

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Fall 2024

1	/16
2	/16
3	/16
4	/16
5	/18
6	/18

Quiz #2

<i>Name</i>	<i>Athena login name</i>	<i>Score</i>
<i>Recitation section</i>		
<input type="checkbox"/> WF 10, 34-301 (Varun)	<input type="checkbox"/> WF 2, 34-303 (Pleng)	<input type="checkbox"/> WF 12, 34-303 (Ezra)
<input type="checkbox"/> WF 11, 34-301 (Varun)	<input type="checkbox"/> WF 3, 34-303 (Pleng)	<input type="checkbox"/> WF 1, 34-303 (Ezra)
<input type="checkbox"/> WF 12, 34-302 (Keshav)	<input type="checkbox"/> WF 10, 34-302 (Hilary)	<input type="checkbox"/> WF 2, 34-302 (Jessica)
<input type="checkbox"/> WF 1, 34-302 (Keshav)	<input type="checkbox"/> WF 11, 34-302 (Hilary)	<input type="checkbox"/> WF 3, 34-302 (Jessica)
		<input type="checkbox"/> opt-out

Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for potential partial credit. You can use the extra white space and the back of each page for scratch work.

Problem 1. Sequential Circuits in Minispec (16 points)

Madeline has two modules Source and Sink she is trying to link together. Their interfaces are:

```
module Source;
    method Maybe#(Bit#(1)) out;
endmodule
module Sink;
    input Maybe#(Bit#(1)) in default = Invalid;
endmodule
```

There's a problem: Source internally interprets data in groups of 4 bits (a “nibble”). For example, to output 0x70 followed by 0x8F, Source would put 0111000010001111 on the output wire, in chronological order from left to right. Meanwhile, Sink **reverses the bits in each nibble** when it interprets the data. To express the same sequence—0x70 followed by 0x8F, Sink must receive 1110000000011111 in chronological order from left to right. Therefore, Madeline needs to add extra logic to reverse each nibble from Source before putting it into Sink. She decides to create a new module called Reverser, which has this interface:

```
module Reverser;
    input Maybe#(Bit#(1)) in default = Invalid;
    method Maybe#(Bit#(1)) out;
endmodule
```

This allows Madeline to specify the connection from Source to Sink by chaining them through Reverser in the middle:

```
module TopLevel;
    Source source;
    Sink sink;
    Reverser reverser;
    rule connect;
        reverser.in = source.out;
        sink.in = reverser.out;
    endrule
endmodule
```

Note Source may not provide an input on some of the cycles (i.e. provides Invalid), in which case, the reverser has to wait until it receives the full nibble. The Invalid gaps are not semantically significant. The reverser can output the reversed nibble without any Invalid gaps.

Madeline's partially-completed code for `Reverser` is on the next page. The design is as follows:

- There are two 4-bit buffers, named `sendBuf` and `recvBuf`. `sendBuf` can be `Invalid`.
- There are two 2-bit counters, `sendIdx` and `recvIdx`, to count how many bits have been sent and received.
- If `in` is valid, then the data bit is shifted-left into `recvBuf`, becoming the rightmost (least-significant) bit (as shown in the table of part A). When `recvBuf` becomes full (recognized through `recvIdx`), it is moved to `sendBuf`.
 - Copying in the same clock cycle is important to maintain the throughput and ensure latency is exactly 4 cycles (assuming no `Invalid` gaps in the inputs).
 - It doesn't matter what `recvBuf` becomes at this point, so the code does the same thing as usual to keep the circuit simple (It shifts-left the input).
- On each cycle, if `sendBuf` is valid, the rightmost bit is output through a method and shifted out. If all bits have been shifted out, then `sendBuf` is reset to `Invalid`, unless there is data to be copied from `recvBuf` in the same clock cycle.
 - Notice that the buffers are used somewhat like a stack, so the nibble gets transmitted in reverse order as intended!

The following is the (vertical) pipeline diagram of Madeline's module. The pipeline diagram has been filled in for the first 6 cycles. `Valid` and `Invalid` have been abbreviated as `V` and `Inv` respectively. This specifies the expected behavior of Madeline's module. Note `recvBuf` was initialized to a don't-care value due to the use of `RegU` type; assume it is initialized to `4'b1111`.

(A) (6 points) What are the values on cycle 9 and 11? Fill the table where indicated. The rest of the table will not be graded; it can be used for scratch work.

