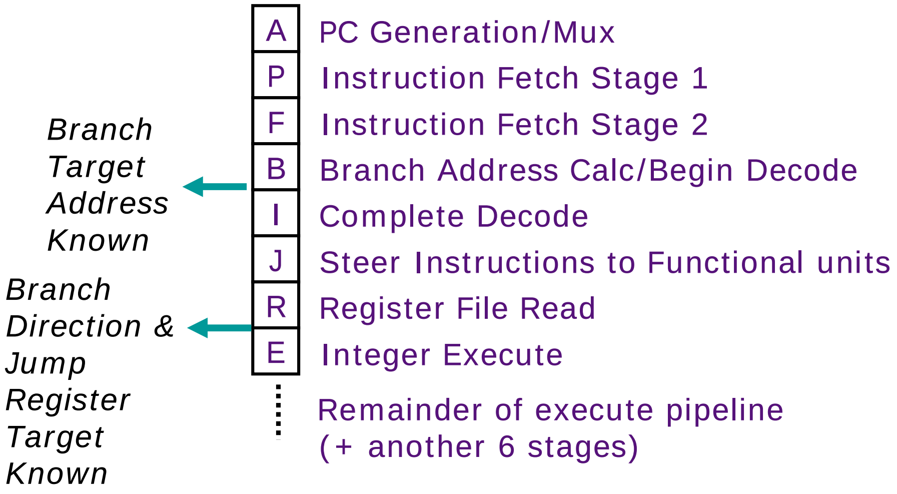
Name ______________________________

CS252 Graduate Computer Architecture
Midterm 1 Solutions

Part A: Branch Prediction (22 Points)

Consider a fetch pipeline based on the UltraSparc-III processor (as seen in Lecture 5). In this part, we evaluate the impact of branch prediction on the processor's performance. Assume 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 1. (12 Points)

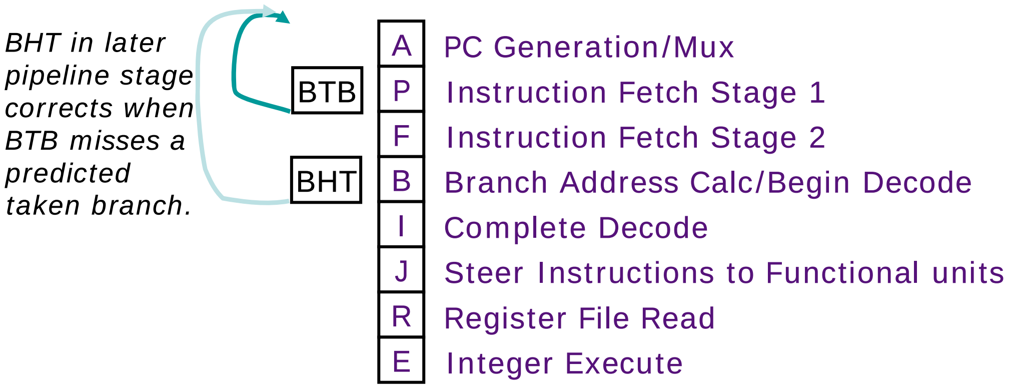
We add a branch history table (BHT) in the fetch pipeline as shown below. The A stage of the pipeline generates the next sequential PC address (PC+4) by default, and this is used to fetch the next instruction unless the PC is redirected by a branch prediction or an actual branch execution.

In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) checks the BHT, but an unconditional jump does not. If the branch is predicted to be taken, some of the instructions are flushed and the PC is redirected to the calculated branch target address. The BHT determines if the PC needs to be redirected at the *end* of the B stage and the new PC is inserted at the *beginning* of the A stage. (The A stage also contains the multiplexing circuitry to select the correct PC.)

To improve the branch performance further, we decide to add a branch target buffer (BTB) as well. Here is a description of 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 checked every cycle. If there is a match for the current fetch PC, the PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction). The BTB determines if the PC needs to be redirected at the *end* of the P stage and the new PC is inserted at the *beginning* of the A stage.

