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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures

Fall 2025

Quiz #2

Name	*Athena login name*	*Score*

Recitation section

□ WF 10, 34-301 (Jan)	□ WF 2, 34-302 (Varun)	□ WF 12, 34-303 (Nathan)
□ WF 11, 34-301 (Jan)	□ WF 3, 34-302 (Varun)	□ WF 1, 34-303 (Nathan)
□ WF 12, 34-302 (Abdullah)	□ WF 10, 34-302 (Christina)	□ WF 2, 34-303 (Grace)
□ WF 1, 34-302 (Abdullah)	□ WF 11, 34-302 (Christina)	□ WF 3, 34-303 (Grace)
		□ 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 (Error-Corrected Communication) (18 points)

Our friend Ben Bitdiddle is working for a communications company. He's tasked with implementing a (Universal Asynchronous Receiver/Transmitter) protocol in Minispec. The protocol transmits data serially one bit at a time, along with start and stop bits to indicate the beginning and end of a data packet. All of this data can be transmitted on a single `tx` (transmission) wire for one-way communication between a sender and receiver module. A standard frame consists of:

- 1 Start Bit (always 0)
- 8 Data Bits (LSB first)
- 1 Parity Bit (parity of the count of 1's in the data bits, high if odd, low if even)
- 1 Stop Bit (always 1)

(A) (8 points) Fill in the following skeleton for the `Transmit` module to send data frames according to the specification above. Assume that the module begins in the `Idle` state and transmits a 1 every cycle (default value for `tx` is 1) until it receives valid input data at which point it begins transmission of a standard frame beginning with a Start Bit of 0. Each bit should be transmitted for exactly 1 clock cycle. Assume that the **data and parity bits** are handled in the Sending state. Also, assume that any input received when the module is not in the `Idle` state is ignored.

```
typedef enum {
  Idle,
  Sending,
  Stop
} TransmitState;

module Transmit;
  Reg#(Bit#(8)) currData;
  Reg#(TransmitState) state(Idle);
  Reg#(Bit#(1)) tx(1);
  Reg#(Bit#(4)) idx;
  Reg#(Bit#(1)) parity;

  input Maybe#(Bit#(8)) inputData default = Invalid;

  rule transmit;
    case (state)
      Idle: begin
        if (______________________________) begin
          ______________________________ <= ______________________________;
          ______________________________ <= 0;
          ______________________________ <= 0;
          state <= ______________________________;
          idx <= ______________________________;
        end
      end
      Sending: begin
        if (idx < ____________) begin
          tx <= ______________________________;
          parity <= ______________________________ ^ ______________________________;
          idx <= ______________________________;
        end
        else begin
          tx <= ______________________________;
          state <= Stop;
        end
      end
      Stop: begin
        ______________________________ <= ______________________________;
        state <= ______________________________;
      end
    endcase
  endrule

  method Bit#(1) txWire;
    return ______________________________;
  endmethod
endmodule
```

(B) (3 points) Fill in the following table to show the values of the internal registers of the `Transmit` module over 13 clock cycles given the `inputData` values shown below. Assume that all registers start with undefined values (denoted by `?`) unless they have an initial value.

Note: `Inv` corresponds to minispec `Invalid` keyword and `V(x)` corresponds to minispec `Valid(x)` keyword.

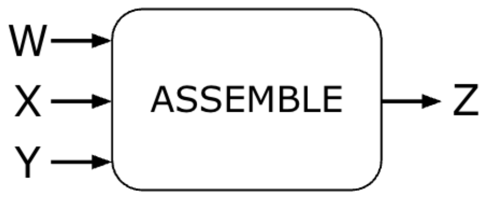
Cycle	0	1	2	3	...	8	9	10	11	12
inputData	V(0xA7)	Inv	Inv	Inv	...	Inv	V(0x11)	Inv	Inv	Inv
currData					...					
state					...					
tx					...					
idx					...					