Cycle	in	recvBuf	recvIdx	sendBuf	sendIdx	out
0	V(0)	4'b1111	0	Inv	0	Inv
1	V(1)	4'b1110	1	Inv	0	Inv
2	Inv	4'b1101	2	Inv	0	Inv
3	V(1)	4'b1101	2	Inv	0	Inv
4	V(1)	4'b1011	3	Inv	0	Inv
5	V(0)	4'b0111	0	V(4'b0111)	0	V(1)
6	V(1)					
7	V(0)					
8	Inv					
9	V(0)					
10	V(1)					
11	V(1)					

(B) (10 points) **Implement the out method**, and fill in the blanks in the rule so the code correctly implements the desired behavior.

```

module Reverser;
  Reg#(Maybe#(Bit#(4))) sendBuf(Invalid);
  RegU#(Bit#(4)) recvBuf;
  Reg#(Bit#(2)) sendIdx(0);
  Reg#(Bit#(2)) recvIdx(0);
  input Maybe#(Bit#(1)) in default = Invalid;
  // return bit 0 of sendBuf if valid
  method Maybe#(Bit#(1)) out;
  // TODO: implement this method

endmethod
rule connect;
  let nSendBuf = sendBuf;
  if (isValid(sendBuf)) begin
    sendIdx <= sendIdx+1;
    // shift out the sent bit, or mark invalid if all 4 bits done

    Bit#(4) sendBuf_v = _____;

    nSendBuf = _____ ? _____ : Valid(sendBuf_v>>1);
  end
  let nRecvBuf = recvBuf;
  // shift-left a bit into the receive buffer
  if (isValid(in)) begin
    recvIdx <= recvIdx+1;

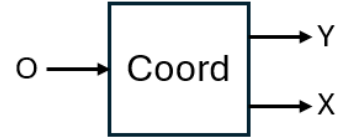
    nRecvBuf = {_____, _____};
    // copy to sendBuf immediately if received all 4 bits

    if (recvIdx == _____) nSendBuf = _____;
  end
  sendBuf <= nSendBuf;
  recvBuf <= nRecvBuf;
endrule
endmodule

```

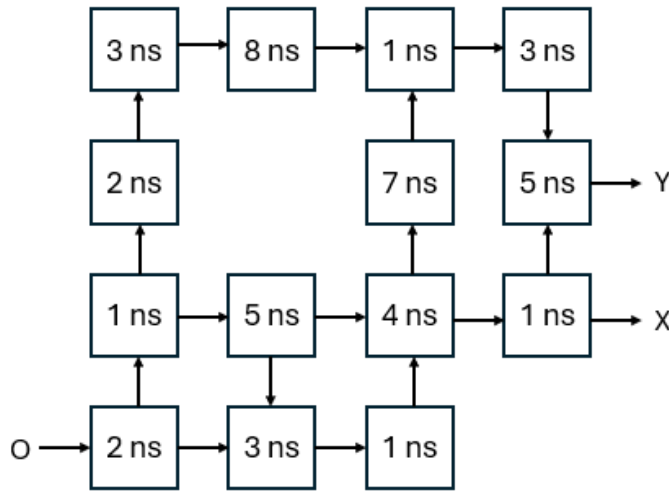
Problem 2. Arithmetic Pipelines (16 points)

Kurt E. Zian comes to you with a module named “Coord.” This module has one input, **O**, and two outputs, **X** and **Y**. He tells you that the outputs **X** & **Y** help him find the coordinates of bugs on his bed, but its throughput is too low for bug catching. You recently learnt about pipelining in class, and so you decide to help him.



For each of the questions below, please create a valid **K-stage pipeline** of the given circuit. Each component in the circuit is annotated with its propagation delay in nanoseconds. **Show your pipelining contours** and place large black circles (●) on the signal arrows to **indicate the placement of pipeline registers**. Give the latency and throughput of each design, assuming ideal registers ($t_{PD}=0$, $t_{SETUP}=0$). Remember that our convention is to place a pipeline register on each output. **Note that invalid pipeline diagrams will receive 0 points.** **Pay close attention to the direction of the arrows.**

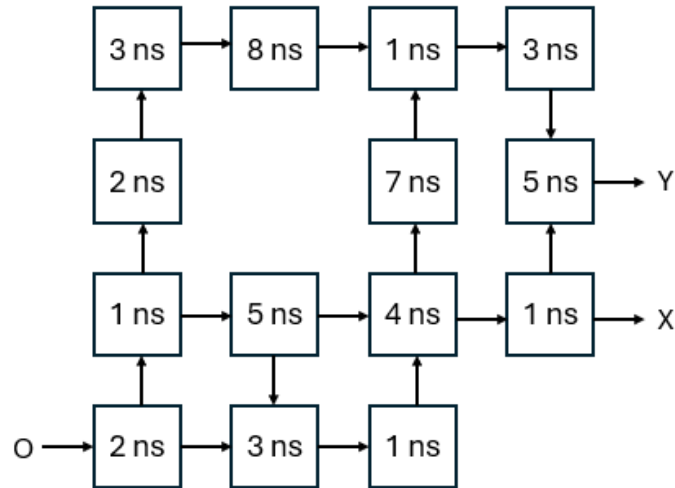
(A) (5 points) Show a **maximum-throughput 3-stage pipeline** using a minimal number of registers. **Invalid pipelines will earn 0 points.** What are the latency and throughput of the resulting circuit? *In case you need them, extra copies of the circuit are available at the end of the exam.*



Latency (ns): _____

Throughput (ns⁻¹): _____

(B) (6 points) Show a **maximum-throughput pipeline** using a minimal number of registers.
Invalid pipelines will earn 0 points. What are the latency and throughput of the resulting circuit? *Extra copies of the circuit are provided at the end of the exam.*



Latency (ns): _____

Throughput (ns⁻¹): _____

(C) Kurt is excited! He now wants to take the X and Y from the Coord module and pass it in as inputs to an add-on module. For this add-on module, he is considering three different models, DEBUG, CATCH, and SWAT, which have the same functionality, but differ in throughput and number of pipeline stages (given in the table below).

