CS152 Computer Architecture and Engineering

ISAs, Microprogramming and Pipelining Problem Set #1

Assigned 8/26/2016 *Due September 13*

---

http://inst.eecs.berkeley.edu/~cs152/fa16

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in his own solution to the problems.

The problem sets also provide essential background material for the quizzes. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the quizzes! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Late homework will not be accepted, except for extreme circumstances and with prior arrangement.

Update History:

- September 14, 2016 (4:47pm): Fixed annotation for BEQ in Problem 2.C (both annotations used to say x1 != x2).
- September 14, 2016 (10:19pm): Fixed addresses in Problem 4.B.

Problem 1: CISC, RISC, accumulator, and Stack: Comparing ISAs

In this problem, your task is to compare four different ISAs. x86 is an extended accumulator, CISC architecture with variable-length instructions. RISC-V is a load-store, RISC architecture with fixed-length instructions (for this problem only consider the 32-bit form of its ISA). We will also look at a simple stack-based ISA and an accumulator architecture.

Problem 1.A CISC

Let us begin by considering the following C code:

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, and register `%edx` contains `result`, and register `%eax` contains `a`. `b` is stored in memory at location `0x8049580`)

```
         xor    %edx,%edx
         xor    %ecx,%ecx
loop:    cmp    0x8049580,%ecx
         jl     L1
         jmp    done
L1:      add    %eax,%edx
         inc    %ecx
         jmp    loop
done:  ...
```

The meanings and instruction lengths of the instructions used above are given in the following table. Registers are denoted with $R_{SUBSCRIPT}$, register contents with $<R_{SUBSCRIPT}>$.

Instruction	Operation	Length
add R_{DEST}, R_{SRC}	$R_{SRC} <= <R_{SRC}> + <R_{DST}>$	2 bytes
cmp imm32, R_{SRC2}	Temp $<= <R_{SRC2}> -$ MEM[imm32]	6 bytes
inc R_{DEST}	$R_{DEST} <= <R_{DEST}> + 1$	1 byte
jmp label	jump to the address specified by `label`	2 bytes
jl label	if (SF != OF) jump to the address specified by `label`	2 bytes
xor R_{DEST}, R_{SRC}	$R_{DEST} <= R_{DEST}$ xor R_{SRC}	2 bytes

Notice that the jump instruction jl (jump if less than) depends on SF and OF, which are status flags. Status flags are set by the instruction preceding the jump, based on the result of the

computation. Some instructions, like the `cmp` instruction, perform a computation and set status flags, but do not return any result. The meanings of the status flags are given in the following table:

Name	Purpose	Condition Reported
OF	Overflow	Result exceeds positive or negative limit of number range
SF	Sign	Result is negative (less than zero)

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem 1.B RISC

Translate each of the x86 instructions in the following table into one or more RISC-V instructions. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed to translate each x86 instruction. Assume that upon entry, x1 contains `b`, x2 contains `a`, x3 contains `i`. x4 should receive `result`. If needed, use x5 as a condition register, and x6, x7, etc., for temporaries. You should not need to use any floating-point registers or instructions in your code. A description of the RISC-V instruction set architecture can be found in the class website, resources page.

x86 instruction	label	RISC-V instruction sequence
`xor %edx,%edx`		
`xor %ecx,%ecx`		
`cmp 0x8049580,%ecx`		
`jl L1`		
`jmp done`		
`add %eax,%edx`		
`inc %ecx` `jmp loop`		
...	`done:`	...

How many bytes is the RISC-V program using your direct translation? How many bytes of RISC-V instructions need to be fetched for b = 10 using your direct translation? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem 1.C Stack

In a stack architecture, all operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The hardware implementation we will assume for this problem set uses stack registers for the top two entries; accesses that involve other stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference. The table below gives a subset of a simple stack-style instruction set. Assume each opcode is a single byte. Labels, constants, and addresses require two bytes.

