CS152
Computer Architecture and Engineering

Out of Order Execution and Branch Prediction
Problem Set #4

Assigned March 17 | *Due April 2*

http://inst.eecs.berkeley.edu/~cs152/sp09

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 their own solutions 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! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Homework will not be accepted once solutions are handed out.

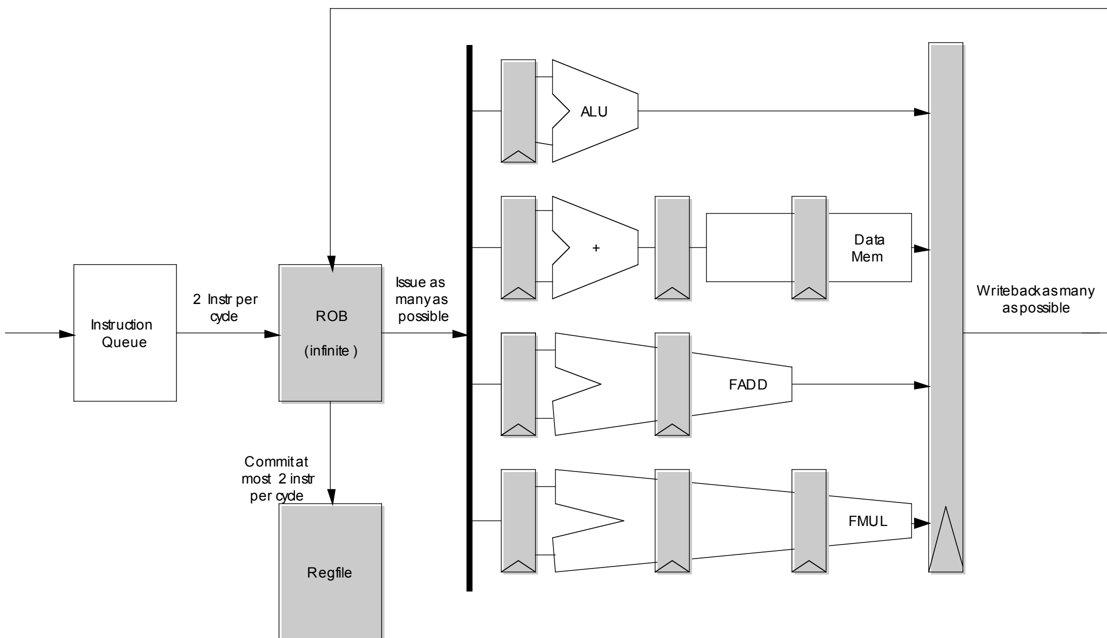
This material will be on Quiz 4, not Quiz 3.