Add-on Module	Throughput (ns^{-1})	Pipeline Stages
DEBUG	1/15	1
CATCH	1/10	2
SWAT	1/5	6

- (i) (3 points) Kurt wants to **maximize throughput**. Which versions of the pipelined Coord module and add-on module should Kurt choose, and what are the resulting latency and throughput? If two combinations have identical throughput, choose the one with better latency.

Module Coord (circle one):

3-stage pipeline Maximum-throughput pipeline

Add-on Module (circle one):

DEBUG (T = 1/15) CATCH (T = 1/10) SWAT (T = 1/5)

Total Latency (ns): _____

Throughput (ns^{-1}): _____

- (ii) (2 points) Oh no! Kurt is worried that his machine is taking too long, and now wants to **minimize latency**. Which versions of the pipelined Coord module and add-on module should Kurt choose, and what are the resulting latency and throughput? If two combinations have identical latency, choose the one with better throughput.

Module Coord (circle one):

3-stage pipeline Maximum-throughput pipeline

Add-on Module (circle one):

DEBUG (T = 1/15) CATCH (T = 1/10) SWAT (T = 1/5)

Total Latency (ns): _____

Throughput (ns^{-1}): _____

Problem 3. Processor Implementation (16 points)

Reggie Ster has written a program in RISC-V assembly that repeatedly performs the following pair of instructions.