Example instruction	Meaning
PUSH A	push M[A] onto stack
POP A	pop stack and place popped value in M[A]
ADD	pop two values from the stack; ADD them; push result onto stack
SUB	pop two values from the stack; SUBtract top value from the 2nd; push result onto stack
ZERO	zeroes out the value at top of stack
INC	pop value from top of stack; increments value by one push new value back on the stack
BEQZ *label*	pop value from stack; if it's zero, continue at *label*; else, continue with next instruction
BNEZ *label*	pop value from stack; if it's not zero, continue at *label*; else, continue with next instruction
GOTO *label*	continue execution at location *label*

Translate the `multiplyByB` loop to the stack ISA. For uniformity, please use the same control flow as in parts a and b. Assume that when we reach the loop, `a` is the only thing on the stack. Assume `b` is still at address 0x8000 (to fit within a 2 byte address specifier).

How many bytes is your program? Using your stack translations from part (c), how many bytes of stack instructions need to be fetched for b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored? If you could push and pop to/from a four-entry register file rather than memory (the Java virtual machine does this), what would be the resulting number of bytes fetched and stored?

Problem 1.D Accumulator

In an accumulator ISA, one operand is implicitly a specific register (the same for all instructions), called the accumulator. For example, to execute C = A + B, the assembly code is:

Load A	Load A into the accumulator
Add B	Add B to the accumulator (which contains A's value)
Store C	Store the accumulator's value into the address contained in C

Notice that all instructions use the same accumulator. Also note that there are no registers in this example, but A, B, and C are all memory addresses. For this question, we will consider a modified architecture to make a solution possible. We will have accumulator instructions denoted by the postfix accumulator, and subtractor instructions which work the same way but reference a separate register. Therefore, a load can be either "load A accumulator" or "load A subtractor" depending on where the value of address A should be loaded. Using this ISA, and addition, subtraction, increment, decrement, zero, and branch instructions similar to the stack ISA, write the assembly for the program of this problem.

How many bytes is your program (assume same byte usage as the stack architecture and that the specifier for accumulator and subtractor requires no extra bytes)? Can the same program be implemented with just one accumulator (i.e., no subtractor)? If not, how would you extend this ISA to implement this program with just one accumulator?

Problem 1.E Conclusions

In just a few sentences, compare the four ISAs you have studied with respect to code size, number of instructions fetched, and data memory traffic. Which one would you choose if you were to build a specialized processor to execute the code in this program, and why?

Problem 1.F Optimization

To get more practice with RISC-V, optimize the code from part B so that it can be expressed in fewer instructions. There are solutions more efficient than simply translating each individual x86 instruction as you did in part B. Your solution should contain commented assembly code, a paragraph that explains your optimizations, and a short analysis of the savings you obtained.

Problem 2: Microprogramming and Bus-Based Architectures

In this problem, we explore microprogramming by writing microcode for the bus-based implementation of the RISC-V machine described in Handout #1 (Bus-Based RISC-V Implementation). Read the instruction fetch microcode in Table H1-3 of Handout #1. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

In order to further simplify this problem, *ignore* the busy signal, and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

Problem 2.A — Implementing Memory-to-Memory Add

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format:

ADDm r_d, r_s, r_t

ADDm performs the following operation:

$M[r_d] <= M[r_s] + M[r_t]$

Fill in Worksheet 2.A with the microcode for ADDm. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except r_d) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (i.e., by doing a microbranch to FETCH0 as discussed above).

