Computer Architecture and Engineering
CS152 Quiz #1

February 14th, 2012

Professor Krste Asanović

Name:______________________

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.
- 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. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Writing name on each sheet	________	1 Points
Question 1	________	10 Points
Question 2	________	25 Points
Question 3	________	20 Points
Question 4	________	24 Points
TOTAL	________	**80 Points**

Name ______________________________

Question 1: ISA Visibility (10 points)

Do the following modifications change the ISA? Circle **yes** or **no**.
Explain your reasoning to receive credit.

Problem 1.A — 64-bit addresses

Changing to using 64-bit addresses from 32-bit addresses.

[✓ **YES**] **NO**

The PC register, and the width of all general-purpose registers must change (to hold 64-bit addresses for JR, LW, etc.), which can't be hidden from the software (for example, shift instructions would behave differently). (-1 point for saying instructions would be different length).

```
li    x1, 0xffff_ffff
srai x2, x1, 16          (does x2 equal 0xffff, or does x2 equal 0xffffffff?)
```

Problem 1.B — branch delay slots

Removing branch delay slots.

[✓ **YES**] **NO**

This changes the behavior of instructions following branches, which compilers must know about to schedule instructions properly. It would certainly break all old software which expects those instructions following branches to execute regardless of branch direction.

Problem 1.C — bus-based micro-architecture

Building a bus-based (micro-coded) micro-architecture, instead of a 5-stage pipeline micro-architecture.

YES [✓ **NO**]

A bus-based design and a 5-stage pipeline design are just different implementations (aka, micro-architectures) and are orthogonal to the ISA (the hardware designer can choose which micro-architecture to implement a given ISA).

Name ______________________________

Problem 1.D **branch-target buffer**

Adding a branch-target buffer, instead of statically predicting PC+4.

YES [✓ **NO**]

Branch speculation is just predicting which instruction to fetch next, and doesn't affect the actual machine state (Regfile, Memory). It only hopes to decrease the CPI and improve the rate at which machine state is changed.

Problem 1.E **registers**

Adding more registers that user code can now address.

[✓ **YES**] **NO**

Adding more registers would require changing instruction encodings (or adding new instructions that can access the higher-numbered registers), and thus be a change to the ISA.

Name ____________________________

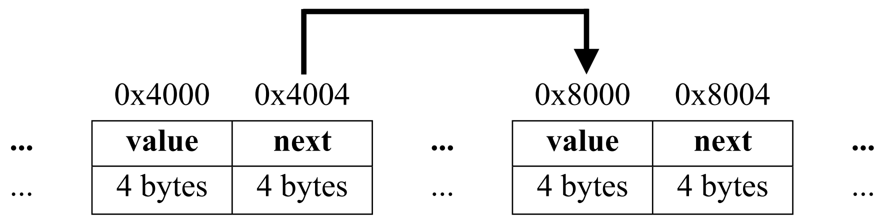
Question 2: Microprogramming (25 points)

In this question we ask you to implement a useful instruction for manipulating linked lists, *Add to Linked List* (**`AddToLinkedList`**).

Each node in the linked list is as follows:

```
struct Node {
     int   value;
     Node* next;
}
```

Both “int” and “Node*” are 4 bytes in size. Thus the node looks like this in memory:



The field “next” is either a memory address pointing to the next Node, or is 0 (NULL), representing the end of the list.

The `AddToLinkedList` instruction walks a linked list, finds the last Node, and updates its “next” field to point to the new Node.

```
AddToLinkedList rs1, rs2
```

5	5	5	10	7
-	**rs1**	**rs2**	**unused**	**`AddToLinkedList`**

The memory location addressed by **rs1** (**M[rs1]**), points to the first node in the linked list. The register **rs2** holds the address (**M[rs2]**) that points to the new Node.

Starting from the memory location addressed by **rs1** (**M[rs1]**), keep walking the list until you find “next” == 0. Once found, replace “next” with the address stored in **rs2**.

For full credit, both source registers **rs1** and **rs2** should remain untouched throughout the execution of `AddToLinkedList` (when finished executing the instruction, **rs1** still points to the head of the list, and **rs2** still points to the new Node).