For a particular branch, if the BHT predicts "taken" and the BTB did not make a prediction, the BHT will redirect the PC. If the BHT predicts "not-taken" it does not redirect the PC.


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

Name ______________________________

a) Fill out the following table of the number of pipeline bubbles (only for conditional branches). (8 points)

	BTB Hit?	**(BHT) Predicted Taken?**	**Actually Taken?**	**Pipeline bubbles**
Conditional Branches	Y	Y	Y	1
	Y	Y	N	6
	Y	N	Y	1
	Y	N	N	6
	N	Y	Y	3
	N	Y	N	6
	N	N	Y	6
	N	N	N	0

For both part b) and part c) below, assume the BTB contains few entries and is fully associative, and the BHT contains many entries and is direct mapped.

b) For the case where the BHT predicts ‘not taken’ and the BTB predicts ‘taken’, why is it more probable that the BTB made a more accurate prediction than the BHT? (2 points)

Since the BTB stores the PC, we are certain that the BTB entry corresponds to that particular branch. Also, because the BTB is much smaller than the BHT, when there is a BTB hit, it is likely to be for a branch that was recently taken. The BHT is direct mapped and untagged and it is possible that there may have been conflicts, so the entry in the BHT may not actually belong to this particular branch. Therefore in this case, the BTB is more likely to make a better prediction.

c) For the case where the BHT predicts ‘taken’ and the BTB predicts ‘not taken’, why is it more probable that the BHT made a more accurate prediction than the BTB? (2 points)

If a branch is not found in the BTB, it could mean either of two things: 1) the branch was predicted not taken, or 2) the branch was predicted taken but it was kicked out of the BTB. Since the BTB has few entries, it is likely that the branch could have been kicked out of the BTB. Since the BHT has more entries, it is likely that the branch is still in the BHT, so the BHT will probably make a better prediction than the BTB in this case.

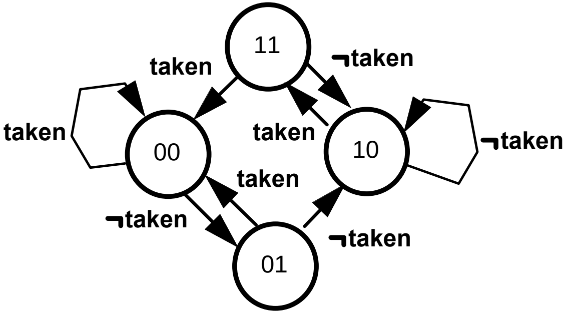
Question 2. (6 Points)

We will be examining the behavior of the branch prediction pipeline for 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

Name ______________________________

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								A	P	F	B	I	J	A	E								
0x1018	SLTI R2, R1, 100									A	P	F	B	I	J	A	E							
0x101C	BNEZ R2, BR1										A	P	F	B	I	J	A	E						
0x1000	BEQZ R5, NEXT												A	P	F	B	I	J	A	E				
0x1014	ADDI R1, R1, #1														A	P	F	B	I	J	A	E		

Timing diagram for Question 2

Question 3. (4 Points)

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

(Valid)		Predicted
V	Entry PC	Target PC
1	0x101c	0x1000
1	0x1000	0x1014

BTB

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

BHT

Part B: Cache Performance Analysis (19 points)

The following MIPS code loop adds the vectors X, Y, and Z and stores the result in vector W. All vectors are of length n and contain words of four bytes each.

```
W[i] = X[i] + Y[i] + Z[i]          for 0 <= i < n
```

Registers 1 through 5 are initialized with the following values:

- R1 holds the address of X
- R2 holds the address of Y
- R3 holds the address of Z
- R4 holds the address of W
- R5 holds n

All vector addresses are word aligned.

The MIPS code for our function is:

```
loop:
     LW   R6, 0(R1)  ; load X
     LW   R7, 0(R2)  ; load Y
     LW   R8, 0(R3)  ; load Z
     ADD  R9, R6, R7 ; do the add
     ADD  R9, R9, R8
     SW   R9, 0(R4)  ; store W
     ADDI R1, R1, 4  ; increment the vector indices
     ADDI R2, R2, 4  ; 4 bytes per word
     ADDI R3, R3, 4
     ADDI R4, R4, 4
     ADDI R5, R5, -1 ; decrement n
     BNEZ R5, loop
```

We run the loop using a cache simulator that reports the average memory access time for a range of cache configurations.