You may want to consult the micro-code found in the micro-coded processor provided in Lab1, which can be viewed at ${LAB1ROOT}/src/rv32_ucode/micrcode.scala for guidance. <u>Warning</u>: While that micro-code passes all provided assembly tests and benchmarks, no guarantees to the optimality of the code can be assured, and there may still be bugs in the provided implementation.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Ex Sel	en Imm	uBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDM0:																

Worksheet 2.A

In this question we ask you to implement a useful string instruction, string copy (STRCPY). This instruction uses the same encoding as the other arithmetic instructions (R-type) on RISC-V (note rs2 should always be 00000 as there's only one source operand):

rd	rs1	rs2	func	opcode
5 bits	5 bits	5 bits	10 bits	7 bits

The STRCPY instruction provides the programmer the ability to copy a string directly from one location in memory (**M[Rs]**) to another location in memory (**M[Rd]**).

For this problem, think of a string as an array of 4-byte words, with the last element being zero (the string is "null terminated").

Starting from the memory location addressed by **Rs** (**M[Rs]**), keep copying one 4-byte word at a time to an other memory location, starting at the address **M[Rd]**, until you hit the null terminating character (zero). Do not forget to copy the null character too!

Finally, once STRCPY has finished, **Rd** and **Rs** will hold the address of the null character at the end of their respective strings.

The instruction definition for STRCPY requires that the strings pointed to by **Rs** and **Rd** do not overlap in memory.

Your task is to fill out Worksheet 2.B for STRCPY instruction. You should try to optimize your implementation for the minimal number of cycles necessary and for which signals can be set to don't-cares.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Imm Sel	en Imm	µBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	S	*
	PC <- A+4; dispatch	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
STRCPY:																

Worksheet 2.B

Problem 2.C Instruction Execution Times

How many cycles does it take to execute the following instructions in the microcoded RISC-V machine? Use the states and control points from RISC-V-Controller-2 in Lecture 2 (or Lab 1, in ${LAB1ROOT}/src/rv32_ucode/micrcode.scala) and assume Memory will not assert its busy signal.

Instruction	Cycles
`ADD  x3,x2,x1`	
`ADDI x2,x1,#4`	
`SW   x1,0(x2)`	
`BNE  x1,x2,label  #(x1 == x2)`	
`BNE  x1,x2,label  #(x1 != x2)`	
`BEQ  x1,x2,label  #(x1 == x2)`	
`BEQ  x1,x2,label  #(x1 != x2)`	
`J    label`	
`JAL  label`	
`JALR x1`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem 3: 6-Stage Pipeline

In this problem, we consider a modification to the fully bypassed 5-stage RISC-V processor pipeline presented in Lecture 4. Our new processor has a data cache with a two-cycle latency. To accommodate this cache, the memory stage is pipelined into two stages, M1 and M2, as shown in Figure 1-A. Additional bypasses are added to keep the pipeline fully bypassed.

Suppose we are implementing this 6-stage pipeline in a technology in which register file ports are inexpensive but bypasses are costly. We wish to reduce cost by removing some of the bypass paths, but without increasing CPI. The proposal is for all integer arithmetic instructions to write their results to the register file at the end of the Execute stage, rather than waiting until the Writeback stage. A second register file write port is added for this purpose. Remember that register file writes occur on each rising clock edge, and values can be read in the next clock cycle. The proposed change is shown in Figure 1-B.

In this problem, assume that the only exceptions that can occur in this pipeline are illegal opcodes (detected in the Decode stage) and invalid memory address (detected at the start of the **M2** stage). Additionally assume that the control logic is optimized to stall only when necessary. ***You may ignore branch and jump instructions in this problem.***

