Computer System Architecture
6.823 Quiz #3
November 4th, 2005
Professor Arvind
Dr. Joel Emer

Name:____________________

This is a closed book, closed notes exam.
80 Minutes
16 Pages

Notes:

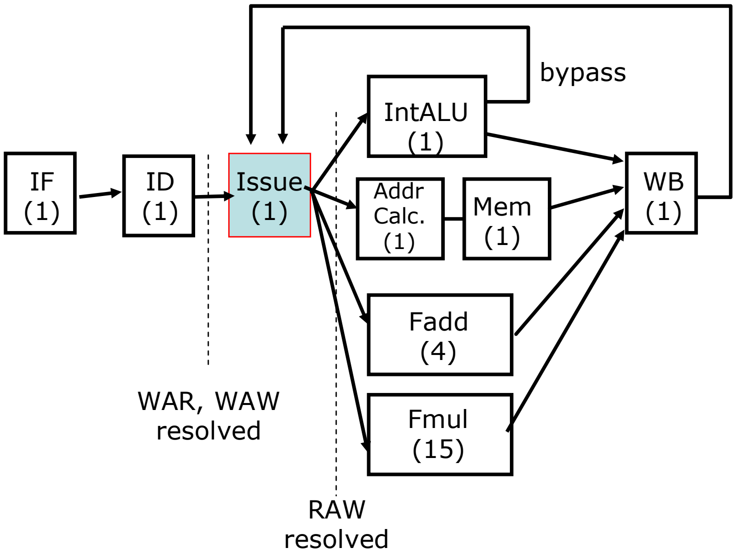
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

Writing name on each sheet	________	2	Points
Part A	________	30	Points
Part B	________	26	Points
Part C	________	22	Points
TOTAL	________	**80**	**Points**

Part A: Out-of-order Scheduling with and without Register Renaming (30 Points)

This part deals with a single-issue, out-of-order processor shown below. Each functional unit, including the LD/ST unit is fully pipelined and has the latency shown in the figure. (Note that a LD/ST takes two cycles--one cycle for address calculation and another cycle for memory access.) Assume that we have perfect branch prediction and the issue buffer can hold as many instructions as needed. The decode stage can add up to one instruction per cycle to the issue buffer, and it does so if the instruction does not have a WAR hazard with any instruction in the issue buffer or a WAW hazard with any instruction that has not written back. Assume that only one instruction can be dispatched from the issue buffer to the functional units at a time.

There are 4 floating-point registers, **`F0-F3`** and these are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (and loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypassing the next cycle after issue.



Single-issue out-of-order processor

Question 1. (6 Points)

Identify all RAW, WAW, WAR dependencies in the loop shown below. Write down the dependencies *within* a single iteration only. Use the following notation, for example, to indicate a dependency between I100 and I101 through F3 register: `I100->I101 (F3)`.

```
                         INSTRUCTION
                  LOOP:
                         // upon entry into loop,
                         // F0 = a (constant), F3 = 0, R1 = 0
        I1               L.D   F1, 0(R1)   ;load X(i)
        I2               MUL.D F2, F1, F0  ;multiply a*X(i)
        I3               ADD.D F3, F3, F2  ;add a*X(i) to F3
        I4               MUL.D F2, F1, F1  ;multiply X(i)*X(i)
        I5               S.D   F2, 0(R1)   ;store to memory
        I6               ADDI  R1, R1, 8
        I7               SGTI  R2, R1, 800
        I8               BEQZ  R2, LOOP
```

RAW:

WAR:

WAW:

Question 2. (6 Points)

The table below represents the execution of one iteration of the loop ***in steady state***. Fill in the cycle numbers for the cycles at which each instruction issues and writes back. The first row has been filled out for you already; please complete the rest of the table. Note that the order of instructions listed is not necessarily the issue order. We define cycle 0 as the time at which instruction I_1 is issued.

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	2	L.D	F1	R1	
I_2				MUL.D	F2	F1	F0
I_3				ADD.D	F3	F3	F2
I_4				MUL.D	F2	F1	F1
I_5				S.D		R1	F2
I_6				ADDI	R1	R1	
I_7				SGTI	R2	R1	
I_8				BEQZ		R3	

Question 3. (4 Points)

In steady state, how many cycles does each iteration of the loop take? _____________ cycles.