Problem 4.1: Superscalar Processor

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

- o Four fully-pipelined functional units: ALU, MEM, FADD, FMUL
- o Instruction Fetch and Decode Unit that renames and sends 2 instructions per cycle to the ROB (assume perfect branch prediction and no cache misses)
- o An unbounded length Reorder Buffer that can perform the following operations on every cycle:
  - o Accept two instructions from the Instruction Fetch and Decode Unit
  - o Dispatch an instruction to each functional unit including Data Memory
  - o Let Writeback update an unlimited number of entries
  - o Commit up to 2 instructions in-order
- o 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(R2)
I2                LD F3, 0(R3)
I3                FMUL F4, F2, F3
I4                LD F2, 4(R2)
I5                LD F3, 4(R3)
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 R2, R2, 8
I12               ADD R3, R3, 8
I13               ADD R4, R4, -1
I14               BNEZ R4, loop
```

Problem 4.1.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10. *Tags should not be reused.*

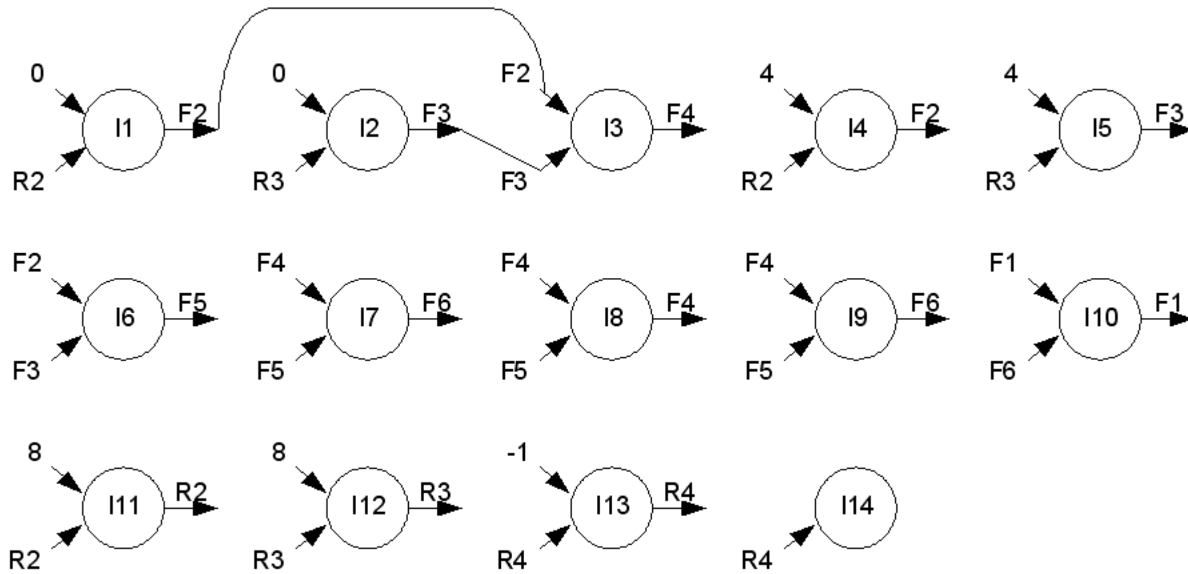
Instr #	**Instruction**	**Dest**	**Src1**	**Src2**
I1	LD F2, 0(R2)	T1	R2	0
I2	LD F3, 0(R3)	T2	R3	0
I3	FMUL F4, F2, F3			
I4	LD F2, 4(R2)		R2	4
I5	LD F3, 4(R3)		R3	4
I6	FMUL F5, F2, F3			
I7	FMUL F6, F4, F5			
I8	FADD F4, F4, F5			
I9	FMUL F6, F4, F5			
I10	FADD F1, F1, F6		F1	

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
R2										
R3										
F1										
F2	T1									
F3		T2								
F4										
F5										
F6										

Problem 4.1.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



Problem 4.1.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 in last two columns up to slot T13 (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	***Cycle dispatched***	***Cycle written back to ROB***
			src1	cycle available	Src2	cycle available	dst reg		
T1	**LD F2, 0(R2)**	1	C	1	R2	1	F2	2	6
T2	**LD F3, 0(R3)**	1	C	1	R3	1	F3	3	7
T3	**FMUL F4, F2, F3**	2			F3	7	F4		
T4	**LD F2, 4(R2)**	2	C	2	R2		F2		
T5	**LD F3, 4(R3)**	3	C	3	R3		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 R2, R2, 8**	6	R2	6	C	6	R2		
T12	**ADD R3, R3, 8**	6	R3	6	C	6	R3		
T13	**ADD R4, R4, -1**	7	R4	7	C	7	R4		
T14	**BNEZ R4, loop**	7			C	Loop			
T15	**LD F2, 0(R2)**	8	C	8			F2	10	14
T16	**LD F3, 0(R3)**	8	C	8			F3	11	15
T17	**FMUL F4, F2, F3**	9					F4		
T18	**LD F2, 4(R2)**	9	C	9			F2		
T19	**LD F3, 4(R3)**	10	C	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 R2, R2, 8**	13			C	13	R2		
T26	**ADD R3, R3, 8**	13			C	13	R3		
T27	**ADD R4, R4, -1**	14			C	14	R4		
T28	**BNEZ R4, loop**	14			C	Loop			

Problem 4.1.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 4.1.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 4.1.F

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

Yes / No

Problem 4.1.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 4.2: 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 R1, R2, R3, R4, and R5, respectively:

```
L.S     F0, 0(R1)
L.S     F1, 0(R2)
MUL.S   F0, F0, F1
L.S     F2, 0(R3)
L.S     F3, 0(R4)
MUL.S   F2, F2, F3
ADD.S   F0, F0, F2
S.S     F0, 0(R5)
```

Problem 4.2.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. 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.

Problem 4.2.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 4.2.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 4.2.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. Show how the code is executed and give the number of cycles required. Compare it with results from optimized execution in 4.2.B.

Problem 4.2E — Effect of Register Spills

Now calculate the effect of running code you wrote in 4.2.C on the single-issue machine with register renaming and out-of-order issue from 4.3.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 4.3: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program:

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, 4
     SUBI R1, R1, 1
b1:
     BEQZ R4, b2
     ADDI R2, R2, 1
b2:
     BNEZ R1, loop
```

Assume the initial value of R1 is n (n>0).
Assume the initial value of R2 is 0 (R2 holds the result of the program).
Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in lecture. We will be using a 2-bit predictor state machine, as shown below.

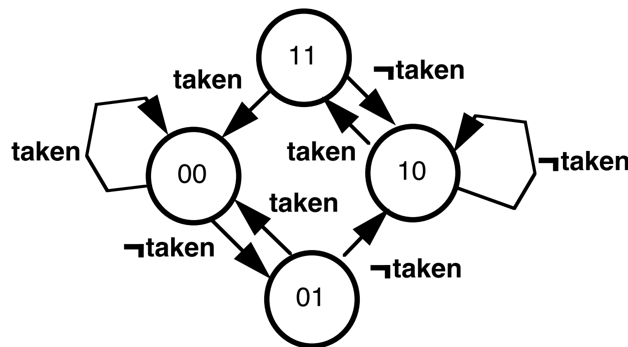


Figure 4.3-A. BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem 4.3.A **Program**

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem 4.3.B **2-bit branch prediction**

Now we will investigate how well our standard 2-bit branch predictor performs. Assume the inputs to the program are n=8 and p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc.; i.e. the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table 4.3-1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table 4.3-1 contains an entry for every branch (either b1 or b2) that is executed. The Branch Predictor (BP) bits in the table are the bits from the BHT. For each branch, check the corresponding BP bits (indicated by the bold entries in the examples) to make a prediction, then update the BP bits in the following entry (indicated by the italic entries in the examples).

Problem 4.3.C **Branch prediction with one global history bit**

Now we add a global history bit to the branch predictor, as described in lecture. Fill out Table 4.3-2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem 4.3.D **Branch prediction with two global history bits**

Now we add a second global history bit. Fill out Table 4.3-3. Again, compute the number of mispredicts you get for the same input.

Problem 4.3.E **Analysis I**

Compare your results from problems 4.3.B, 4.3.C, and 4.3.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For large n, what prediction scheme will work best? Explain briefly.

Problem 4.3.F **Analysis II**

The input we worked with in this problem is quite regular. How would you expect things to change if the input were random (each array element were equally probable 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n?

What does this tell you about when additional history bits are useful and when they hurt you?