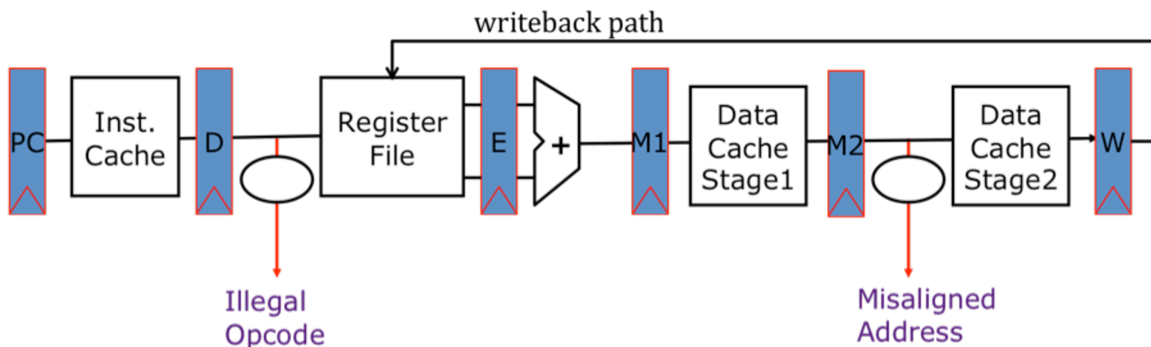


Figure 1-A. 6-stage pipeline. For clarity, bypass paths are not shown.

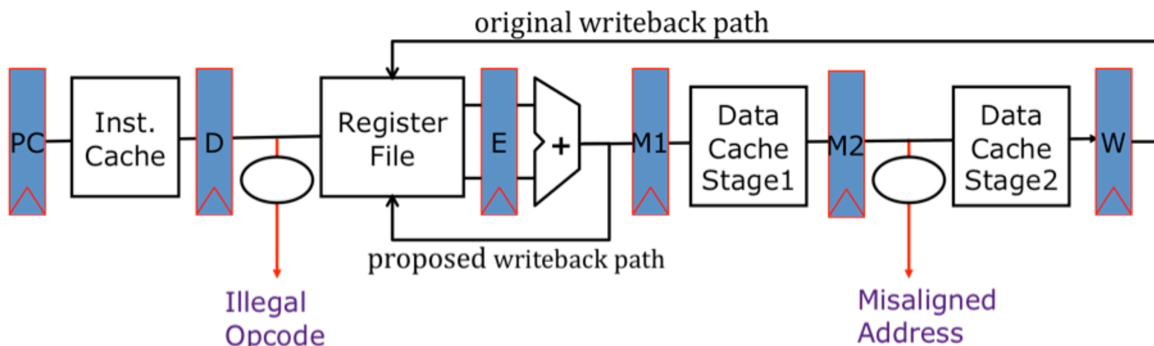


Figure 1-B. 6-stage pipeline with proposed additional write port.

Problem 3.A — Hazards: Second Write Port

The second write port allows some bypass paths to be removed without adding stalls in the decode stage. Explain how the second write port improves performance by eliminating such stalls *and* give a short code sequence that would have required an interlock to execute correctly with only a single write port and with the same bypass paths removed.

Problem 3.B — Hazards: Bypasses Removed

After the second write port is added, which bypass paths can be removed in this new pipeline without introducing additional stalls? List each removed bypass individually.

Problem 3.C — Precise Exceptions

Without further modifications, this pipeline may not support precise exceptions. Briefly explain why, and provide a minimal code sequence that will result in an imprecise exception.

Problem 3.D — Precise Exceptions: Implemented using a Interlock

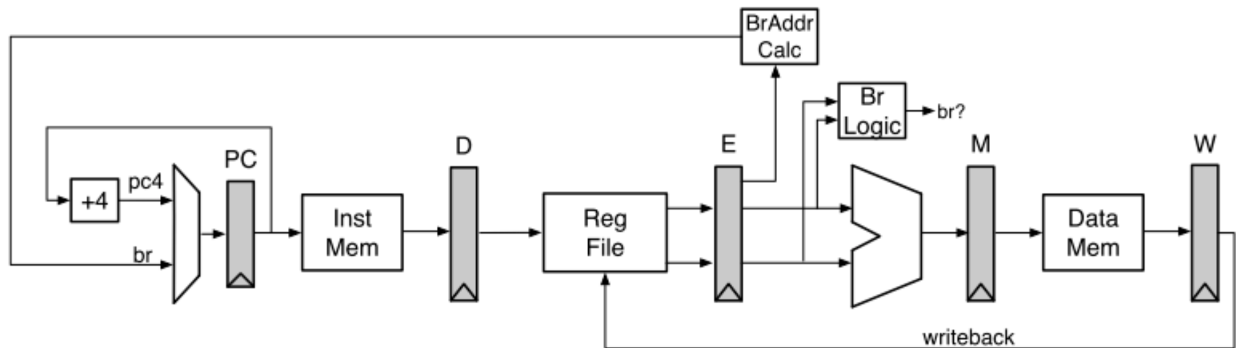
Describe how precise exceptions can be implemented by adding a new interlock. Provide a minimal code sequence that would engage this interlock. Qualitatively, what is the performance impact of this solution?