(C) (7 points) Now fill in the following `Receive` module. Assume this will always be connected to a `tx` wire coming from a proper `Transmit` module and keep in mind the ordering of the incoming bits. Assume that the module remains in the `Idle` state until it sees a 0 which it interprets as a valid Start bit and moves to the Receiving state to begin processing the data and parity. The `receivedByte` method should return `Invalid` unless the Receive module just finished receiving an entire **valid** message. **Any data frames that do not exactly match the specification above (including the start, stop, and parity bits) are considered invalid by the receiver.**

```
typedef enum {
  Idle,
  Receiving,
  Stop
} ReceiveState;


module Receive;
  // Collects a single-byte message and only returns valid data
  // if the parity of the received data and the expectedParity bits match.
  // Expect the data to be coming in with the least-significant bit first
  // and ensure that the output is returned in the correct order.
  Reg#(ReceiveState) state(Idle);
  Reg#(Bit#(1)) parity, expectedParity;
  Reg#(Bit#(4)) idx;
  Reg#(Bit#(8)) data;

  input Bit#(1) txWire;
  rule receive;
    case (state)
      Idle: begin
        if (______________________________) begin
          state <= Receiving;
          // initialization logic
          expectedParity <= ______________________________________________;
          idx <= ______________________________________________;
          data <= ______________________________________________;
        end
      end
      Receiving: begin
        if (idx == ______________________________) begin
          state <= Stop;
          parity <= ______________________________________________;
        end else begin
          data[idx] <= ______________________________________________;
          expectedParity <= ______________________________________________;
        end
        idx <= idx + 1;
      end
      Stop: begin
        state <= ______________________________;
      end
    endcase
  endrule

  method Maybe#(Bit#(8)) receivedByte;
  return (state == _______________)
      && (_______________________________)
      && (_______________________________) ?
        Valid(data) : Invalid;
  endmethod
endmodule
```

Problem 2. Arithmetic Pipelines (16 points)

Papa Louie is almost finished creating his pizza-making module, but he needs your help implementing the last submodule called *ASSEMBLE*.
This submodule takes in three inputs: **W**, **X**, and **Y** and has one output: **Z**. Papa Louie tells you that *ASSEMBLE* works, but the throughput is too low. He asks you to help pipeline the submodule.



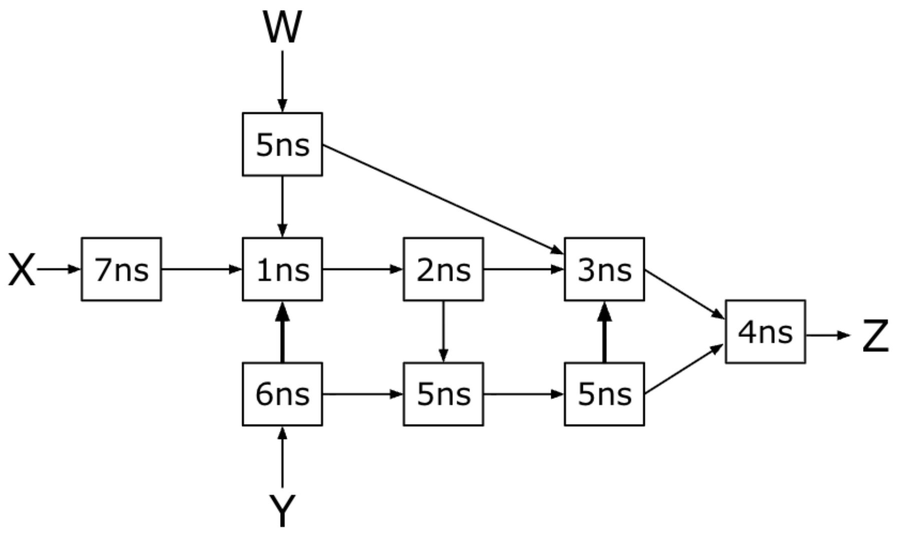
For each of the questions below, please create a valid K-stage pipeline of the given circuit. Each component in the circuit is annotated with its propagation delay in nanoseconds. **Show your pipelining contours** and place large black circles (●) on the signal arrows to **indicate the placement of pipeline 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.**

(A) (1 points) Based on the circuit shown in part (B), what are the latency and throughput of the **1-stage pipeline** for *ASSEMBLE*? Pay close attention to the direction of the arrows in the circuit.

