Name _____

Computer System Architecture
6.823 Quiz #2
April 4th, 2014
Professors Daniel Sanchez and Joel Emer

This is a closed book, closed notes exam.

80 Minutes
15 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	_____	30 Points
Part B	_____	25 Points
Part C	_____	30 Points
Part D	_____	15 Points
TOTAL	**_____**	**100 Points**

Name _____

Part A: Virtual Memory (30 pts)

Ben Bitdiddle decides to build a cache with the following properties:

- 16 blocks with 64-bytes per block.
- 2-way set associative organization.
- Virtually indexed, physically tagged.
- LRU replacement.

His memory system looks like the following:

- 16-bit virtual and physical addresses.
- 1024-byte pages.
- A single-level page table stored in physical memory. The page table base register (equal to `0x1000`) holds the address of the start of the page table.
- Page table entries are 16 bits, with the highest order bits indicating the physical page number, and the rest as status bits.
- 8-entry fully associative TLB.

While running his architecture, he is curious about the performance of caches, so he asks for some help running through some operations. Fill out the final state of the cache and the TLB after the following virtual memory address accesses:

```
0x05DB: 0000 0101 1101 1011
0x0B49: 0000 1011 0100 1001
0x17FB: 0001 0111 1111 1011
0x1C5E: 0001 1100 0101 1110
~~0x35E3: 0011 0101 1110 0011~~
```

The contents of physical memory follow after the cache diagrams on the next page.

Name _____

Cache:

Index	Way 0		Way 1	
	Valid?	Tag	Valid?	Tag
0	0		0	
1	1	1010110	0	
2	0		0	
3	0		0	
4	0		0	
5	1	1101101	0	
6	0		0	
7	1	1011100	1	0111011

TLB:

VPN	PPN
000001	101110
000010	110110
000101	011101
000111	101011

From the problem description, addresses are broken up as follows:

- Virtual memory: 10-bit page offset, 6-bit page number (virtual and physical).
- Cache: 6-bit block offset, 3-bit tag, 7-bit tag.

The tag is larger than the page number so we don't need to worry about using more bits of the address than is conventional.

We proceed by looking up the physical page number in the page table from the virtual page number. For example, the virtual page number of the first address (0x05DB) is the top six bits, or 000001. This tells us to take the first (counting from zero) page table entry. Since page table entries are 16 bits and the page table starts at 0x1000, this is memory address 0x1002. From the physical memory given on the next page, the page table entry is 0xBABE. The physical page number is the top six bits (from problem description), which in this case is 101110. We can now enter this into the TLB as the first entry (the TLB is fully associative).

Now, onto the cache. The PPN gives us the first six bits of the tag. The tag is seven bits, so we take the top bit of the page offset to complete the tag, which is 0 in this case, so the tag is 1010110. The index for this address is the next three bits of the page offset, or 111. This tells us to insert this tag into the last set in our cache, which we do and mark the set valid.

This process repeats for the remaining addresses mechanically, with the third address (0x17FB) mapping to the same set as the first and thus using the second way.

Name _____

Name _____

Memory contents:

Page Table Base Register
0x1000

Physical Memory

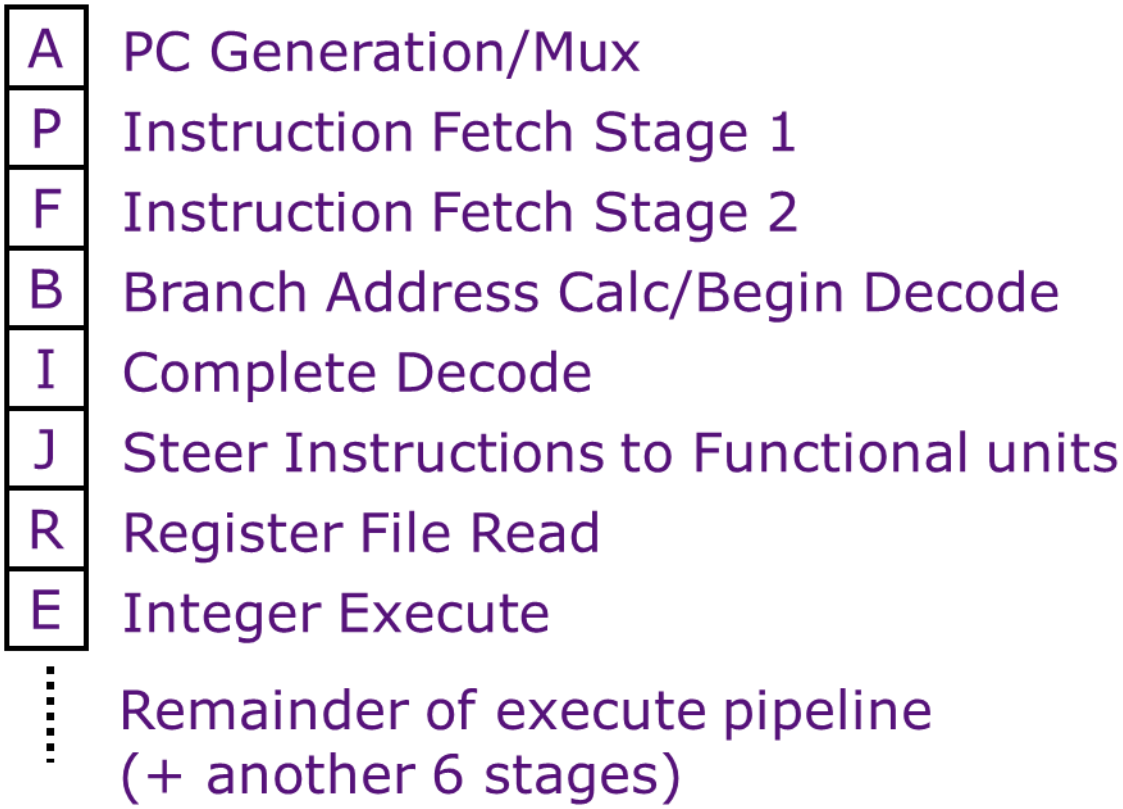
Address		Value
0x11FE	:	0x7777
0x11FC	:	0xBABA
0x11FA	:	0xAB00
0x11F8	:	0x1BD1
0x11F6	:	0x9001
0x11F4	:	0xAAAA
0x11F2	:	0xB789
0x11F0	:	0xDEF0

…

0x1022	:	0xBADE
0x1020	:	0xBEAD
0x101E	:	0xDEAF
0x101C	:	0xFACE
0x101A	:	0x9ABC
0x1018	:	0x5678
0x1016	:	0x1234
0x1014	:	0xFEED
0x1012	:	0xD666
0x1010	:	0xDEAD
0x100E	:	0xADD0
0x100C	:	0x00F0
0x100A	:	0x7734
0x1008	:	0x3704
0x1006	:	0x1337
0x1004	:	0xDAB0
0x1002	:	0xBABE
0x1000	:	0xACE0

Name _____

Part B: Complex Pipelining (25 points)



You are designing a processor with the complex pipeline illustrated above. For this problem assume there are no unconditional jumps or jump register—*only* conditional branches.

Suppose the following:

- Each stage takes a single cycle.
- Branch addresses are known after stage Branch Address Calc/Begin Decode.
- Branch conditions (taken/not taken) are known after Register File Read.
- Initially, the processor *always* speculates that the next instruction is at PC+4, without any specialized branch prediction hardware.
- Branches always go through the pipeline without any stalls or queuing delays.

Name _____

Question 1 (5 points):

How much work is lost (in cycles) on a branch misprediction in this pipeline?

6 cycles are lost when stalls are inserted into pipeline stages A, P, F, B, I and J.