Problem 3.E — Precise Exceptions: Implemented using an Extra Read Port

Suppose you are additionally given the budget to add a new register file *read* port. Propose an alternative solution to implement precise exceptions in this pipeline without requiring any new interlocks.

Problem 4: Branch Speculation

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.



Problem 4.A — Motivating Branch Speculation

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 $\mathbf{t_{12}}$.

Hint: Throughout this question, make sure you also show instructions that were speculated to be executed and then flushed (it would help to mark them explicitly) in the instruction/time diagrams, as they also consume pipeline resources.

Chart 1: Using Standard Always Predict PC+4

PC	Instr	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}
0x2000	ADDI	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ													

Problem 4.B — Motivating Branch Speculation (2)

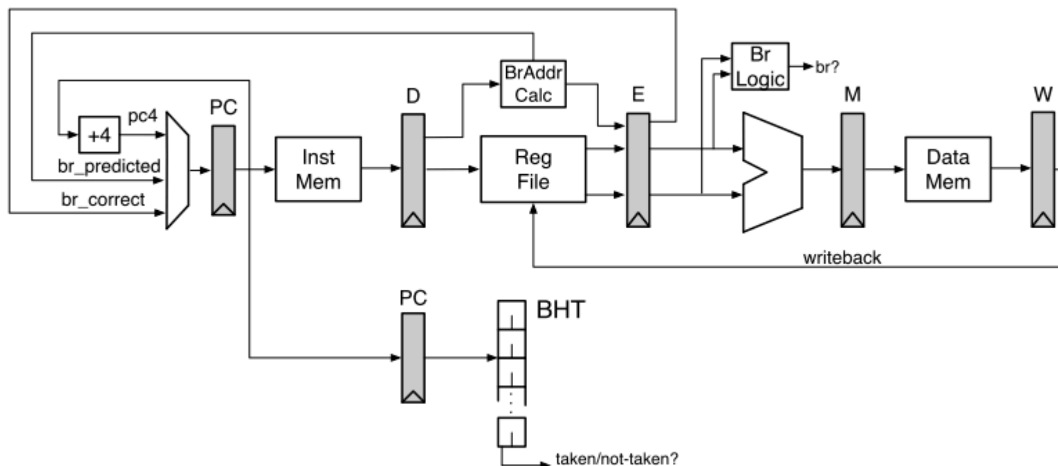
Fill in the following pipeline diagram (Chart 2), using the code segment below. Notice, the immediates used in the first two instructions (ADDI) are different from the previous question!

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

Chart 2: Using Standard Always Predict PC+4

PC	Instr	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}
0x2000	ADDI	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ													

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).