Name ______________________________

Question 4. (6 Points)

Now we modify the pipeline to support register renaming with unlimited hardware resources for renaming registers. Assume register renaming can be done instantly (taking 0 cycles) and ROB has as much capacity as needed. The table below shows instructions from our benchmark for two iterations using the same format as in Question 2.

First, fill in the new register names for each instruction, where applicable. Rename both integer and floating-point instructions.

Next, fill in the cycle numbers for the cycles at which each instruction issues and writes back. The decode stage can add up to one instruction per cycle to the reorder buffer (ROB).

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	2	L.D	T0	R1	▨
I_2	0	3		MUL.D	T1	T0	F0
I_3				ADD.D	T2	F3	T1
I_4				MUL.D			
I_5			▨	S.D	▨	R1	
I_6			▨	ADDI			▨
I_7			▨	SGTI			▨
I_8			▨	BEQZ	▨		▨
I_1				L.D			▨
I_2				MUL.D			
I_3				ADD.D			
I_4				MUL.D			
I_5			▨	S.D	▨		
I_6			▨	ADDI			▨
I_7			▨	SGTI			▨
I_8			▨	BEQZ	▨		▨

Name ____________________________

Question 5. (4 Points)

With register renaming, what is the estimated number of cycles for each iteration of the loop in steady state?

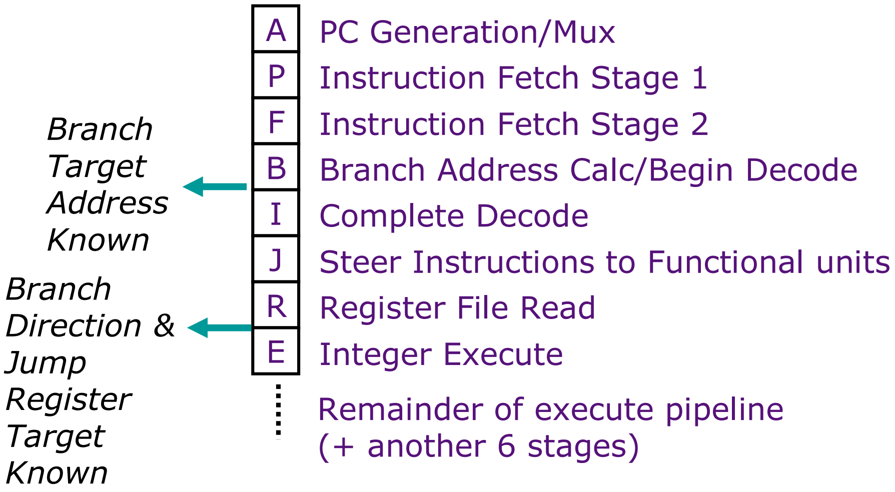
Question 6. (4 Points)

What will be the minimum size of ROB to sustain the performance in Question 5? (That is, what is the minimum number of ROB entries not to introduce additional stalls for lack of ROB space?) Assume that we maintain a freelist for ROB entries so that we can reuse an unused ROB entry at any location.

Give us a brief justification of your number, too.

Part B: Branch Prediction (26 Points)

Consider the fetch pipeline of UltraSparc-III processor. In this part, we evaluate the impact of branch prediction on the processor's performance. There are no branch delay slots.

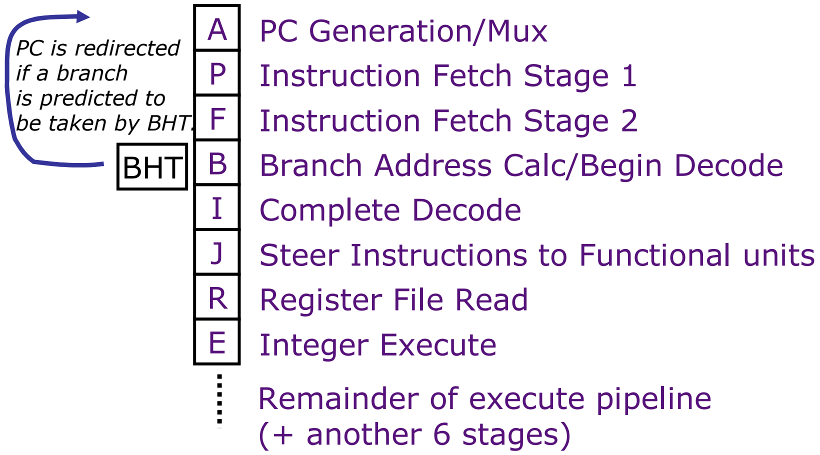

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline (+ another 6 stages)
Branch Target Address Known
Branch Direction & Jump Register Target Known

Here is a table to clarify when the direction and the target of a branch/jump is known.