In Figure B-1 (on the next page) we plot the results of one set of experiments. For these simulations, we used a single level cache of *fixed capacity* and *varied the block size*. We used large values of n to ensure the loop settled into a steady state. For each block size, we plot the average memory access time, i.e., the average time in cycles required to access each data memory operand (cache hit time plus memory access time). Assume that capacity (C) and block size (b) are restricted to powers of 2 and measured in four byte words. The cache is fully associative and uses an LRU replacement policy. Assume the processor stalls during a cache miss and that the cache miss penalty is independent of the block size.

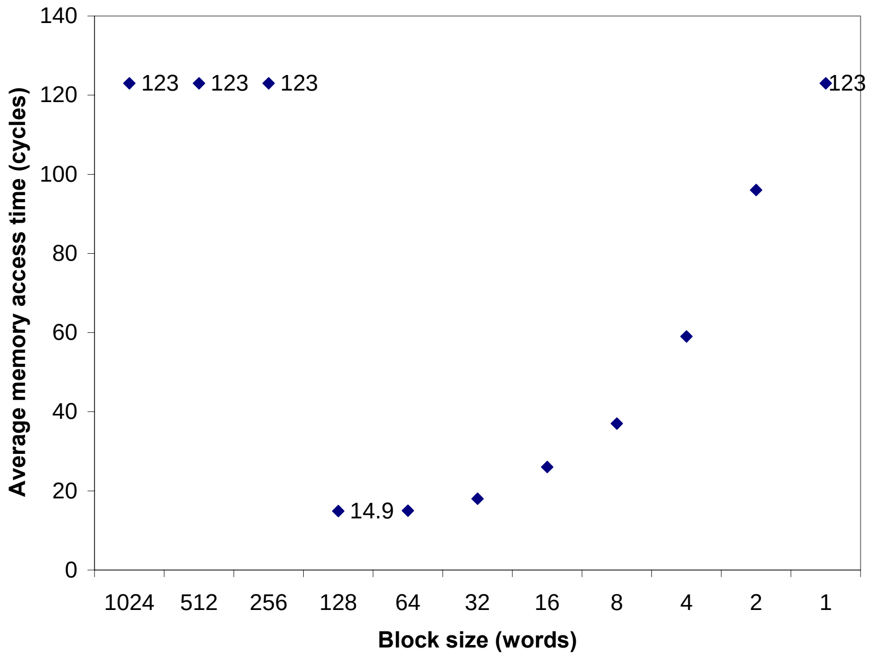


Figure B-1: Simulator Results For Varying Block Size

Question 4 (14 points)

a) Based on the results in Figure B-1, what is the cache capacity C? (6 points)

When C<4b, there are less than 4 lines in the cache, so all 4 vectors (X, Y, Z, W) cannot fit in the cache at the same time. With an LRU replacement policy, the cache line for each vector's access will be evicted each time around the loop, resulting in a miss on every access.

When C=>4b, there are enough cache lines to fit all 4 vectors. Therefore a vector's block will survive in cache until the next iteration. A number of hits will occur and either the function will be completed, or the end of the cache line will be reached and the next cache line from the vector will have to be loaded in. Larger values of b will result in fewer compulsory misses, and hence smaller average memory access times.

Therefore, the curve should drop at C=4b and then climb again as b decreases.

On the curve above, the drop is at b=128. Therefore C=512 words or 2K bytes.

b) Assume that the cache capacity, C, remains the same as in a). When b is ≥ 256, what is the relation for the average memory access time in terms of 'b', the hit time 'h', the miss penalty 'p', and the number of elements in each vector 'n'? (4 points)

When b>=256 (C<4b), all accesses will be misses.