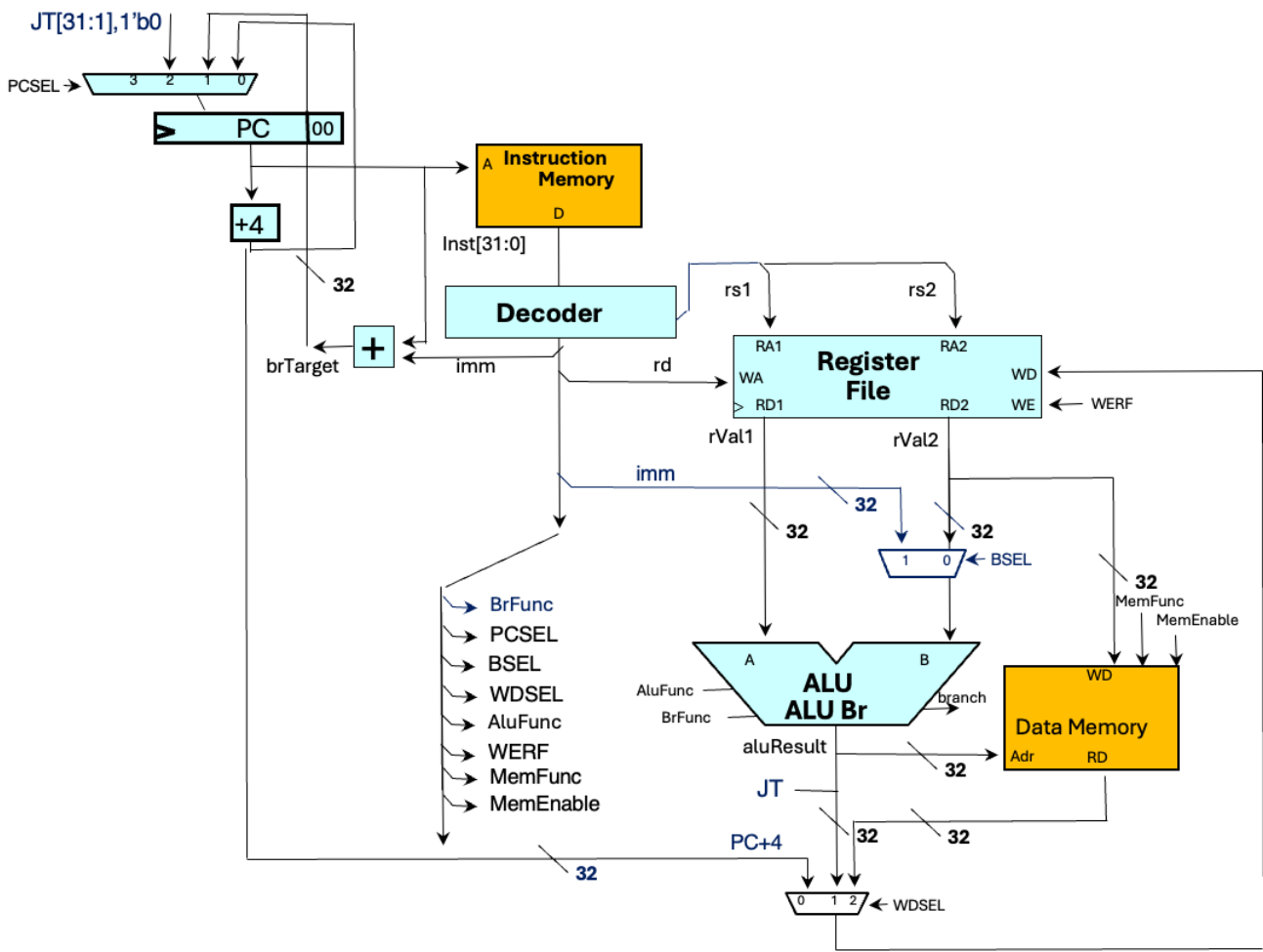
```
add t0, sp, a0
sw a0, 0(t0)
```

Reggie wants to reduce the number of instructions in his program by combining these two into one instruction. He comes up with a **new Add Address and Store instruction**, and wants to replace the add and sw instructions above with the aas instruction below:

```
aas rd, rs1, rs2
```

```
reg[rd] = reg[rs1] + reg[rs2]
Mem[reg[rs1] + reg[rs2]] = reg[rs2]
```

Reggie hopes that the general processor implementation from lecture (shown below) can implement his new instruction.



(A) (4 points) Without **modifying the processor implementation** on the previous page, fill in the table below with what the decoder should output for the `aas rd, rs1, rs2` instruction. Write “?” for don’t care values.

AluFunc: Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra

BrFunc: Eq, Neq, Lt, Ltu, Ge, Geu

MemFunc: Lw, Lh, Lhu, Lb, Lbu, Sw, Sh, Sb

aas rd, rs1, rs2	Field	Value
	imm	
	AluFunc	
	BrFunc	
	BSEL	
	MemFunc	
	MemEnable	
	WDSEL	
	WERF	
	PCSEL	

(B) (1 point) What `aas` instruction should Reggie write to replace his frequently used instruction pair (copied below from previous page)?

```
add t0, sp, a0
sw a0, 0(t0)
```

Instruction: _____

Reggie is looking to speed up his program even more, and spots a similar pair of instructions that are also repeated many times:

```
add a0, a0, t0
sw a0, 0(t0)
```

He wants to combine these two instructions into a **new Add Value and Store instruction** with the syntax and execution described below.

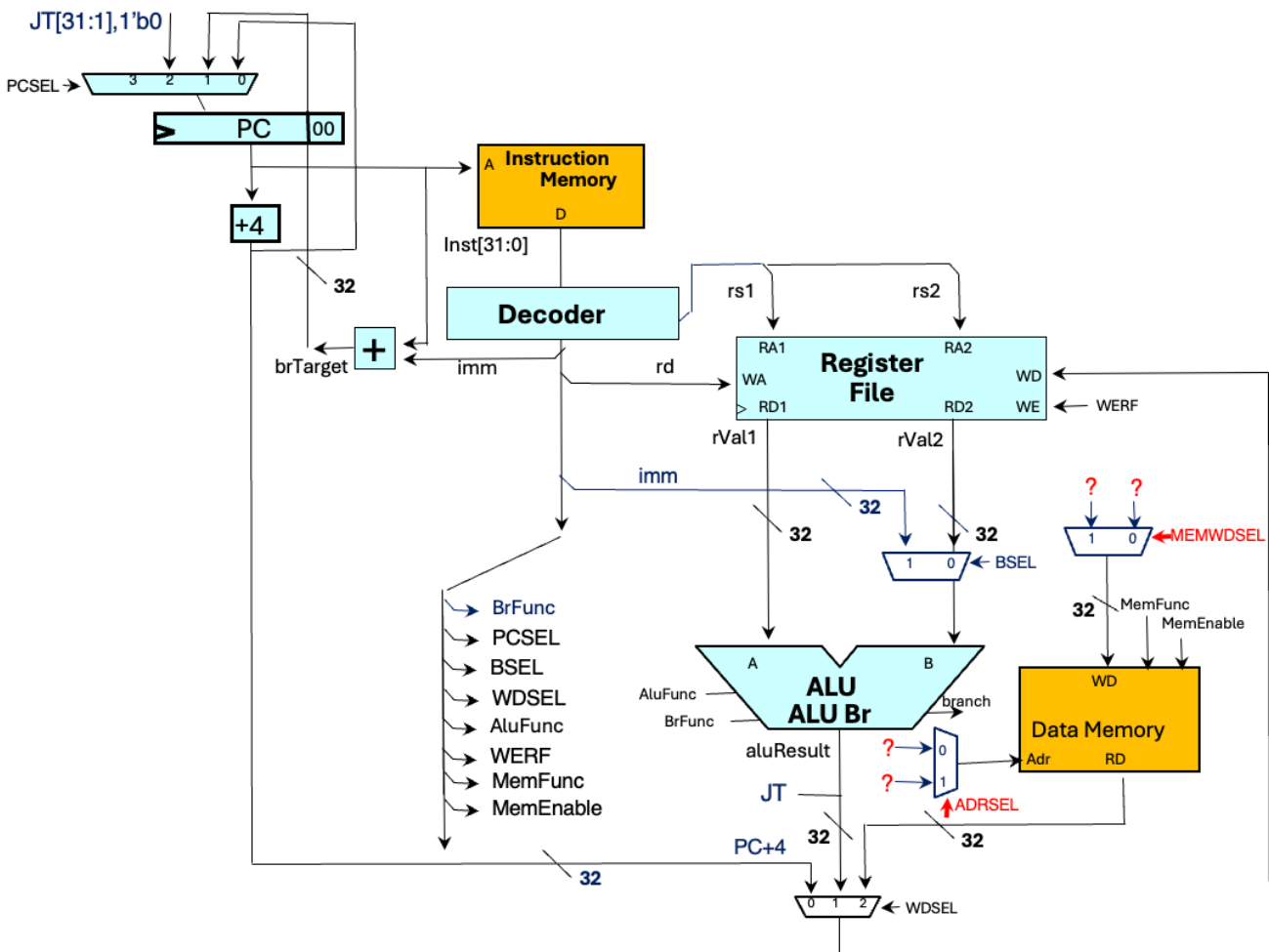
```
avs rd, rs1, rs2
```

```
reg[rd] = reg[rs1] + reg[rs2]
Mem[reg[rs2]] = reg[rs1] + reg[rs2]
```