In pseudo C code, `AddToLinkedList` behaves as follows:

```
struct Node {
     int   value;
     Node* next;
}

void AddToLinkedList(Node* head_ptr, Node* new_ptr)
{
   Node* curr_ptr = head_ptr;

   while (curr_ptr->next != 0)
   {
      curr_ptr = curr_ptr->next;
   }

   curr_ptr->next = new_ptr;
}
```

For reference, we have included the actual bus-based datapath in Appendix A (Page 21) and a RISC-V instruction table in Appendix B (Page 22)**.** You do not need this information if you remember the bus-based architecture from the online material. **Please detach the last two pages from the exam and use them as a reference while you answer this question.**

Q2.A Microcoding (16 points)

Fill out Worksheet 1 for the `AddToLinkedList` instruction. You should try to optimize your implementation to reduce the number of cycles necessary and to have as many signals be "don't cares" as possible. You may not need all the lines in the table for your solution.

Remember: memory access can be variable latency, so you will have to make use of the "S" micro-branch signal when appropriate (just like in Problem Set #1).

Name ______________________________

(scratch paper)

First, it may be helpful to first write out a lower-level of Psuedo C code:

```
void AddToLinkedListLowLevel(Node* head_ptr, Node* new_ptr) //aka, (rs1, rs2) are the inputs
{
  int a,b;

  a = head_ptr; //rs1
  a =  a + 4;
  b = a

  a = *(a);

  while (a != 0)
  {
    b = a + 4;
    a = *(b);
  }

  *(b) = new_ptr;
}
```

More optimally:

```
void AddToLinkedListLowLevel(Node* head_ptr, Node* new_ptr) //aka, (rs1, rs2) are the inputs
{
  int a,ma;
  a = head_ptr; //rs1

do {
  ma =  a + 4;
   a = *(a);
} while (a != 0);

  *(ma) = new_ptr; //rs2
}
```

Now we can more easily translate this C code into psuedo micro-ops

```
AddToLinkedList:
   A     <- rs1           # head_ptr
Loop:
   MA    <- A+4           # point to next_ptr
   A     <- Mem           # load value of next_ptr
  if (A!=0) ubr to LOOP   # points to null? We're done.
Done:
   Mem   <- rs2           # update the last next_ptr to point to new_ptr
   ujmp  Fetch
```

both methods are shown below. Full credit giving for the more optimal design.

Name ______________________________

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Im Sel	en Imm	µBr	Next State
FETCH:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	S	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH
ATLL:	A <- rs1	0	rs1	0	1	1	*	*	0	*	*	0	*	0	N	
LOOP:	MA<- A+4	0	*	*	0	*	*	INC_A_4	1	1	*	0	*	0	N	
	A <- Mem	0	*	*	0	1	*	*	0	0	0	1	*	0	S	
	If A!=0 j loop	0	*	*	0	0	*	COPY_A	0	0	*	0	*	0	NZ	LOOP
DONE:	Mem <- rs2	0	rs2	0	1	*	*	*	0	0	1	1	*	0	S	
	Jmp to Fetch	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Im Sel	en Imm	µBr	Next State
FETCH:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	S	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH
ATLL:	A <- rs1	0	rs1	0	1	1	*	*	0	*	*	0	*	0	N	
	MA,B<- A+4	0	*	*	0	*	1	INC_A_4	1	1	*	0	*	0	N	
	A <- Mem	0	*	*	0	1	0	*	0	0	0	1	*	0	S	
LOOP:	If A==0 Jmp to Done	0	*		0	0	0	COPY_A	0	*	*	0	*	0	EZ	DONE
	B,MA <- A+4	0	*	*	0	*	1	INC_A_4	1	1	*	0	*	0	N	
	A <- Mem	0	*	*	0	1	0	*	0	0	0	1	*	0	S	
	Jmp to Loop	0	*	*	0	0	0	*	0	*	*	0	*	0	J	LOOP
DONE:	MA <- B	0	*	*	0	*	*	COPY_B	1	1	*	0	*	0	N	
	Mem <- rs2	0	rs2	0	1	*	*	*	0	0	1	1	*	0	S	
	Jmp to Fetch	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH

Q2.B CPI of AddToLinkedList (2 points)

For a benchmark in which the average linked list contains 7 elements, what CPI do you expect your implementation of `AddToLinkedList` to achieve?

instruction fetch = 3 uops
1 element list = 3+6 uops (base case)
2 element list = 3+9 uops
3 element list = 3+12 uops
etc.

3uops in inst fetch 1 uop in the prologue, 3 in the loop (executed N times), 2 in the epilogue.