Latency (ns): ______________________

Throughput (ns^{-1}): ______________________

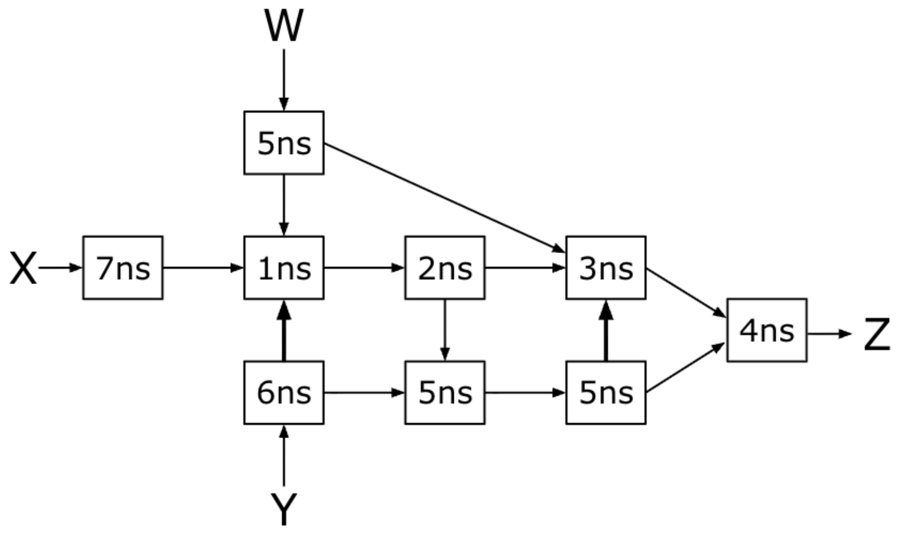
(B) (3 points) Show the **maximum-throughput 2-stage pipeline** for *ASSEMBLE* using a minimal number of registers. What are the latency and throughput of the resulting circuit?
Note: Extra copies of the circuit are provided at the end of the exam.



Latency (ns): ______________________

Throughput (ns^{-1}): ______________________

(C) (4 points) Show the **maximum-throughput pipeline** for *ASSEMBLE* using a minimal number of registers. What are the latency and throughput of the resulting circuit?



Latency (ns): ____________________

Throughput (ns^{-1}): ____________________

(D) (2 points) Now that we have three different pipelined versions of *ASSEMBLE*, Papa Louie wants you to help him implement the pizza-making module. He gives you his other submodules: *DOUGH* and *SAUCE*, shown below. The submodule *DOUGH* creates the pizza dough from scratch. It takes in three inputs **A**, **B**, and **C** and has two outputs: **F** and **G**. The submodule *SAUCE* cooks Papa Louie's secret sauce from scratch. It takes in two inputs: **D** and **E** and has one output: **H**. Given the implementations of *DOUGH* and *SAUCE* in figures 1 and 2, **draw the contour lines through each module's given registers**. Then, fill out the latency and throughput for each circuit in the table on the next page.

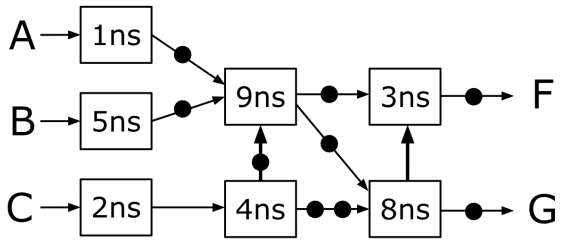


Figure 1: *DOUGH* module

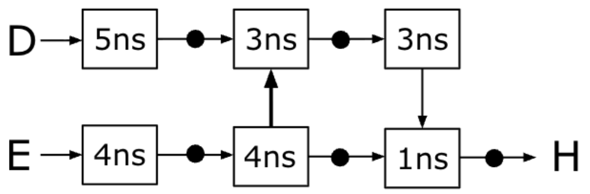
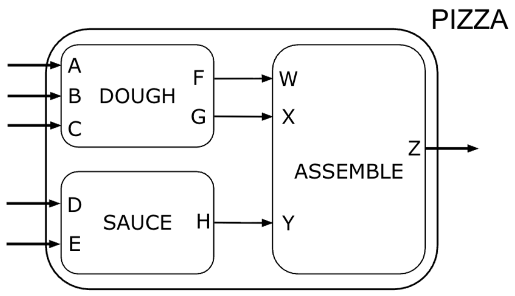


Figure 2: *SAUCE* module

Module	***Latency (ns)***	***Throughput (ns^{-1})***
DOUGH		
SAUCE		

(E) (6 points) We can now consider the complete pizza-making module below. Once the dough and sauce are made in their *DOUGH* and *SAUCE* modules, respectively, they are sent into the *ASSEMBLE* module, which outputs customers' assembled pizza.