(C) (2 points) The `avs` instruction that combines the functionality of the `add a0, a0, t0` and the `sw a0, 0(t0)` instructions cannot be implemented on the current processor. Explain why not.

Explanation:

Reggie really likes the idea of an `avs` instruction and decides to modify the processor to implement it. He needs your help to set the correct signals! For the rest of the problem, use the updated processor diagram shown below, with Reggie's proposed modifications in red.



(D) (2 points) He decides to add a signal `ADRSEL`, whose mux output connects to the `Adr` port of Data Memory. He wants the decoder to set `ADRSEL = 1` for the `avs` instruction and `ADRSEL = 0` for all other instructions. What existing signals should the `ADRSEL` mux inputs be connected to?

ADRSEL input 0 signal: _____

ADRSEL input 1 signal: _____

(E) (2 points) He also adds a signal `MEMWDSEL`, whose mux output connects to the `WD` port of Data Memory. He wants this signal to be 1 for the `avs` instruction and 0 for all other instructions. What existing signals should the `MEMWDSEL` mux inputs be connected to?

MEMWDSEL input 0 signal: _____

MEMWDSEL input 1 signal: _____

(F) (2 points) For each of the following signals, determine whether the mux being controlled by that signal needs an extra input to accommodate the new instruction. If so, indicate the name or value of **the signal that needs to be added as an input to the mux**. If not, indicate which existing value of **the mux control signal** (i.e., 0, 1, 2) is required to make the instruction work properly.

BSEL: Needs new input? **YES NO**

New input/Existing control signal: _____

WDSEL: Needs new input? **YES NO**

New input/Existing control signal: _____

(G) (3 points) Reggie is very excited about his new instruction! He wonders if he can store `reg[rs1] + reg[rs2]` in memory at the value of any register, not just `rs2`. For instance, he wants to calculate the result of adding `a0` and `a1`, write that to the `ra` register, and store it on the stack, like the two instructions below.

```
add ra, a0, a1
sw ra, 0(sp)
```

Can the processor be modified further to support this? Note that **only the processor's control signals can be modified**. If yes, explain the changes you would make to the processor, especially the inputs and outputs to control mux signals. If no, explain why the processor cannot implement this execution in one instruction.

Can we implement the code above in one instruction? Circle one: **YES NO**

Explanation:

Problem 4: Caches (16 points)

The two parts of this problem are independent of each other.

(A) (6 points) Consider the following 2-way set associative cache.

Way 1						Way 0						
LRU		V	D	tag	Word 1	Word 0	V	D	tag	Word 1	Word 0	
0	0	1	0	0x20	0x0123	0x1234	0	0	1	0x03	0x2345	0x3456
1	1	0	1	0x43	0x4567	0x5678	1	1	1	0x28	0x6789	0x789A
1	2	1	1	0x30	0x29AB	0x9ABC	2	1	0	0x21	0x1BCD	0x2CDE
0	3	1	0	0x41	0x3DEF	0x0EF0	3	1	0	0x63	0x4F01	0x5012

For each of the following memory accesses, determine if it results in a hit or miss. If it is a hit, specify what data is returned. *Enter NA in data returned column for a miss.* If it is a miss, does any data need to be written back to main memory? If so, provide the addresses that need to be updated in main memory. If no write back to memory is required, then write NONE. *If the access was a hit then write NA in the addresses to update column.* **Consider each request independently assuming that initial cache state is as shown above. Note that there are 1w and 1b requests below.**

Instruction	Data Returned If hit, what data is returned? If miss, enter NA.	Addresses to Update If hit, enter NA. If miss, list all addresses that need to be updated in main memory, or enter NONE if no updates are necessary.
lw x1, 0x430(x0)		
lw x1, 0x30(x0)		
lb x1, 0x83D(x0)		

(B) (10 points) The following code accesses two 100 element arrays A and B and swaps their elements. Array A begins at memory location 0x400, and array B begins at memory location 0x800.

```
    addi x1, x0, 100      // array size (x1) initialized to 100
    addi x2, x0, 0        // array index (x2) initialized to 0

    . = 0x100            // loop code begins at address 0x100
loop: slli x3, x2, 2      // compute offset into array
    lw x4, 0x400(x3)
    lw x5, 0x800(x3)
    sw x4, 0x800(x3)
    sw x5, 0x400(x3)
    addi x2, x2, 1        // go to next element
    bne x2, x1, loop
```

Consider this code once the loop has been running for many iterations. Compute the steady state hit rate of executing this loop on the following two cache configurations, each of which can hold a total of 16 words.