Instruction	Taken known? (At the end of)	Target known? (At the end of)
BEQZ/BNEZ	R	B
J	B (always taken)	B
JR	B (always taken)	R

Question 7. (6 Points)

As a first step, we add a branch history table (BHT) in the fetch pipeline as shown below. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) looks up the BHT, but an unconditional jump does not. If a branch is predicted to be taken, some of the instructions are flushed and the PC is redirected to the calculated branch target address. The instruction at PC+4 is fetched by default unless PC is redirected by an older instruction.


PC is redirected if a branch is predicted to be taken by BHT
BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline (+ another 6 stages)

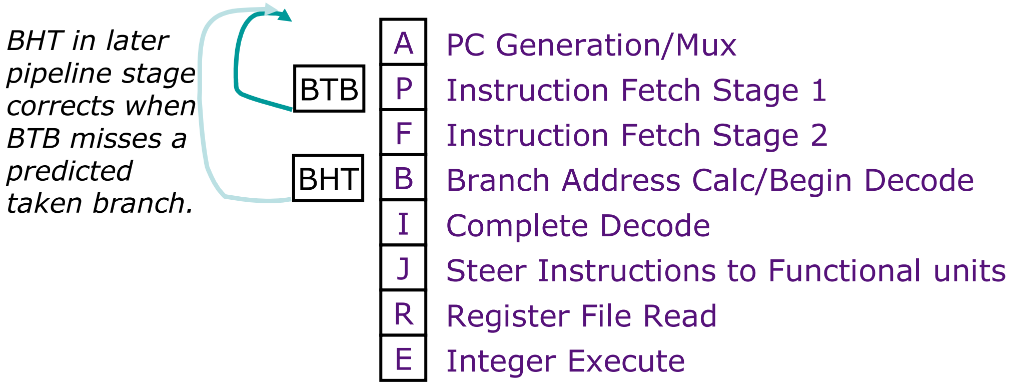
For each of the following cases, write down the number of pipeline bubbles caused by a branch or jump. If there is no bubble, you can simply put 0. (Y = yes, N= no)

	Predicted Taken?	**Actually Taken?**	**Pipeline bubbles**
BEQZ/ BNEZ	Y	Y	
	Y	N	
	N	Y	
	N	N	
J	Always taken (No lookup)	Y	
JR	Always taken (No lookup)	Y	

Question 8. (8 Points)

To improve the branch performance further, we decide to add a branch target buffer (BTB) as well. Here is a description for the operation of the BTB.

1. The BTB holds entry_PC, target_PC pairs for jumps and branches predicted to be taken. Assume that the target_PC predicted by the BTB is always correct for this question. (Yet the direction still might be wrong.)
2. The BTB is looked up every cycle. If there is a match with the current PC, PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.



BHT in later pipeline stage corrects when BTB misses a predicted taken branch.

BTB

BHT

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Fill out the following table of the number of pipeline bubbles (only for conditional branches).

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	
	Y	Y	N	
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	
	N	Y	N	
	N	N	Y	
	N	N	N	

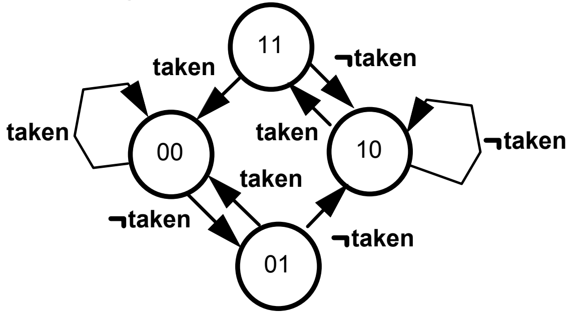
Question 9. (6 Points)