The average memory access time will be:

`Tave = h + p,` (where h+p is the total time for servicing a miss)
or
`Tave = p,` (where p is the total time for servicing a miss)

c) Assume that the cache capacity, C, remains the same as in a). What is the relation for the average memory access time when b is ≤ 128? (4 points)

When b<=128, C>=4b. There will be 1 cold start miss for a given vector, then b-1 hits for the remaining elements on each cache line. This means there will be 1 miss every b accesses, resulting in a miss rate of 1/b.

```
Tave = h + p*(miss rate) = h + p/b
```

Alternatively, if we assumed that p is the total time for servicing a miss, we would get

```
Tave = (hit rate) * h + (miss rate) * p
     = (1 - (1/b))*h + (1/b)*p
```

Question 5 (5 points)

Now let us replace the data cache with one that consists of only a SINGLE block (with 4 words total cache capacity). Explain how to rewrite the vector code to get better performance from the single-block cache. Note that total code size is not a constraint in this problem.

Only a single 4-word chunk from one of the vectors can be contained in the block-sized cache at any given moment in time. Thus, in order to optimize the loop, we must unroll the loop 4 times, and load each successive element into a different register. If we make 4 consecutive accesses to the same vector, than we can amortize the cost of a miss with 3 subsequent hits.

This solution results in 5 memory operations each loop-unrolled iteration: 4 memory accesses corresponding to 1 miss for each vector, and 1 memory operation corresponding to writing back W to memory before loading again. This results in:

```
 5 memory ops/iteration * (n/4) iterations =
                                    5n/4 total memory ops
or

5/4 memory ops per original loop iteration
```

For the solution below, we assume the (R5)%4 = 0 (R5 contains the loop counter) since loop execution is works on 4 elements per iteration.

```
loop:
     LW   R6, 0(R1)      ; load $X_1$
     LW   R10, 4(R1)     ; load $X_2$
     LW   R14, 8(R1)     ; load $X_3$
     LW   R18, 12(R1)    ; load $X_4$
     LW   R7, 0(R2)      ; load $Y_1$
     LW   R11, 4(R2)     ; load $Y_2$
     LW   R15, 8(R2)     ; load $Y_3$
     LW   R19, 12(R2)    ; load $Y_4$
     LW   R8, 0(R3)      ; load $Z_1$
     LW   R12, 4(R3)     ; load $Z_2$
     LW   R16, 8(R3)     ; load $Z_3$
     LW   R20, 12(R3)    ; load $Z_4$
     ADD  R9, R6, R7     ; do the add for element 1
     ADD  R9, R9, R8
     ADD  R13, R10, R11 ; do the add for element 2
     ADD  R13, R13, R12
     ADD  R17, R14, R15 ; do the add for element 3
     ADD  R17, R17, R16
     ADD  R21, R18, R19 ; do the add for element 4
     ADD  R21, R21, R20
     SW   R9, 0(R4)      ; store $W_1$
     SW   R13, 4(R4)    ; store W2
     SW   R17, 8(R4)    ; store W3
     SW   R21, 12(R4)   ; store W4
     ADD  R1, R1, 16    ; increment the vector indices
     ADD  R2, R2, 16    ; 4 bytes per word
     ADD  R3, R3, 16
     ADD  R4, R4, 16
     SUB  R5, R5, 4  ; decrement 4
     BNEZ R5, loop
```

We gave partial credit for the following answer:

Interleave the elements of the vectors X, Y, Z, W, so that one cache block will contain one element from each array.

This answer results in 1 miss per iteration of the loop. However, since we store to W, we need to mark the block dirty and write it back to memory on each iteration before we fetch in the next block. This results in 2 memory operations every iteration of the loop, resulting in a total of 2n memory operations. The previous solution requires only 5/4 memory operations per iteration so is a better answer.

Part C: VLIW machines (19 points)

In this question, we will deal with the following C code sample on a VLIW machine. You should assume that arrays **A**, **B** and **C** do not overlap in memory.