Cache A: A 4-way set associative cache with a block size of 2 and an LRU replacement strategy.

Cache B: A fully associative cache with a block size of 4 and an LRU replacement strategy.

We have provided diagrams of caches A and B on the following page. You may use them if you find them helpful, but you do not need to fill them out.

Steady state instruction hit rate of cache A: _____

Steady state data hit rate of cache A: _____

Steady state instruction hit rate of cache B: _____

Steady state data hit rate of cache B: _____

Part B code repeated for convenience:

```

    addi x1, x0, 100      // array size (x1) initialized to 100
    addi x2, x0, 0       // array index (x2) initialized to 0

    . = 0x100           // loop code begins at address 0x100
loop: slli x3, x2, 2     // compute offset into array
    lw x4, 0x400(x3)
    lw x5, 0x800(x3)
    sw x4, 0x800(x3)
    sw x5, 0x400(x3)
    addi x2, x2, 1      // go to next element
    bne x2, x1, loop

```

Cache A: 4-way set associative with block size of 2

Way 0				Way 1			
Index	Tag	Word 1	Word 0	Index	Tag	Word 1	Word 0
0				0			
1				1			

Way 2				Way 3			
Index	Tag	Word 1	Word 0	Index	Tag	Word 1	Word 0
0				0			
1				1			

Cache B: Fully associative with block size of 4

Tag	Word 3	Word 2	Word 1	Word 0

Problem 5. Pipelined Processors (18 points)

Alyssa P. Hacker has hacked into Ben Bitdiddle’s computer! Though at the moment, the only thing Alyssa can do is look at what instruction is currently in each stage of Ben’s processor. However, Alyssa has plans to use this to extract Ben’s password.

Consider the following code, which checks if the value of input x, in register a0, matches the value at memory address 0x234

```

lw a1, 0x234(x0)      // load secret into a1
li a2, 1              // initialize match to True
li a5, 0
loop:
  andi a3, a1, 1      // get last bit of secret
  andi a4, a0, 1      // get last bit of x
  beq a3, a4, skip    // if x[0] != secret[0], then match is False
  li a2, 0
  addi a5, a5, 1
skip:
  srli a1, a1, 1      // right shift secret by 1
  srli a0, a0, 1      // right shift x by 1
  bnez a1, loop
end:
  mv a0, a2           // move match to a0
  ret

. = 0x234
.word ???             // super secret password

```

Alyssa knows that Ben’s processor uses a 4-stage pipeline (IF, DEC, EXE/MEM, WB). In this pipeline:

- The EXE and MEM stages have been merged into one pipeline stage.
- The result of a lw operation is available at the beginning of the WB stage.
- Branches are predicted **not taken**.
- The processor has hardware to resolve branches and jumps in the **DEC** stage.
- Full bypassing is implemented.

(A) (8 points) Fill in the pipeline diagram below assuming that the loop has been running for a while. Cycle 0 begins a new loop iteration by fetching the `andi a3, a1, 1` instruction. Assume that the `beq` and `bnez` branches **are both taken** in this iteration of the loop. **Draw arrows indicating each use of bypassing.** Ignore cells shaded in gray.

	0	1	2	3	4	5	6	7	8	9	10
IF	andi										
DEC											
EXE											
WB											

(B) (3 points) In steady state, how many cycles does the loop take when the `beq` branch is **taken**? How many cycles are wasted due to stalls? How many are wasted due to annulments?

Number of cycles per loop iteration: _____

Number of cycles per loop iteration wasted due to stalls: _____

Number of cycles per loop iteration wasted due to annulments: _____

(C) (6 points) Now, let's see what happens when the `beq a3, a4, skip` branch is **not taken** (indicating a mismatch). How many cycles does the loop take when the `beq` branch is not taken? How many cycles are wasted due to stalls? How many are wasted due to annulments?

To answer these questions, you may find it helpful to fill in the following pipeline diagram, but you will only be graded on your numeric answers below the pipeline diagram. Assume that the loop has been running for a while and in cycle 0 a new loop iteration begins by fetching the `andi a3, a1, 1` instruction. Recall that here the `beq a3, a4, skip` branch is not taken. Assume that the `bnez` branch is still taken to repeat the loop.

	0	1	2	3	4	5	6	7	8	9	10
IF	andi										
DEC											
EXE											
WB											

Number of cycles per loop iteration: _____

Number of cycles per loop iteration wasted due to stalls: _____

Number of cycles per loop iteration wasted due to annulments: _____

(D) (1 points) Alyssa tries guessing 0x0 and 0x1, and notices that 0x1 took 0.1ps longer to finish executing than 0x0 did. What is the last bit of secret?