We will be working on the following program:

```
ADDRESS                INSTRUCTION
 0x1000          BR1:  BEQZ R5, NEXT      ; always taken
 0x1004                ADDI R4, R4, #4
 0x1008                MULT R3, R5, R3
 0x100C                ST   R3, 0(R4)
 0x1010                SUBI R5, R5, #1
 0x1014         NEXT:  ADDI R1, R1, #1
 0x1018                SLTI R2, R1, 100   ; repeat 100 times
 0x101C          BR2:  BNEZ R2, BR1
 0x1020                NOP
 0x1024                NOP
 0x1028                NOP
```

Given a snapshot of the BTB and the BHT states on entry to the loop, fill in the timing diagram of one iteration (plus two instructions) on the next page. (Don't worry about the stages beyond the E stage.) We assume the following for this question:

1. The initial values of R5 and R1 are zero, so BR1 is always taken.
2. We disregard any possible structural hazards. There are no pipeline bubbles (except for those created by branches.)
3. We fetch only one instruction per cycle.
4. We use a two-bit predictor whose state diagram is shown below. In state 1X we will guess not taken; in state 0X we will guess taken. BR1 and BR2 do not conflict in the BHT.



5. We use a two-entry fully-associative BTB with the LRU replacement policy.

Initial Snapshot

(Valid) V	Entry PC	Predicted Target PC
1	0x101C	0x1000

BTB

	...	
BR1	1	1
	...	
BR2	0	0
	...	

BHT

TIME →

Address	Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0x1000	BEQZ R5, NEXT	A	P	F	B	I	J	R	E															
0x1014	ADDI R1, R1, #1																							
0x1018	SLTI R2, R1, 100																							
0x101C	BNEZ R2, LOOP																							
0x1000	BEQZ R5, NEXT																							
0x1014	ADDI R1, R1, #1																							

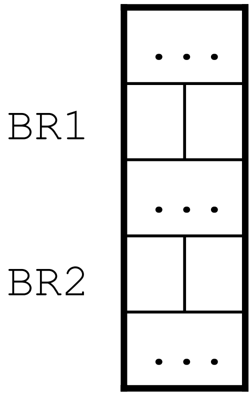
Timing diagram for Question 9

Question 10. (6 Points)

What will be the BTB and BHT states right after the 6 instructions in Question 9 have updated the branch predictors' states? Fill in (1) the BTB and (2) the entries corresponding to BR1 and BR2 in the BHT.

(Valid) V	Entry PC	Predicted Target PC

BTB



BHT

Name ____________________________

Part C: Out-of-order Execution with Unified Physical Registers (22 Points)

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). These components operate as described in Lecture 14 (2005).

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 11 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.

Name ____________________________

Question 11. (12 Points)

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

Step 1. The first instruction from the next loop iteration (lw) is written into the ROB (note that the bne instruction has been predicted taken) → Label whatever changes with □.

Step 2. The second instruction from the next loop iteration (addi) is written into the ROB. → Label whatever changes with □.

Step 3. The third instruction from the next loop iteration (beqz) is written into the ROB. → Label whatever changes with □.

Step 4. 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**. → Label with □.

Step 5. As many instructions as possible commit. → Label whatever changes with □

(We ask you to label the changes with circled numbers to give partial credits.)

Update the diagram below to reflect the processor state after these events have occurred. (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)

	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit →	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			
next available →										

Name ____________________________

Question 12. (10 Points)

Assume that the same initial ROB states as in Question 11 (reproduced in the figure below). The following events occur in order:

Step 1. The processor executes the top two instructions in the ROB (lw, addi), writes back their results (to the physical register file and ROB), and commits.
Step 2. The processor executes the beqz instruction and detects that it has mispredicted the branch outcome, and recovery action is taken to repair the processor state.
Step 3. The beqz instruction commits.
Step 4. The correct next instruction is fetched and is written into the ROB.

Fill in the diagram on the next page to reflect the processor state after these events have occurred. Although you are not given the rename table snapshot at the beqz instruction, you should be able to deduce the necessary information. You do not need to show invalid entries in the diagram, but be sure to fill in all fields which have valid data, and update the "next to commit" and "next available" pointers. Also make sure that the free list contains all available registers.

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)

	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit →	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			
next available →										

Name ______________________________

Rename Table

R1			
R2			
R3			
R4			

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		

Free List

⋮

Reorder Buffer (ROB)

next to commit

next available

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

The final states after the three steps in Question 12 (You should fill in.)