CPI = 3+ 1 + 3*N + 2 = 6 + 3*N = 6 + 21
CPI = 27 cycles on average for this benchmark

Q2.C Precise Exceptions (4 points)

Briefly describe how you might implement precise exceptions for this instruction in a microcoded machine (you do not have to actually write microcode).

First, you must decrement the PC register by 4, since it was speculatively incremented to PC+4 in the instruction fetch. Then, you can jump to the exception handler routine, and then restart execution of ATLL and try again. This works, because ATLL doesn't change any of the state of the machine until the very last write to memory, and is idempotent. Notice, we don't have to save any of the temporary state (A,B,MA) because it can be re-created by starting ATLL from the beginning. Students proposed other schemes, but without describing how to save and restore the temporary state, full credit couldn't be awarded.

Q2.D Issues with Precise Interrupts (3 points)

What practical problems might arise with the instruction `AddToLinkedList` in a system with precise interrupts?

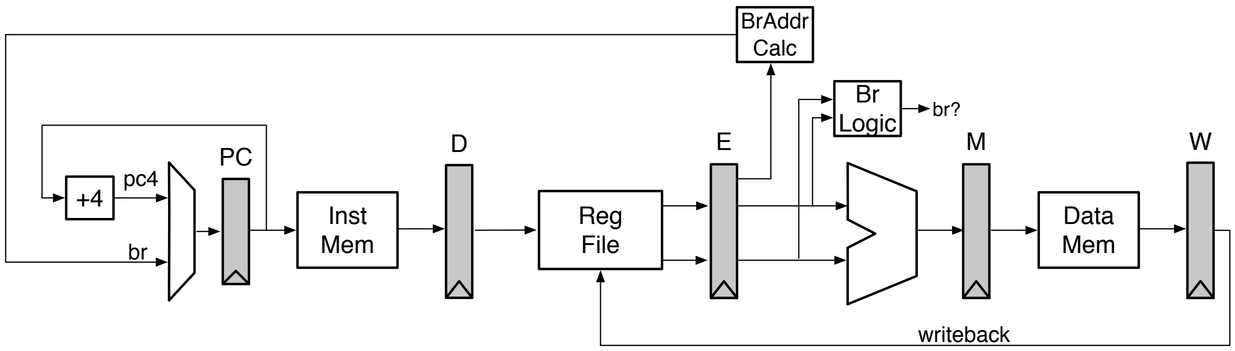
If the linked list is very long, the probability of it getting interrupted increases (by system interrupts, etc.). Because we will restart the instruction over again from the very beginning, we can't actually guarantee it will ever finish before it gets interrupted again!

(Although this is more related to Unit 2 material, you can also think about how each memory access could cause a page fault/TLB miss, which throws an exception and requires an exception handler to service the page miss. If the linked list is very long, the TLB may not be able to hold all of the pages required to keep the full linked list resident, and so running ATLL causes continuous TLB misses that keep undoing each other and forcing a restart of the instruction from the very beginning, and thus ensuring that no forward progress can be made).

Name ____________________________

Question 3: Branch Speculation (20 points)

For this question, consider a fully bypassed 5-stage RISC-V processor (as shown in Lecture 4, and used in Lab 1). We have reproduced the pipeline diagram below (bypasses are not shown). Branches are resolved in the **Execute Stage**, and the **Fetch Stage** always speculates that the next PC is PC+4. For this problem, we will ignore unconditional jumps, and only concern ourselves with conditional branches.



Q3.A Motivating Branch Speculation (2 points)

To get a better understanding of how the pipeline behaves, please fill out the following instruction/time diagrams for the following set of instructions:

```
0x2000: ADDI x4, x0, 0
0x2004: ADDI x5, x0, 1
0x2008: BEQ  x4, x5, 0x2000
0x200c: LW   x7, 4(x6)
0x2010: OR   x5, x7, x5
0x2014: XOR  x7, x7, x3
0x2018: AND  x3, x2, x3
```

The first two instructions have been done for you. Please fill out the rest of the diagram for the remaining instructions. The sequence of instructions may finish before cycle t_{12}.