1. Because of his busy pizzeria, Papa Louie would like you to maximize the throughput of *PIZZA*. Which pipelined version of *ASSEMBLE* should we implement? Circle the correct answer and provide the resulting latency and throughput of your ***PIZZA*** module. If two implementations have the same throughput, choose the implementation with the better latency.

***ASSEMBLE* (select one):** 1-stage pipeline 2-stage pipeline Maximum-throughput pipeline

Latency (ns): ______________

Throughput (ns^{-1}): ______________

2. Papa Louie is being fickle and decides that he'd rather minimize the latency of *PIZZA*. Which pipelined version of *ASSEMBLE* should we implement? Circle the correct answer and provide the resulting latency and throughput of your ***PIZZA*** module. If two implementations have the same latency, choose the implementation with the better throughput.

***ASSEMBLE* (select one):** 1-stage pipeline 2-stage pipeline Maximum-throughput pipeline

Latency (ns): ______________

Throughput (ns^{-1}): ______________

Problem 3. Processor Implementation (16 points)

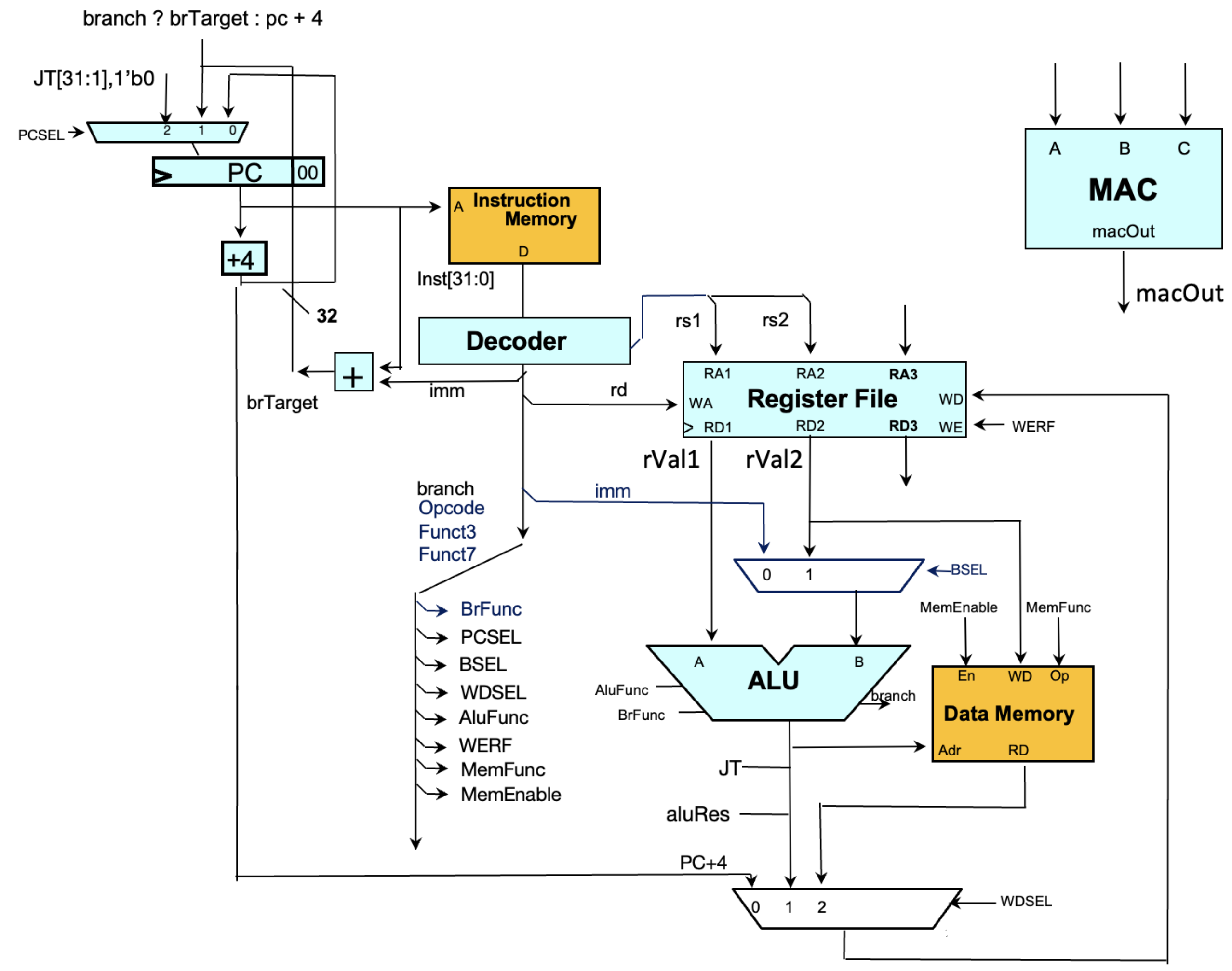
Philip Lopz really enjoyed taking 18.06 and wanted to try and implement matrix multiplication in RISC-V assembly for maximum performance. While writing his implementation, he realizes a pattern in the math and wants to create an instruction to speed up the algorithm. This leads to Philip's first attempt at creating a multiply and accumulate instruction:

```
mac1 rd, rs1, rs2 # reg[rd] <= (reg[rs1] * reg[rs2]) + reg[rd]
```

