Name _____

Email _____@mit.edu

Computer System Architecture
6.823 Quiz #2
April 3th, 2015
Professors Daniel Sanchez and Joel Emer

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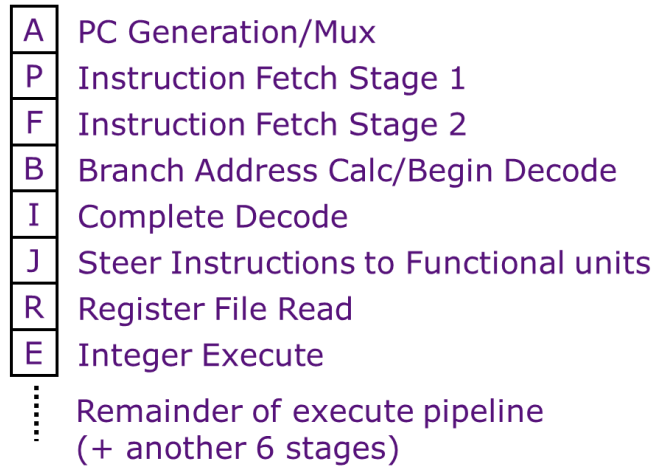
80 Minutes
17 Pages

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.
- Show your work to receive full credit.
- 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.

Part A	_____	26 Points
Part B	_____	32 Points
Part C	_____	30 Points
Part D	_____	12 Points

TOTAL	**_____**	**100 Points**

Part A: Branch Prediction (26 points)

Ben Bitdiddle is designing a processor with the complex pipeline illustrated below:

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)

The processor has the following characteristics:

- Issues at most one instruction per cycle.
- Branch addresses are known at the end of the B stage (Branch Address Calc/Begin Decode).
- Branch conditions (taken/not taken) are known at the end of the R stage (Register File Read).
- Branches always go through the pipeline without any stalls or queuing delays.

Ben's target program is shown below:

```
for(int i = 0; i <= 1000000; i++)
{
    if(i % 2 == 0) //Branch B1
    { //Not taken
        (Do something A)
    }
    if(i % 4 == 0) //Branch B2
    { //Not taken
        (Do something B)
    }
}    //Branch LP
```

```
        ANDi R1 0
LOOP:MODi R2 R1 2
        BNE  R2 M4      // B1
        (Do something A)
        ... ...
M4:   MODi R3 R1 4
        BNE  R3 END     // B2
        (Do something B)
        ... ...
END: SUBi R4 R1 1000000
        BNE  R4 LOOP    // LP
        ... ...
```

The MODi (modulo-immediate) instruction is defined as follows:
MODi Rd Rs imm: Rd <- Rs Mod imm

Question 1 (3 points)

In steady state, what is the probability for each branch in the code to be taken/not taken on average? Fill in the table below.

Branch	Probability to be **taken**	Probability to be **not taken**
B1		
B2		
LP		

Question 2 (3 points)

In steady state, how many cycles per iteration are lost on average if the processor always speculates that every branch is not taken (i.e., next PC is PC+4)?
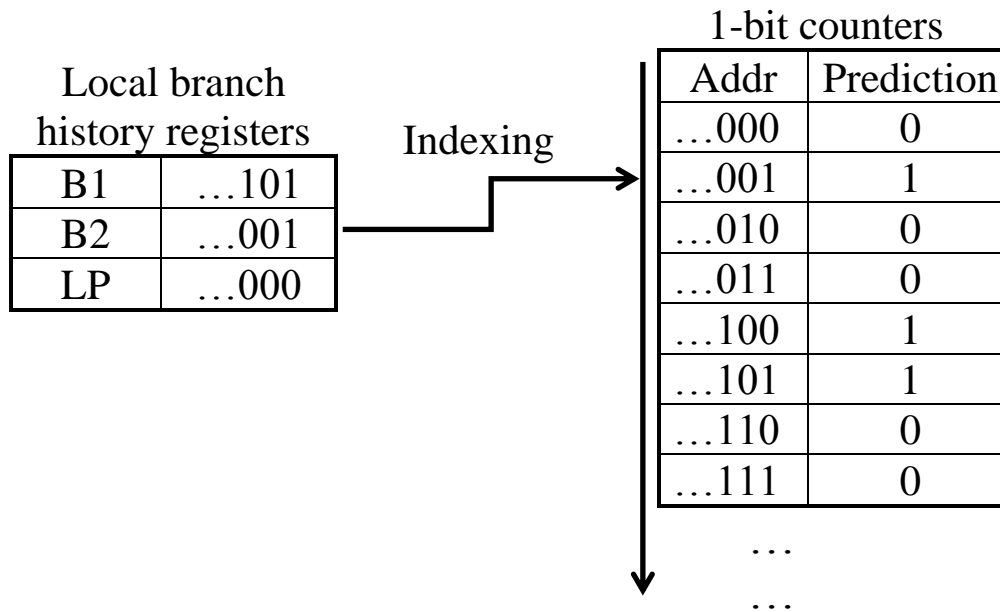
Question 3 (5 points)