Hint: Throughout this question, make sure you also show speculated instructions in the instruction/time diagrams (exactly as the Trace files in Lab 1 would), as they also consume pipeline resources.

```
0x2000: ADDI x4, x0, 0
0x2004: ADDI x5, x0, 1
0x2008: BEQ  x4, x5, 0x2000
0x200c: LW   x7, 4(x6)
0x2010: OR   x5, x7, x5
0x2014: XOR  x7, x7, x3
0x2018: AND  x3, x2, x3
```

Name ______________________

Chart 1: Using Standard Always Predict PC+4

PC	Instr	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
0x2000	ADDI	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ			F	D	X	M	W						
0x200c	LW				F	D	X	M	W					
0x2010	OR					F	D	D	X	M	W			
0x2014	XOR						F	F	D	X	M	W		
0x2018	AND								F	D	X	M	W	

The point in time that a branch comparison occurs is circled above. The second circle (OR) is when the decode stage recognizes a dependent load in the EXE stage (at t5) and stalls.

Q3.B Motivating Branch Speculation II (2 points)

Fill in the following pipeline diagram (Chart 2), using the code segment below. *Notice*, the immediates used in the first two instructions (ADDI) are different from the previous question!

```
0x2000: ADDI x4, x0, 1
0x2004: ADDI x5, x0, 1
0x2008: BEQ  x4, x5, 0x2000
0x200c: LW   x7, 4(x6)
0x2010: OR   x5, x7, x5
0x2014: XOR  x7, x7, x3
0x2018: AND  x3, x2, x3
```

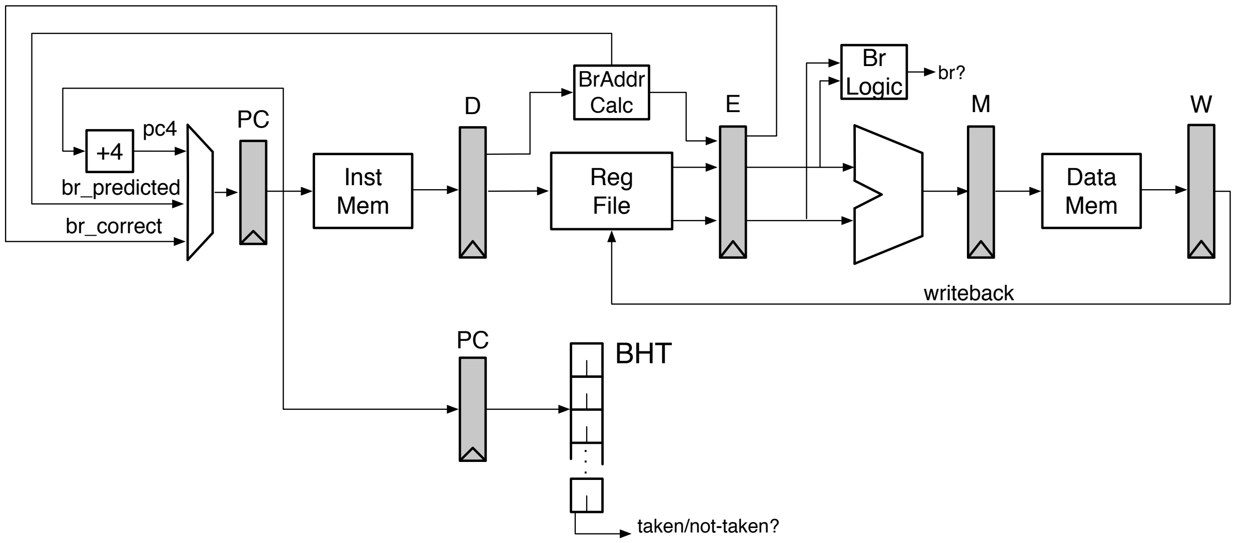
PC	Instr	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
0x2000	ADDI	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ			F	D	X	M	W						
0x200c	LW				F	D	-	-	-					
0x2010	OR					F	-	-	-	-				
0x2000	ADDI						F	D	X	M	W			
0x2004	ADDI							F	D	X	M	W		

The IF_KILL and DEC_KILL signal goes out in t5, when the "mispredict" is discovered. Bubbles are inserted into the pipeline, and show up on t6.

Q3. Adding a BHT

As you showed in the first parts of this question, branches in RISC-V can be expensive in a 5-stage pipeline. One way to help reduce this branch penalty is to add a Branch History Table (BHT) to the processor.

This new proposed datapath is shown below:



The BHT has been added in the **Decode Stage**. The BHT is indexed by the PC register in the **Decode Stage**. Branch address calculation has been moved to the **Decode Stage**. This allows the processor to redirect the PC if the BHT predicts *"Taken"*.