Question 2 (5 points):

If one quarter of instructions are branches, and half of these are taken, then how much should we expect branches to increase the processor's CPI (cycles per instruction)?

This answer is asking how much CPI is spent on branches in the machine, increase relative to a machine that never stalls on branches (e.g. has "magic fetch").

Branch CPI = misprediction rate x misprediction penalty

From the problem description, we always predict PC+4 or "not taken". So the misprediction rate is just the rate of taken branches.

Branch CPI = fraction branches x fraction taken x misprediction penalty

From the question:

Branch CPI = ¼ x ½ x 6 = ¾

Question 3 (5 points):

You are unsatisfied with this performance and want to reduce the work lost on branches. Given your hardware budget, you can add only one of the following:

- A branch predictor to your pipeline that resolves after Instruction Fetch Stage 1.
- Or a branch target buffer (BTB) that resolves after Instruction Fetch Stage 2.

If each make the same predictions, which do you prefer? In one or two sentences, why?

Branch predictions earlier than B are unhelpful since we don't have an address to jump to even if the branch is predicted taken. So although the BTB is available later in the pipeline, it is better to have the BTB since it gives us an address we can use to redirect fetch.

Name _____

Question 4 (10 points):

You decide to add the BTB (not the branch predictor). Your BTB is a fully tagged structure, so if it predicts an address other than PC+4 then it always predicts the branch address of a conditional branch (but not the condition!) correctly. **For partial credit, show your work.**

If the BTB correctly predicts a next PC other than PC+4, what is the effect on the pipeline?

We inject two stalls into stages A and P and redirect fetch to the BTB address. So we lose 2 cycles.

If the BTB predicts the next PC incorrectly, what is the effect on the pipeline?

The BTB has exactly the same misprediction penalty as the baseline machine—6 cycles. This is true regardless of whether the BTB predicted PC+4 or a different address, since no matter what after an incorrect prediction the branch will be followed by six stalls in the pipeline. (The penalties are not additive.)

Assume the BTB predicts PC+4 90% of the time. When the BTB predicts PC+4 it is accurate 90% of the time. Otherwise it is accurate 80% of the time. How much should we expect branches to increase the CPI of the BTB design? *(Don't bother trying to compute exact decimal values.)*

This is simply a matter of computing the probabilities of all combinations of prediction and accuracy and their associated penalties.

Denote each case as "prediction/actual". So "T/NT" means the BTB predicted a PC other than PC+4, but it turned out that PC+4 was the actual branch resolution.

Branch CPI = T/T CPI + T/NT CPI + NT/T CPI + NT/NT CPI

NT/NT CPI is zero since this just means the BTB predicted PC+4 and no stalls happened.

T/T CPI incurs a penalty of 2 cycles (see above), and this happens when the BTB predicts a PC other than PC+4 (10%) *and* it is correct (80%). So T/T CPI = 2 * 0.1 * 0.8

T/NT CPI and NT/T CPI both incur a penalty of 6 cycles (see above). These occur when the BTB is incorrect about its prediction: T/NT rate is 0.1 * 0.2, NT/T rate is 0.9 * 0.1. So T/NT CPI = 0.1 * 0.2 * 6 and NT/T CPI = 0.9 * 0.1 * 6.

Branch CPI = (0.1 * 0.2 + 0.9 * 0.1) * 6 + 0.1 * 0.8 * 2 = 0.82

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

In this problem, we are going to update the state of the processor when different events happen. You are given an out-of-order processor in some initial state, as described by the registers (renaming table, physical registers, and free list), one-bit branch predictor, and re-order buffer. Your job is to show the changes that occur when some event occurs, starting from the same initial state except where noted. For partial credit, briefly describe what changes occur.

Name _____

Question 1 (10 points):

Show the state of the processor if the first load completes (but does not commit).