C code

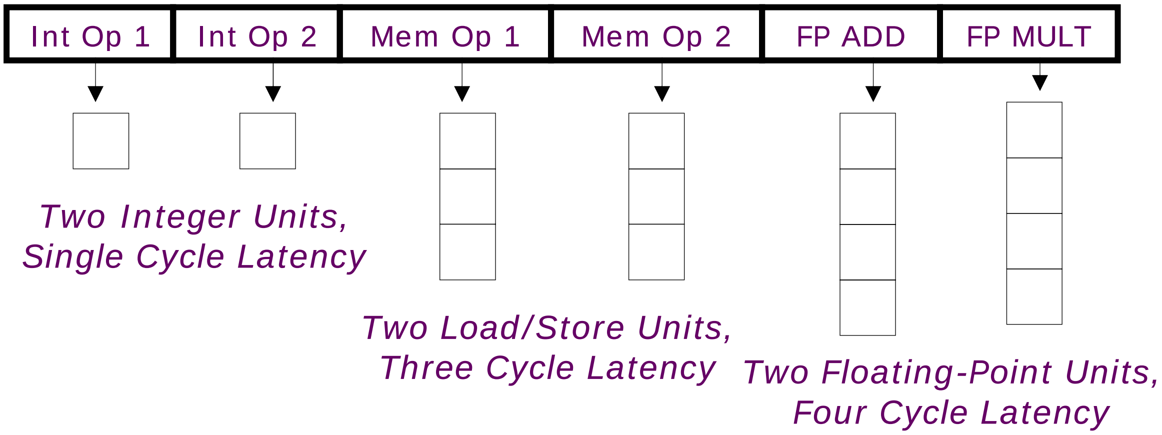
```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

Our machine will have six execution units:

- Two ALU units, latency one cycle, also used for branch instructions
- Two memory units, latency three cycles, fully pipelined, each unit can perform either a store or a load
- Two FPU units, latency four cycles, fully pipelined, one unit can perform **`fadd`** instructions, the other **`fmul`** instructions.

Our machine has no interlocks, so the programmer has to make sure that appropriate data is in the registers when an instruction to read it is issued. Assume that the latency of each execution unit includes writing back to the register file.

Below is a diagram of our VLIW machine:



The C code above translates to the following instructions:

```
loop:    ld f1, 0(r1)        ; (I1) f1 = A[i]
         ld f2, 0(r2)        ; (I2) f2 = B[i]
         fmul f4, f2, f1     ; (I3) f4 = f1*f2
         st f4, 0(r1)        ; (I4) A[i] = f4
         ld f3, 0(r3)        ; (I5) f3 = C[i]
         fadd f5, f4, f3     ; (I6) f5 = f4+f3
         st f5, 0(r3)        ; (I7) C[i] = f5
         add r1, r1, 4       ; (I8) i++
         add r2, r2, 4       ; (I9)
         add r3, r3, 4       ; (I10)
         add r4, r4, -1      ; (I11)
         bnez r4, loop       ; (I12)loop
```

Question 6 (3 points)

Table C-1, on the next page, shows our program rewritten for our VLIW machine, with some instructions missing (I2, I6, and I7). We have rearranged the instructions to execute as soon as they possibly can, but ensuring program correctness.

Please fill in missing instructions in Table C-1. (Note, you may not need all the rows)

ALU1	ALU2	MU1	MU2	FADD	FMUL
add r1, r1, 4	add r2, r2, 4	Ld f1, 0(r1)	ld f2, 0(r2)		
add r3, r3, 4	add r4, r4, -1	Ld f3, 0(r3)			
					fmul f4, f2, f1
			st f4, -4(r1)	fadd f5, f4, f3	
	bnez r4, loop	st f5, -4(r3)			

Table C-1: VLIW Program (For Problem 6)

Question 7 (2 points)

How many cycles does it take to complete one execution of the loop in Table C-1? What is the performance (flops/cycle) of the program?

12 cycles to complete one execution of the loop.

2/12 flops per cycle

Question 8 (5 points)

If you were to software pipeline the VLIW code in Table C-1 with the goal of obtaining the smallest software pipelined loop body, how many VLIW instructions would it contain? Ignore the prologue and epilogue, we are only interested in the number of static VLIW instructions in the repeated loop. Hint: You do not need to write the code for the software pipelined version.

3 instructions, because there are 5 memory ops and 5 alu ops, and we can only issue 2 of those per instruction. Note, one does not need to write the code to get an answer, because the code is highly data parallel and there is no limitation on register usage because of the lack of interlocks (we can keep reading the old value from a register while an instruction that will overwrite it is in flight). It's just a question of how many VLIW instructions are needed to express all the operations.

Question 9 (2 points)

What would be the performance (flops/cycle) of the software pipelined program from Question 8? How many iterations of the loop would we have executing at the same time?

2/3 flops per cycle

4 iterations at a time

Question 10 (5 points)

If we unrolled the loop in Table C-1 once (two copies total) and then software pipelined it, could that give us better performance than the solution to Question 8? Explain. How many flops/cycle can we get?

If we unrolled the loop once, we would double the number of memory ops (5 to 10) and floating point ops (2 to 4). We don't need to double the number of adds and bnez. When we pipeline this unrolled loop, we are constrained by the number of memory units. Now, we need a total of 5 VLIW instructions to do all 10 memory ops. This gives us 4/5 flops per cycle, which gives us better performance than the solution in Question 8.

Question 11 (2 points)

What is the maximal flops/cycle for this program on this VLIW architecture? Explain.

Same as above: 4/5 flops/cycle. We are fully utilizing the memory units, so we can't execute more loops/cycle.

Part D: Out-of-order Execution with Unified Physical Registers (20 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 5.

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
```