Ben designs a **<u>static branch predictor</u>** to improve performance. This predictor always predicts **<u>not taken for forward jumps</u>** and **<u>taken for backward jumps</u>**. The prediction is available at the end of the **<u>B</u>** stage. In steady state, how many cycles per iteration are lost on average?

Question 4 (7 points)

To improve performance further, Ben designs a **dynamic branch predictor with local branch history registers and 1-bit counters**.

Each local branch history registers store the last several outcomes of a single branch (branches B1, B2 and LP in our case). By convention, the most recent branch outcome is the least significant bit, and so on. The predictor uses the local history of the branch to index a table of 1-bit counters. It predicts not taken if the corresponding 1-bit counter is 0, and taken if it is 1. Assume local branch history registers are always correct.

1-bit counters

Local branch history registers		Indexing	Addr	Prediction
B1	…101		…000	0
B2	…001		…001	1
LP	…000		…010	0
			…011	0
			…100	1
			…101	1
			…110	0
			…111	0

. . .

. . .

How many bits per branch history register do we need to perform perfect prediction in steady state?

Question 5-1 (4 points)

The local-history predictor itself is a speculative structure. That is, for subsequent predictions to be accurate, the predictor has to be updated speculatively.

Explain what guess the local history update function should use.

Question 5-2 (5 points)

Ben wants to design the data management policy (i.e., how to manage the speculative data in different structures of the predictor) for the local-history branch predictor to work well. Use a couple of sentences to answer the following questions.

1) What data management policies should be applied to each structure?

2) For your selected data management policies, is there any challenge for the recovery mechanism when there is misspeculation? If so, what are the challenges?

Part B: Speculative Execution and Recovery (32 points)

You are given an out-of-order processor that

- Issues at most one instruction per cycle
- Commits at most one instruction per cycle
- Uses an unified physical register file

Question 1 (6 points)

Consider the following code sequence:

```
        Addr
I0  (0x24)        lw    r2, (r4), #0
I1  (0x28)        addi  r2, r2,   #16
I2  (0x2C)        lw    r3, (r4), #4
I3  (0x30)        blez  r3, L1
I4  (0x34)        addi  r4, r2, #8
I5  (0x38)        mul   r1, r2, r3
I6  (0x3C)        addi  r3, r2, #8
I7  (0x40)  L1:   add   r2, r1, r3
```

Assume the branch instruction (blez) is not taken. Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence.

Older Instruction

Younger Instruction	I0	I1	I2	I3	I4	I5	I6	I7
I0	-							
I1	WAW RAW	-						
I2			-					
I3				-				
I4					-			
I5						-		
I6							-	
I7								-

In Questions 2 through 4, you should update the state of the processor when different events happen. The starting state in each question is the same, and the event specified in each question is the ONLY event that takes place for that question. The starting state is shown in the different structures: renaming table, physical registers, free list, two-bit branch predictor, global history buffer, and reorder buffer (ROB).

Note the following conventions:

- The valid bit for any entry is represented by "1".
- The valid bit can be cleared by crossing it out.
- In the ROB, the "ex" field should be marked with "1" when an instruction starts execution, and the "use" field should be cleared when it commits. Be sure to update the "next to commit" and "next available" pointers, if necessary.
- Fill out the "after" fields in all the tables. Write new values in these boxes if the values change due to the event specified in the question. You do not have to repeat the values if they do not change due to the event.

In Questions 2 through 4, we will use the same code sequence as in Question 1:

```
        Addr
I0  (0x24)          lw     r2, (r4), #0
I1  (0x28)          addi   r2, r2,   #16
I2  (0x2C)          lw     r3, (r4), #4
I3  (0x30)          blez   r3, L1
I4  (0x34)          addi   r4, r2, #8
I5  (0x38)          mul    r1, r2, r3
I6  (0x3C)          addi   r3, r2, #8
I7  (0x40)  L1: add    r2, r1, r3
```

The starting state of the processor is as follows:

- Instructions I0-I4 are already in the ROB.
- I0 (lw) has already finished execution.
- I1 (addi) and I2 (lw) have started executing but have not finished yet.
- I3 (blez) has been predicted to be Not-Taken by the branch predictor.
- I5 (mul) has completed the decode stage.
- I6 (addi) has completed the Fetch Stage.
- The next PC is set to 0x40, which is the PC of I7 (add).