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

RENAMING TABLE	
R1	P4
R2	P6
R3	P5
R4	P3

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		p
P5		
P6		
P7		

FREE LIST
P7

RE-ORDER BUFFER (ROB)

	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
Next to commit →	X	X	LD	p	P1			R1	P0	P4
	X		ADD	p	P4	p	P3	R3	P2	P5
	X		ADD	p	P4	p	P1	R2	P1	P6
Next available →	X		BGEZ	p	P3					

Name _____

Question 2 (10 points):

Show the state of the processor after the next instruction is issued.

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

RENAMING TABLE	
R1	P4
R2	P6
R3	~~P5~~ P7
R4	P3

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		
P5		
P6		
P7		

FREE LIST
~~P7~~

RE-ORDER BUFFER (ROB)

	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
Next to commit →	X		LD	p	P1			R1	P0	P4
	X		ADD		P4	p	P3	R3	P2	P5
	X		ADD		P4	p	P1	R2	P1	P6
	X		BGEZ	p	P3					
	X		LD		P6			R3	P5	P7
Next available →										

Name _____

Question 3 (5 points):

From the state at the end of Question 2, as the next action can the processor issue (not execute) another instruction?

No. There are no physical registers on the free list.

In one or two sentences, what does this say about our design? How can we improve it?

We didn't solve Little's Law correctly when we sized our physical register file. We need to make it bigger so it can support the number of instructions we have in flight in the ROB.

Name _____

Question 4 (5 points):

Show the state of the processor if the first `LD` triggers a page fault and after abort finishes.