System State		Branch Predictor		Branch Behavior	
PC	R3/R4	b1 bits	b2 bits	Predicted	Actual
b1	4/1	**10**	10	**N**	**N**
b2	4/1	*10*	**10**	**N**	**T**
b1	8/0	**10**	*11*	**N**	**T**
b2	8/0	*11*	**11**	**N**	**T**
b1	12/1	**11**	*00*		
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table 4.3-1

System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits		Predicted	Actual
			set 0	set 1	set 0	set 1		
b1	4/1	**1**	10	**10**	10	10	**N**	**N**
b2	4/1	**0**	10	*10*	**10**	10	**N**	**T**
b1	8/0	**1**	10	**10**	*11*	10		
b2	8/0							
b1	12/1							
b2	12/1							
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								

Table 4.3-2

System State			Branch Predictor								Behavior	
PC	R3/R4	history	b1 bits				b2 bits					
		bits	set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11	Predicted	Actual
b1	4/1	**11**	10	10	10	**10**	10	10	10	10	**N**	**N**
b2	4/1	**01**	10	10	10	*10*	10	**10**	10	10	**N**	**T**
b1	8/0	**10**	10	10	**10**	10	10	*11*	10	10		
b2	8/0											
b1	12/1											
b2	12/1											
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												

Table 4.3-3

Problem 4.4: Managing Out-of-order Execution

This problem investigates the operation of a superscalar processor with branch prediction, register renaming, and out-of-order execution. The processor holds all data values in a **physical register file**, and uses a **rename table** to map from architectural to physical register names. A **free list** is used to track which physical registers are available for use. A **reorder buffer (ROB)** contains the bookkeeping information for managing the out-of-order execution (but, it does not contain any register data values

When a branch instruction is encountered, the processor predicts the outcome and takes a snapshot of the rename table. If a misprediction is detected when the branch instruction later executes, the processor recovers by flushing the incorrect instructions from the ROB, rolling back the "next available" pointer, updating the free list, and restoring the earlier rename table snapshot.

We will investigate the execution of the following code sequence (assume that there is **no** branch-delay slot):

```
loop:  lw    r1, 0(r2)    # load r1 from address in r2
       addi  r2, r2, 4    # increment r2 pointer
       beqz  r1, skip     # branch to "skip" if r1 is 0
       addi  r3, r3, 1    # increment r3
skip:  bne   r2, r4, loop # loop until r2 equals r4
```

The diagram for Question 4.4.A on the next page shows the state of the processor during execution of the given code sequence. An instance of each instruction in the loop has been issued into the ROB (the beqz instruction has been predicted not-taken), but none of the instructions have begun execution. In the diagram, old values which are no longer valid are shown in the following format: ~~P1~~. The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

Problem 4.4.A

Assume that the following events occur in order (though not necessarily in a single cycle):

Step 1. The first three instructions from the next loop iteration (lw, addi, beqz) are written into the ROB (note that the bne instruction has been predicted taken).

Step 2. All instructions which are ready after Step 1 execute, write their result to the physical register file, and update the ROB. Note that this step only occurs **once**.

Step 3. As many instructions as possible commit.

Update the diagram below to reflect the processor state after these events have occured. **Cross out** any entries which are no longer valid. Note that the "**ex**" field should be **marked** when an instruction executes, and the "**use**" field should be **cleared** when it commits. Be sure to update the "**next to commit**" and "**next available**" pointers. If the **load** executes, assume that the data value it retrieves is **0**.

Rename Table

R1	~~P1~~	P4	
R2	~~P2~~	P5	
R3	~~P3~~	P6	
R4	P0		

Physical Regs

P0	8016	p
P1	6823	p
P2	8000	p
P3	7	p
P4		
P5		
P6		
P7		
P8		
P9		

Free List

~~P4~~
~~P5~~
~~P6~~
P7
P8
P9
⋮

Reorder Buffer (ROB)

next to commit → (first row)

next available → (sixth row)

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		lw	p	P2			r1	P1	P4
x		addi	p	P2			r2	P2	P5
x		beqz		P4					
x		addi	p	P3			r3	P3	P6
x		bne		P5	p	P0			

In an attempt to keep this problem set from being REALLY long the remainder of this problem has been removed. For additional practice and enlightenment feel free to look at last year's: http://inst.eecs.berkeley.edu/~cs152/sp08/assignments/ps4.pdf