Figure D-1, on the page after next, 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.

Question 12 (16 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).

Step 2. The second instruction from the next loop iteration (addi) is written into the ROB.

Step 3. The third instruction from the next loop iteration (beqz) is written into the ROB.

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.

Step 5. As many instructions as possible commit.

Update Figure D-1 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.)

An extra copy of Figure D-1 is attached at the end of the midterm, in case you need it.

Name ______________________________

Rename Table

R1	~~P1~~	~~P4~~	P7
R2	~~P2~~	~~P5~~	P8
R3	~~P3~~	P6	
R4	P0		

Physical Regs

P0	8016	p
P1	~~6823~~	~~p~~
P2	~~8000~~	~~p~~
P3	7	p
P4	0	p
P5	8004	p
P6	8	p
P7		
P8		
P9		

Free List

~~P4~~
~~P5~~
~~P6~~
~~P7~~
~~P8~~
P9
P1
P2
⋮

Reorder Buffer (ROB)

next to commit →

next available →

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
~~x~~	x	lw	p	P2			r1	P1	P4
~~x~~	x	addi	p	P2			r2	P2	P5
x		beqz	p	P4					
x	x	addi	p	P3			r3	P3	P6
x		bne	p	P5	p	P0			
x		lw	p	P5			r1	P4	P7
x		addi	p	P5			r2	P5	P8
x		beqz		P7					

Figure D-1 for Problem 12

If you assumed in step 4 to propagate the execution of instructions as earlier instructions wake up later ones, we also gave you credit.

Question 13 (4 points)

Consider (1) a superscalar version of the out-of-order machine described in Part D and (2) a single-issue, in-order processor with no branch prediction. Assume that both processors have the same clock frequency and the same latencies for all functional units. Consider the steady state execution rate of the given loop, assuming that it executes for many iterations.

a) Under what conditions, if any, might the loop in Part D execute at a faster rate on the in-order processor compared to the out-of-order processor? (2 points)

If the out-of-order processor frequently mispredicts either of the branches, it is likely to execute the loop slower than the in-order processor. For this to be true, we must also assume that the branch misprediction penalty of the out-of-order processor is sufficiently longer than the branch resolution delay of the in-order processor (this is likely to be the case). The mispredictions may be due to deficiencies in the out-of-order processor's branch predictor, or the data-dependent branch may be fundamentally unpredictable in nature.

b) Under what conditions, if any, might the program in Part D execute at a faster rate on the superscalar out-of-order processor compared to the in-order processor? (2 points)

If the out-of-order processor predicts the branches with high enough accuracy, it can execute more than one instruction per cycle, and thereby execute the loop at a faster rate than the in-order processor.

Additionally, if there is a high cache miss rate (and memory accesses take a long time), the out-of-order processor can hide the memory latency by executing later instructions, while the in-order processor needs to stall.