Question 2 (6 points)

The following figure shows the starting state of the processor. Suppose the decoded instruction I5 (mul) is now inserted into the ROB. Update the diagram to reflect the processor state after this event has occurred.

Prediction Counter		
Index	Before	After
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Fetched Inst. Queue	
PC	Inst.
0x3C	I6 (addi)

Decoded Inst. Queue
Inst.
I5 (mul)

Branch Global History	
Before	After
0010110	

Next PC to be fetched	
Before	After
0x40	

Rename Table (Latest)		
Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Physical Registers		
Name	Value	Valid
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Free List
P8
P9
P10

Rename Table (Snapshot 1)	Valid
	1
Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	P3	

Reorder Buffer (ROB)									
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
1	1	lw	1	P3			r2	P1	P4
1	1	addi	1	P4			r2	P4	P5
1	1	lw	1	P3			r3	P2	P6
1		blez		P6					
1		addi		P5			r4	P3	P7

Next to commit

Next available

Question 3 (8 points)

Start from the same processor state, shown below. Suppose now I1 (addi) has completed execution. Commit as many instructions as possible. Update the diagram to reflect the processor state after I1 execution completes and as many instructions as possible have committed. Again, assume no other events take place.

Prediction Counter		
Index	Before	After
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Fetched Inst. Queue	
PC	Inst.
0x3C	I6 (addi)

Decoded Inst. Queue
Inst.
I5 (mul)

Branch Global History	
Before	After
0010110	

Next PC to be fetched	
Before	After
0x40	

Rename Table (Latest)		
Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Physical Registers		
Name	Value	Valid
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Free List
P8
P9
P10

Rename Table (Snapshot 1)		Valid
		1
Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	P3	

Reorder Buffer (ROB)									
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
1	1	lw	1	P3			r2	P1	P4
1	1	addi	1	P4			r2	P4	P5
1	1	lw	1	P3			r3	P2	P6
1		blez		P6					
1		addi		P5			r4	P3	P7

Next to commit ← (points to first ROB row)

Next available ← (points to sixth ROB row)

Question 4 (12 points)

Start from the same processor state, shown below. Suppose instruction I2 (lw) triggers an ALU overflow exception. Restore the architectural and microarchitectural state to recover from misspeculation. The exception handler for the processor is at address 0x8C (control is transferred to the exception handler after recovery). You do not need to worry about the number of cycles taken by recovery. Show the processor state after recovery.

Prediction Counter

Index	Before	After
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Rename Table (Latest)

Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Rename Table (Snapshot 1)

Name	Before	After	Valid
			1
R1	P0		
R2	P5		
R3	P6		
R4	P3		

Fetched Inst. Queue

PC	Inst.
0x3C	I6 (addi)

Decoded Inst. Queue

Inst.
I5 (mul)

Branch Global History

Before	After
0010110	

Next PC to be fetched

Before	After
0x40	

Physical Registers

Name	Value	Valid
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Free List

P8
P9
P10

Reorder Buffer (ROB)

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd	
1	1	lw	1	P3			r2	P1	P4	← Next to commit
1	1	addi	1	P4			r2	P4	P5	
1	1	lw	1	P3			r3	P2	P6	
1		blez		P6						
1		addi		P5			r4	P3	P7	← Next available

Part C: Out-of-order Execution (30 points)

You are given an out-of-order processor with unlimited decode, issue, commit bandwidth. The processor's ISA has 16 architectural registers. To achieve an efficient design, you are asked to calculate the average occupancy of various structures for different implementation alternatives. We will use the following code:

```
while(true) {
    i = i + 1
    A[i] = A[i]+1
}
```

The above pseudo code can be unrolled (thus eliminating branches) and translated into the following instruction sequence, with four instructions per iteration:

```
I0    addi   r1, r1,   #1
I1    lw     r2, (r1), #0
I2    addi   r2, r2,   #1
I3    sw     r2, (r1), #0
I4    addi   r1, r1,   #1
I5    lw     r2, (r1), #0
I6    addi   r2, r2,   #1
I7    sw     r2, (r1), #0
            ⋮
```

Below are two different diagrams that show the cycles at which instructions are decoded, issued, and committed in steady state (use the one you find more convenient). First, the following table shows these cycles for the instructions in the Nth loop iteration:

Instruction Number	Opcode	Decode	Issue	Commit
4N	addi	N	N+1	N+5
4N+1	lw	N	N+2	N+5
4N+2	addi	N	N+4	N+5
4N+3	sw	N	N+5	N+6

For example, instruction I8 (addi) is decoded at cycle 2, issued at cycle 3, and committed at cycle 7. Second, the waterfall diagram on the next page also describes how instructions are scheduled in steady state:

Time:	N	N+1	N+2	N+3	N+4	N+5	N+6	N+7
I4N (addi)	D	I				C		
I4N+1 (lw)	D		I			C		
I4N+2 (addi)	D				I	C		
I4N+3 (sw)	D					I	C	
I4N+4 (addi)		D	I				C	
I4N+5 (lw)		D		I			C	
I4N+6 (addi)		D				I	C	
I4N+7 (sw)		D					I	C

Hint: To answer these questions, you do not need to derive the instruction scheduling for more iterations.

Question 1 (6 points)

Assume store instructions spend 5 cycles on average in the store buffer. In steady state, how many store buffer entries are in use on average?

Question 2 (6 points)

Assume we have a reorder buffer (ROB) that holds data values as described in lecture. It works as follows:
- At decode stage: an instruction is decoded and written to the ROB. The instruction grabs an ROB entry at the beginning of the cycle.
- At issue stage: the instruction enters the execution pipeline.
- At commit stage: the instruction leaves the ROB at the end of the cycle.

In steady state, how many ROB entries are in use on average?

Question 3 (6 points)

Assume we have the same ROB as in Question 2. Suppose all load instructions miss in the cache. As a result, the issue stage for the addi and sw instructions after each lw instruction is delayed by 100 cycles, and the commit stage for every instruction is also delayed by 100 cycles.

In steady state, how many ROB entries are in use on average?

Question 4 (6 points)

Assume every load hits in the cache again. Instead of storing data in the ROB, we use a unified physical register file to hold all speculative and non-speculative copies of the 16 architectural registers. If an instruction needs a new physical register, it grabs an entry in the physical register file at the beginning of the decode stage and releases the previously mapped physical register at the end of the commit stage.

In steady state, how many physical registers are in use on average?

Question 5 (6 points)

A lot of logic in the ROB is dedicated to decide when an instruction is ready to issue. To simplify the ROB implementation, we decide to have a separate, smaller issue queue to handle instructions waiting to be issued. This way, when an instruction is issued, it does not continue to occupy an "expensive" slot with issue logic:

- At decode stage: an instruction is decoded. The instruction grabs an ROB entry as well as an entry in the issue queue at the beginning of the cycle.
- At issue stage: the instruction leaves the issue queue at the end of the cycle.
- At commit stage: the instruction leaves the ROB at the end of the cycle.

Assume every load hits in the cache. In steady state, how many issue queue entries are in use on average?

Part D: Multithreading (12 points)

Consider the following instruction sequence.

```
        addi   r3, r0, 256
  loop: lw     f1, r1, #0
        lw     f2, r2, #0
        mul    f3, f1, f2
        sw     f3, r2, #0
        addi   r1, r1, #4
        addi   r2, r2, #4
        addi   r3, r3, #-1
        bnez   r3, loop
```

Assume that memory operations take 4 cycles (i.e., if instruction I1 starts execution at cycle N, then instructions that depend on the result of I1 can only start execution at or after cycle N+4); multiply instructions take 6 cycles; and all other operations take 1 cycle. Assume the multiplier and memory are pipelined (i.e., they can start a new request every cycle). Also assume perfect branch prediction.

Question 1 (3 points)

Suppose the processor performs fine-grained multithreading with fixed round-robin switching: the processor switches to the next thread every cycle, and if the instruction of the next thread is not ready, it inserts a bubble into the pipeline. What is the minimum number of threads required to fully utilize the processor every cycle while running this code?

Question 2 (9 points)

Suppose the processor performs coarse-grained multithreading, i.e. the processor only switches to another thread when there is a L2 cache miss. Will the following three metrics increase or decrease, compared to fixed round-robin switching? Use a couple of sentences to answer the following questions.

1) Compared to fixed round-robin switching, will the **number of threads needed for the highest achievable utilization** increase or decrease? Why?

2) Compared to fixed round-robin switching, will the **highest achievable pipeline utilization** increase or decrease? Why?

3) Compared to fixed round-robin switching, will **cache hit rate** increase or decrease? Why?