Last bit of secret: _____

Problem 6. Pipelined Processor Performance (18 points)

In this problem, you will explore the two different processor designs shown in Figures 1 and 2 in parts (A) and (B). The processor in Figure 1 is a fully bypassed five-stage RISC-V processor. The processor in Figure 2 moves branch resolution and jump target resolution from the Execute stage into the Decode stage by adding a dedicated branch comparator and adder after the bypass muxes. Both processors use **magic (combinational read, clocked write) instruction and data memories that return the result of a load in the same cycle it was requested.**

Table 1 contains the delay of the various components in the datapath.

We will examine how these two processors execute the following function written in assembly. The function compares each element in an array of integers to two search values. If a value is found, then the function returns the index of the value. If neither of the two search values are found, then the function returns -1. Assume the following initial register values: **t4** initially holds the pointer to the array of integers; **t5** holds the size of the array; **t6** and **t3** hold the two search values. **t2** holds the return value. Assume that **t5** is initially 64 and that none of the search values are actually present in the array (i.e., the loop executes 64 times).

```

0x1000    addi t0, zero, 0
        loop:
0x1004    lw   t1, 0(t4)
           // check value 1
0x1008    bne t1, t6, L1
0x100c    j    done
        L1:
           // check value 2
0x1010    bne t1, t3, L2
0x1014    j    done
        L2:
0x1018    addi t4, t4, 4
0x101c    addi t0, t0, 1
0x1020    bne t0, t5, loop
0x1024    addi t2, zero, -1
0x1028    j    end
        done:
0x102c    addi t2, t0, 0
0x1030    j    end
        end:

```

Component	Delay (ns)
Register read/write	1
Register File read/write	10
Memory read/write	20
+4 unit	4
Decode	5
Br cmp / Adder	8
Mux	3
Execute	20

Table 1: Datapath component delays.

(A)(10 points) Performance when Resolving Branch in EXE Stage

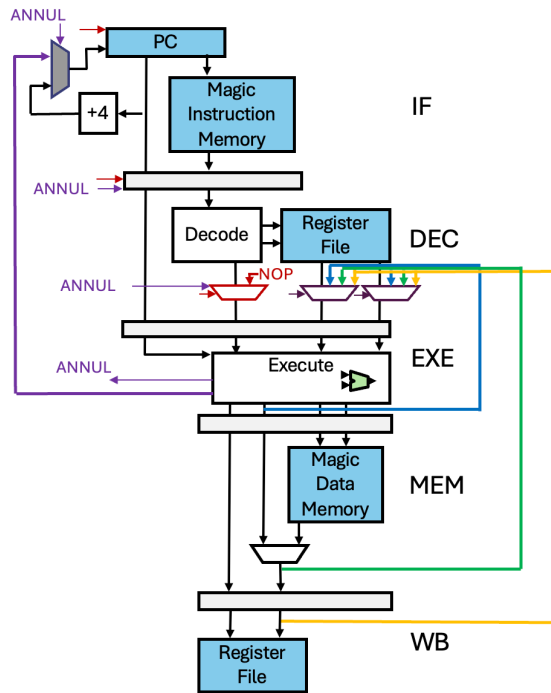


Figure 1

- i. Given the datapath component delays in Table 1, identify and **highlight the critical path in Figure 1**. Then compute the cycle time and enter it in **Table 2** on the next page.
- ii. Use the following pipeline diagram to illustrate how the first iteration of the loop executes on the processor in Figure 1, a fully bypassed five-stage RISC-V pipelined processor with magic memories. Assume that the `lw` instruction is fetched in cycle 1. Remember that branches are resolved in the EXE stage and incorrectly fetched instructions are annulled (**even if the same instruction will get fetched again**). Assume that the branch target and jump target computations are also performed in the EXE stage. You must account for all bypass paths used but you do not need to draw in the bypass arrows in the pipeline diagram. **Make sure to include the fetch of `lw` on the second iteration. You do not need to fill in any columns beyond the fetch of that second `lw`.**

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	lw															
DEC																
EXE																
MEM																
WB																

- iii. Based on this pipeline diagram, compute the execution time for one iteration of this loop. Fill in the appropriate row of Table 2. You must show your work.

Part	Branch Resolution	Useful Instructions/ Loop	Avg Cycles/ Instruction	Time (ns) / Cycle	Time (ns) / Loop
A	EXE				
B	DEC				

Table 2: Performance comparison between datapaths of Figure 1 and Figure 2.

(B) (8 points) Performance when Resolve Branch in DEC Stage

- i. Given the datapath component delays in Table 1, identify and highlight the critical path in Figure 2 below. Then compute the cycle time and enter it in Table 2 above. This processor is identical to the processor in Figure 1 except for one key difference: the branch is resolved in the DEC stage and the branch and jump target addresses are computed in the DEC stage as well.