Unfortunately he notices a few problems with implementing this instruction. Two problems being the ALU has no multiplication instruction and three registers cannot be read in the same cycle. Despite this problem, Philip decides he will do whatever it takes to make this instruction a reality and adds a read port to the register file and a MAC unit that takes in inputs a, b, c and computes the output (a * b) + c.

(A) (4 points) The following processor diagram is a modified version of a single-cycle processor that you have seen in lecture that includes the MAC unit and a register file with an extra read port. Label all unconnected arrows (e.g. RA3 port), signal outputs from the decode unit, and input signals for the muxes to allow the processor to support the **mac1** instruction. The instruction has been repeated here for your convenience:

```
mac1 rd, rs1, rs2 # reg[rd] <= (reg[rs1] * reg[rs2]) + reg[rd]
```



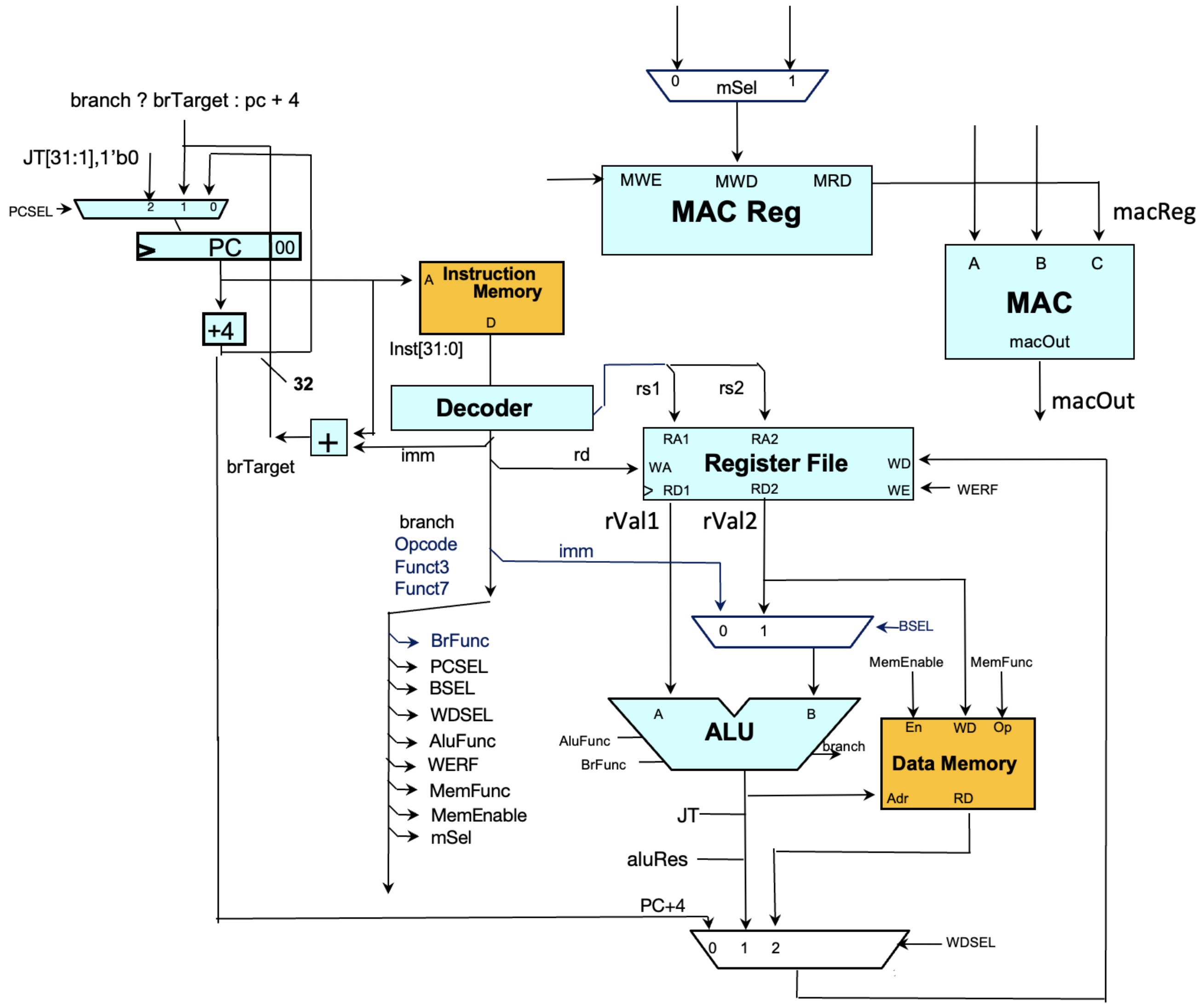
(B) (4 points) What does the decoder need to output for the following fields if given the instruction **mac1 x2, x5, x10**? Add any additional signals you may have added in part A. Mark any Don't Care values with a ?

mac1 x2, x5, x10	**Field**	**Value**
	imm	
	rs1	
	rs2	
	rd	
	AluFunc	
	BrFunc	
	BSEL	
	MemFunc	
	MemEnable	
	WDSEL	
	WERF	
	PCSEL	