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
```

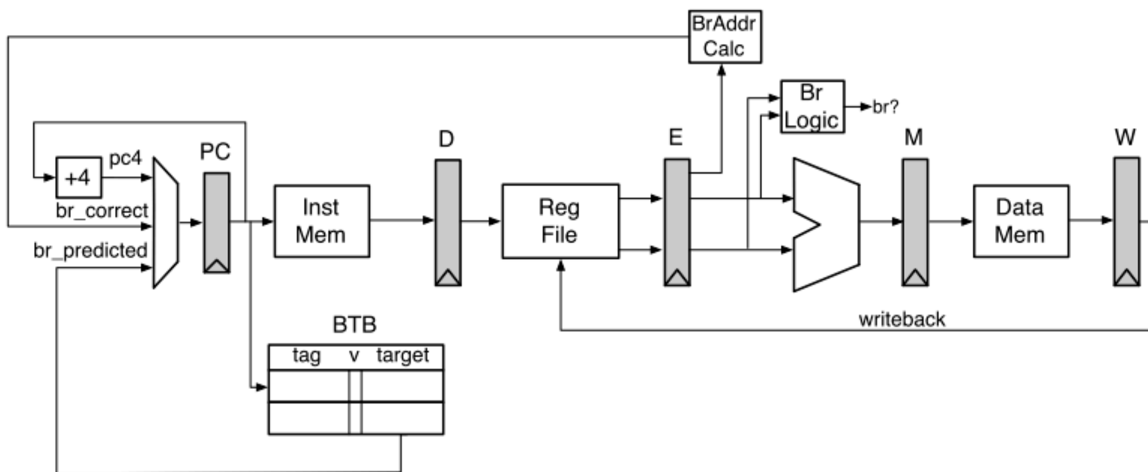
```
0x2000: LW   x7, 0(x6)
0x2004: ADDI x2, x2, 1
0x2008: BEQ  x2, x3, 0x2000
0x200c: SW   x7, 0(x6)
0x2010: OR   x5, x5, 4
0x2014: OR   x7, x7, 5
```

Chart 3: Using BHT

PC	Instr	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																		

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 to 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*).

Using the code segment below (the exact same code from **4.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.

```
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
```

Chart 4: Using BTB

PC	Instr	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																		

Problem 5: CISC vs RISC

For each of the following questions, circle either *CISC* or *RISC*, depending on which ISA you feel would be best suited for the situation described. Also, briefly *explain your reasoning*.

Problem 5.A **Lack of Good Compilers I**

Assume that compiler technology is poor, and therefore your users are far more apt to write all of their code in assembly. A _____ ISA would be best appreciated by these programmers.

CISC

RISC

Problem 5.B **Lack of Good Compilers II**

You desire to make compilers better at targeting your *yet-to-be-designed* machine. Therefore, you choose a _____ ISA, as it would be easiest for a compiler to target, thus allowing your users to write code in higher-level languages like C and Fortran and raise their productivity.

CISC

RISC

Problem 5.C — Fast Logic, Slow Memory

Assume that CPU logic is fast, *very* fast, while instruction fetch accesses are at least 10x slower (say, you're the lead architect of the "709"). Which ISA style do you choose as a best match for the hardware's limitations?

CISC

RISC

Problem 5.D — Higher Performance(?)

Starting with a clean slate in the year 2016 (area/logic/memory is cheap), you think that a _____ ISA that would lend itself best to a very high performance processor (e.g., high frequency, highly pipelined).

CISC

RISC

Problem 6: Iron Law of Processor Performance

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

For the final column "Overall Performance", mark whether the following modifications **increase**, **decrease**, have **no effect**, or whether the modification will have an **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the tradeoff in which it would be a beneficial modification or in which it would a detrimental modification (i.e., as an engineer would you suggest using the modification or not and why?).

		Instructions / Program	Cycles / Instruction	Seconds / Cycle	Overall Performance
a)	Adding a branch delay slot				
b)	Adding a complex instruction				
c)	Reduce number of registers in the ISA				
d)	Improving memory access speed				

e)	Adding 16-bit versions of the most common instructions in RISC-V (normally 32-bits in length) to the ISA (i.e., make RISC-V a variable length ISA)				
f)	For a given CISC ISA, changing the implementation of the micro-architecture from a microcoded engine to a RISC pipeline (with a CISC-to-RISC decoder on the front-end)				