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

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

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		
P5		
P6		
P7		

FREE LIST
P7
P4
P5
P6

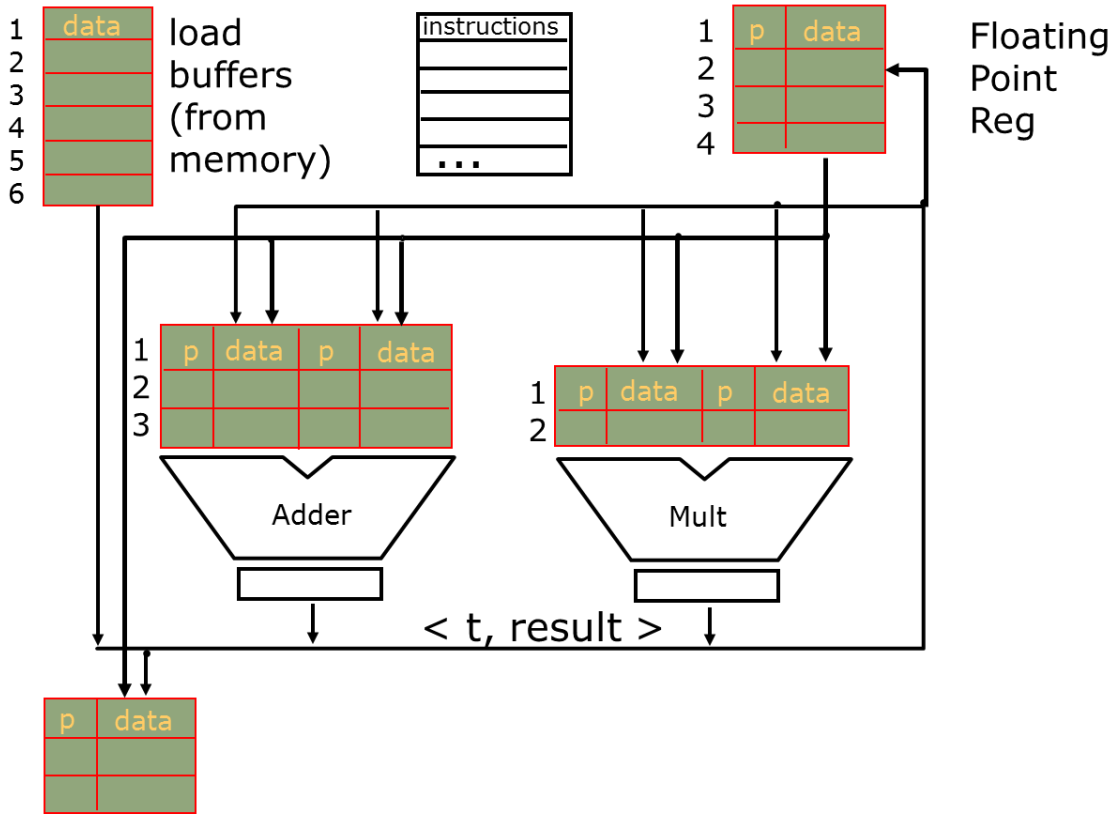
Next available →

Next to commit →

RE-ORDER BUFFER (ROB)									
Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
~~X~~		~~LD~~	~~p~~	~~P1~~			~~R1~~	~~P0~~	~~P4~~
~~X~~		~~ADD~~		~~P4~~	~~p~~	~~P3~~	~~R3~~	~~P2~~	~~P5~~
~~X~~		~~ADD~~		~~P4~~	~~p~~	~~P1~~	~~R2~~	~~P1~~	~~P6~~
~~X~~		~~BGEZ~~	~~p~~	~~P3~~					

Name _____

Part D: Out-of-order Processor Design (15 points)



You are designing an out-of-order processor similar to the IBM 360/91 Tomasulo design shown above. This design distributes the re-order buffer around the processor, placing entries near their associated functional units. In such a design, the distributed ROB entries are called "reservation stations". Entries are allocated when the instruction is decoded and freed when the instruction is dispatched to the functional unit.

Your design achieves an average throughput of 1.5 instructions per cycle. Two-thirds of instructions are adds, and one-third are multiplies. The latency of each instruction type *from allocation to completion* is 5 cycles for adds and 14 cycles for multiplies.

Type of operation	Fraction of instructions	Average latency
Add	2/3	5
Multiply	1/3	14

The adder and multiplier are each fully pipelined with full bypassing. *Once an instruction is dispatched to the FU*, the adder takes 2 cycles and the multiplier takes 5 cycles.

Throughput	Add latency	Multiply latency
1.5	2	5

Name _____

Questions:

How many entries are in use, on average, in the reservation station at each functional unit (adder, multiplier) in the steady state? Assume there are infinite entries available if needed. What is the average latency of an instruction in this machine? *For partial credit, feel free to give any formulae you believe may be important to answer this question.*

This is a Little's Law question: T = N / L.

From the fraction of instructions and the machine's total throughput, we can get the throughput of each type of instruction.

Tadd = 2/3 * 3/2 = 1
Tmul = 1/3 * 3/2 = 1/2

To solve for the number of entries in use, we need to know the average latency an instruction spends in the reservation station. From the problem description, reservation stations are in use from allocation until the instruction is dispatched to the functional unit. So the latency in the reservation station itself is the end-to-end latency minus the latency of the functional unit.

Lr,add = Ladd – Lfu,add = 5 – 2 = 3 cycles
Lr,mul = Lmul – Lfu,mul = 14 – 5 = 9 cycles

Thus the number of entries in use is on average:

Nadd = Tadd * Lr,add = 3
Nmul = Tmul * Lr,mul = 9 / 2 = 4.5

The average latency can be computed from the frequency of instructions directly:

L = 2/3 Ladd + 1/3 Lmul = 2/3 * 5 + 1/3 * 14 = 8

Or from Little's Law, but this is more complicated. We now want to know the number of adds and multiplies in flight. This is the number of entries plus the number of instructions in the FU themselves. The adder has an issue rate of 1, so the adder is always full. The multiplier has an issue rate of ½, so it is half full. Therefore:

L = N / T = (3 + 2 + 4.5 + 5/2) / 1.5 = 8

It's nice to see that they agree, but really the first formulation is much easier.