	`ADD x3, x3, 8`	13			C	13	x3		
T27	`ADD x4, x4, -1`	14			C	14	x4		
T28	`BNE x4, x0, loop`	14			C	Loop			
T29									

Problem 1.D

Identify the instructions along the longest latency path in completing this iteration of the loop (up to instruction 13). Suppose we consider an instruction to have executed when its result is available in the ROB. How many cycles does this iteration take to execute?

Problem 1.E

Do you expect the same behavior, i.e., the same dependencies and the same number of cycles, for the next iteration? (You may use the slots from T15 onwards in the attached diagram for bookkeeping to answer this question). Please give a simple reason why the behavior may repeat, or identify a resource bottleneck or dependency that may preclude the repetition of the behavior.

Problem 1.F

Can you improve the performance by adding at most one additional memory port and a FP Multiplier? Explain briefly.

Yes / No

Problem 1.G

What is the minimum number of cycles needed to execute a typical iteration of this loop if we keep the same latencies for all the units but are allowed to use as many FUs and memory ports and are allowed to fetch and commit as many instructions as we want.

Problem 2: Register Renaming and Static vs. Dynamic Scheduling

The following MIPS code calculates the floating-point expression E = A * B + C * D, where the addresses of A, B, C, D, and E are stored in x1, x2, x3, x4, and x5, respectively:

```
L.S     F0, 0(x1)
L.S     F1, 0(x2)
MUL.S   F0, F0, F1
L.S     F2, 0(x3)
L.S     F3, 0(x4)
MUL.S   F2, F2, F3
ADD.S   F0, F0, F2
S.S     F0, 0(x5)
```

Problem 2.A — Simple Pipeline

Calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive) on a simple in-order pipelined machine that has no bypassing. The datapath includes a load/store unit, a floating-point adder, and a floating-point multiplier. Assume that loads have a two-cycle latency, floating-point multiplication has a four-cycle latency and floating-point addition has a two-cycle latency. Write-back for floating-point registers takes one cycle (i.e., it takes one cycle to write a floating point register). Also assume that all functional units are fully pipelined and ignore any write back conflicts. Give the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive.

Cycle	Decoded Instruction (Enters Issue)	Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
0	`L.S F0, 0(x1)`	Stall	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

13			
14			
15			
16			
17			
18			

Problem 2.B **Static Scheduling**

Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the simple in-order pipeline.

Problem 2.C **Fewer Registers**

Rewrite the code sequence, but now using only two floating-point registers. Optimize for minimum run-time. You may need to use temporary memory locations to hold intermediate values (this process is called register-spilling when done by a compiler). List the code sequence and give the number of cycles this takes to execute.

Problem 2.D **Register renaming and dynamic scheduling**

Calculate the effect of running the original code on a single-issue machine with register renaming and out-of-order issue. Ignore structural hazards apart from the single instruction decode per cycle (for example, you have infinite functional units and read/write ports). Show how the code is executed and give the number of cycles required. Compare it with results from optimized execution in 2.B.

Problem 2.E **Effect of Register Spills**

Now calculate the effect of running code you wrote in 2.C on the single-issue machine with register renaming and out-of-order issue from 2.D. Compare the number of cycles required to execute the program. What are the differences in the program and/or architecture that change the number of cycles required to execute the program? You should assume that all load instructions before a store must issue before the store is issued, and load instructions after a store must wait for the store to issue.

Problem 3: Importance of Features

For the following snippets of code, select the single architectural feature that will *most* improve the performance of the code. *Explain your choice*, including description of why the other features will not improve performance as much and your assumptions about the machine design. The features you have to choose from are: out-of-order issue with renaming, branch prediction, and superscalar execution. Loads are marked whether they hit or miss in the cache.

Problem 3.A

```
ADD.D F0, F1, F8
ADD.D F2, F3, F8
ADD.D F4, F5, F8
ADD.D F6, F7, F8
```

Circle one:

- ☐ Out-of-Order Issue with Renaming
- ☐ Branch Prediction
- ☐ Superscalar

Problem 3.B

```
loop: ADD R3 R4 R0
      LD R4, 8(R4) # cache hit
      BNEQZ R4, LOOP
```

Circle one:

- ☐ Out-of-Order Issue with Renaming
- ☐ Branch Prediction
- ☐ Superscalar

Problem Q3.C

```
LD  R1 0(R2)  # cache miss
ADD R2 R1 R1
LD R1 0(R3) # cache hit
LD R3 0(R4) # cache hit
ADD R3 R1 R3
ADD R1 R2 R3
```

Circle one:

- ☐ Out-of-Order Issue with Renaming
- ☐ Branch Prediction
- ☐ Superscalar

Problem 4: Out-of-order Machine Design

An out-of-order superscalar processor uses a unified physical register file for register renaming and also separates the reorder buffer from the instruction window. During decode, instructions are allocated a slot in the reorder buffer, have their registers renamed, and then are placed in the instruction window to await issue into execution.

Part A) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **instruction window entries**? Explain.

Part B) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **physical registers**? Explain.

Part C) Should the number of **instruction window entries** be greater than, equal to, or less than the number of **physical registers**? Explain.