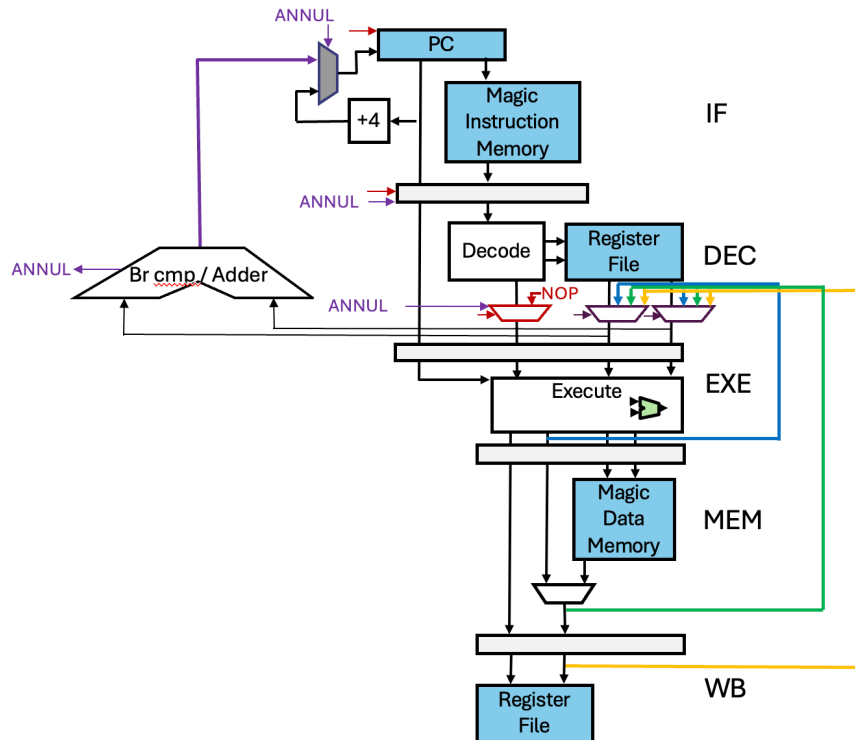


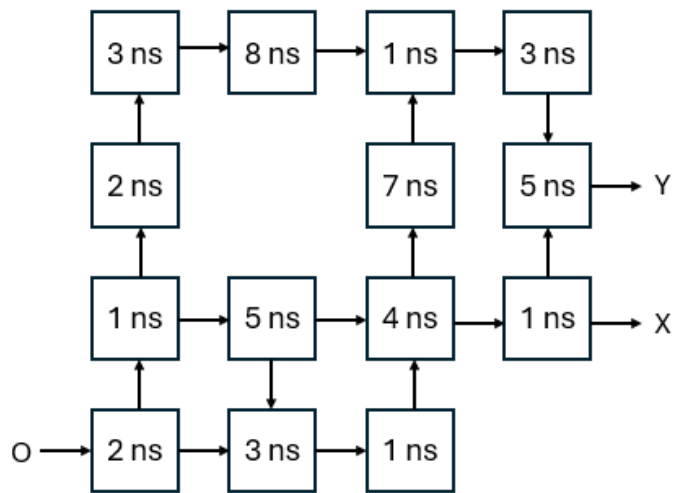
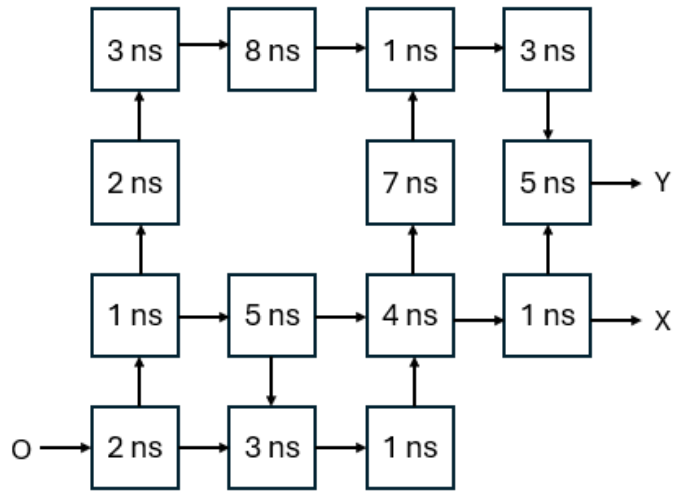
Figure 2

- ii. Use the following pipeline diagram to illustrate how the first iteration of the loop executes on the processor in Figure 2. Assume that the `lw` instruction is fetched in cycle 1. Recall that in this processor branches are resolved in the DEC stage and incorrectly fetched instructions are annulled (even if the same instruction will get fetched again). In addition, branch target and jump target computations are also performed in the DEC stage. This is achieved by adding a dedicated branch comparison and adder unit to the decode stage. You must account for all bypass paths used but you do not need to draw in the bypass arrows in the pipeline diagram. **Make sure to include the fetch of `lw` on the second iteration. You do not need to fill in any columns beyond the fetch of that second `lw`.**

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	lw															
DEC																
EXE																
MEM																
WB																

- iii. Based on this pipeline diagram, **compute the execution time for one iteration of this loop. Fill in the appropriate row of Table 2. You must show your work.**

Extra pipeline diagrams for problem 2:



Extra pipeline diagrams for problem 6:

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	lw															
DEC																
EXE																
MEM																
WB																

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	lw															
DEC																
EXE																
MEM																
WB																

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	lw															
DEC																
EXE																
MEM																
WB																

END OF QUIZ 2!