On a BHT mis-prediction, (1) the branch comparison logic in the **Execute Stage** detects mis-predicts, (2) kills the appropriates stages, and (3) starts the **Instruction Fetch** using the correct branch target (*br_correct*).

Remember: the **Fetch Stage** is still predicting PC+4 every cycle, unless corrected by either the BHT in the **Decode Stage**(*br_predicted*) or by the branch logic in the **Execute Stage** (*br_correct*).

Name ______________________________

Q3.C BHT Performance (9 points)

Using the code segment below, fill in the following pipeline diagram. **Initially**, the BHT counters are all initialized to *"strongly-taken"*. The register **x2** is initialized to 0, while the register **x3** is initialized to 2. The first instruction has been done for you. It is okay if you do not use the entire table.

```
0x2000: LW   x7, 0(x6)
0x2004: ADDI x2, x2, 1
0x2008: BEQ  x2, x3, 0x2000
0x200c: SW   x7, 0(x6)
0x2010: OR   x5, x5, 4
0x2014: OR   x7, x7, 5
```

Loop is not taken. The BHT is "strongly" taken, so BHT predicts "taken" when we see BEQ. BHT is in Decode, and Fetch stage always predicts PC+4, so we eat 1 cycle when the PHT predicts taken branch, and we eat another cycle if BHT predicts taken, but branch is actually not taken (i.e., it just degregates to the original 2-cycle branch penalty).
At t4 (as circled), the Decode stage kills the fetch stage to redirect it down the "taken" path. However, at t5 we resolve the branch comparison in Execute and must correct for the BHT's misprediction, and kill Fetch and Decode.

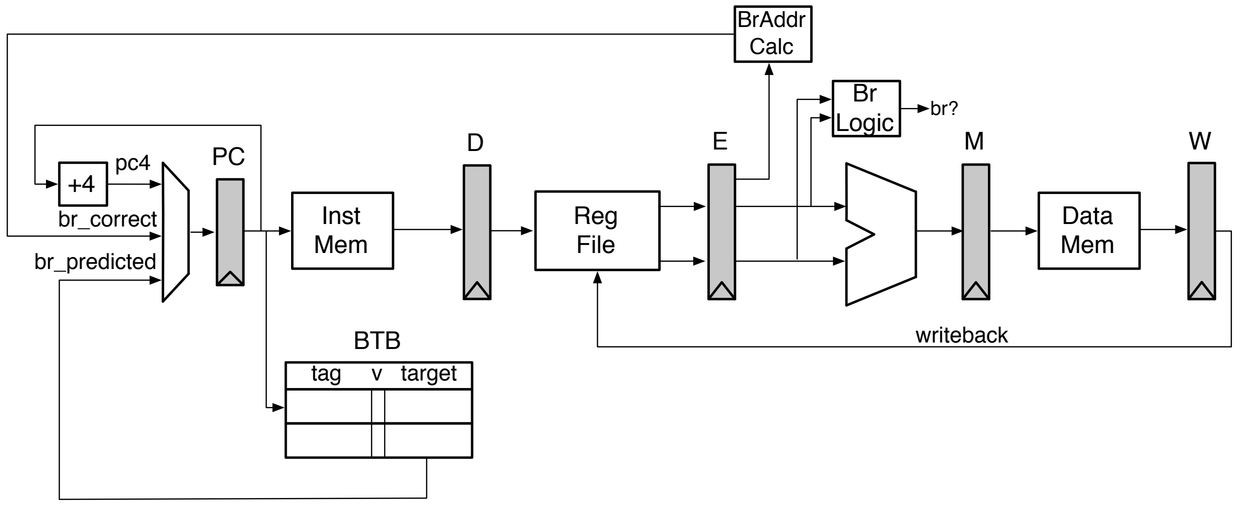
PC	Ins tr	t_1	t_2	t_3	t_4	t_5	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}
0x2000	LW	F	D	X	M	W													
0x2004	ADDI		F	D	X	M	W												
0x2008	BEQ			F	D	X	M	W											
0x200c	SW				F	-	-	-	-										
0x2000	LW					F	-	-	-	-									
0x200c	SW						F	D	X	M	W								
0x2010	OR							F	D	X	M	W							
0x2010	OR								F	D	X	M	W						

Name ____________________________

Q3. Adding a BTB

Unfortunately, while the BHT is an improvement, we still have to wait until we know the branch address to act on the BHT's prediction. We can solve this by using a two-entry Branch Target Buffer (BTB).

The new pipeline is shown below. For this question, *we have removed the BHT and will only be using the BTB.*



The BTB has been added in the **Fetch Stage**. The BTB is indexed by the PC register in the **Fetch Stage**. Branch address calculation has been moved back in the **Execute Stage**.

On a branch mis-prediction, (1) the branch comparison logic in the **Execute Stage** detects the mis-predict, (2) kills the appropriates stages, and (3) starts the **Instruction Fetch** using the correct branch target (*br_correct*).

Remember: the **Fetch Stage** is still predicting PC+4 every cycle, unless either the BTB makes a prediction (has a matching and valid entry for the current PC) or the branch logic in the **Execute Stage** corrects for a branch mis-prediction (*br_correct*).

Name ______________________________

Q3.D BTB Performance (7 points)

Using the code segment below (the exact same code from **Q3.C**), fill in the following pipeline diagram. Upon entrance to this code segment, the register **x2** is initialized to 0, while the register **x3** is initialized to 2.

```
0x2000: LW   x7, 0(x6)
0x2004: ADDI x2, x2, 1
0x2008: BEQ  x2, x3, 0x2000
0x200c: SW   x7, 0(x6)
0x2010: OR   x5, x5, 4
0x2014: OR   x7, x7, 5
```

Initially, the BTB contains:

Tag	V	Target PC
0x2008	1	0x2000
0x201c	0	0x2010

(For simplicity, the Tag is 32-bits, and we match the entire 32-bit PC register in the **Decode Stage** to verify a match). It is okay if you do not use the entire instruction/time table.

BTB mispredicts the exit, and it takes two cycles for branch logic in Exe to catch the mistake.

The first circle is drawn to show when the BTB had a hit and predicted "taken". The second circle in t3 shows when the branch comparison catches a mispredict and kills two cycles.

PC	Instr	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
0x2000	LW	F	D	X	M	W													
0x2004	ADDI		F	D	X	M	W												
0x2008	BEQ	F	D	X	M	W													
0x2000	LW		F	D	-	-	-												
0x2004	ADDI			F	-	-	-	-											
0x200c	SW				F	D	X	M	W										
						F	D	X	M	W									
							F	D	X	M	W								

Name ____________________________

Question 4: Iron Law of Processor Performance

(24 points)

Mark whether the following modifications will cause each of the *three* categories to **increase, decrease**, or whether the modification will have **no effect**.

Assume the rest of the machine remains unchanged. Also, we are measuring these metrics from the viewpoint of the user code. Thus, an Operating System call will simply appear to be a single instruction that takes many, many cycles to execute.

Explain your reasoning to receive credit.

Name ______________________________

		Instructions / Program	Cycles / Instruction	Seconds / Cycle
a)	Changing a 2-stage processor (fetch,execute) into a 3-stage processor (fetch,execute,writeback). Both processors are fully bypassed.	unchanged (not ISA visible)	unchanged (Writeback occurs well after branch resolution, load-use, and any other possible stall conditions. So no additional stalls will occur due to adding a writeback stage.)	decreases (We cut down on the critical path by moving register-write to an extra stage.)
b)	Move the branch and jump logic from the Execute stage to the Decode stage.	unchanged (not ISA visible)	decreases (One less stage gets killed on a branch mispredict.)	increases (more logic is added to the critical path in Decode: register file read->register-register comparison->pc select. This is likely to become the critical path and thus hurt seconds per cycle.)

Worksheet 2

Name ______________________________

		Instructions / Program	Cycles / Instruction	Seconds / Cycle
c)	Switching from a fully interlocked 5-stage design to a fully bypassed 5-stage design.	unchanged (not ISA visible)	decreases (Far fewer stalls will now occur due to back-to-back read-after-write hazards.)	increases (The bypasses will add to the critical path.)
d)	Merge the *Decode* and *Execute* stages into a single stage (i.e., perform a register read, then an ALU execution in the same cycle).	unchanged (not ISA visible)	decreases will decrease due to branches/ jumps in more detail: Since branches are resolved in *Execute*, branch CPI goes down because instead of eating two cycles on a mispredict, we only lose 1 (because of the merged Dec+Exe) Common mistake to think we lose some bypassing potential (we can still bypass back-to-back instructions by bypassing from beginning of Memory to before the ALU.	increases (More logic is being performed in a single stage, thus hurting the critical path.)

Worksheet 3