Philip wants to continue exploring ways to implement MAC into RISC-V, so he comes up with a second implementation of an ISA with MAC that includes the MAC unit along with an additional register used exclusively for the MAC that will be referred to as macReg. This new ISA includes 3 new instructions:

```
macw rs1, constant     # macReg <= reg[rs1] + constant
macr rd                # reg[rd] <= macReg
mac2 rs1, rs2          # macReg <= (reg[rs1] * reg[rs2]) + macReg
```

(C) (4 points) The following processor diagram is a modified version of a single-cycle processor that includes the MAC unit and MAC Reg unit. MAC Reg has Write Enable (MWE) and Write Data (MWD) inputs and a Read Data (MRD) output. Label all unconnected arrows (e.g. MWE port), add any signal outputs from the decode unit, and input signals for the muxes (especially the new mSel mux) to allow this processor to support this ISA (macw, macr, mac2).



(D) (4 points) What does the decoder need to output for the following fields if given the instruction **macw x0, 0x100** and **macr x10**? Add any additional signals you may have added in part C. Mark any Don't Care values with a ?

	Field	**Value**
macw x0, 0x100	imm	
	rs1	
	rs2	
	rd	
	AluFunc	
	BrFunc	
	BSEL	
	MemFunc	
	MemEnable	
	WDSEL	
	WERF	
	PCSEL	
	MSEL	

	Field	**Value**
macr x10	imm	
	rs1	
	rs2	
	rd	
	AluFunc	
	BrFunc	
	BSEL	
	MemFunc	
	MemEnable	
	WDSEL	
	WERF	
	PCSEL	
	MSEL	

Problem 4. Caches (18 points)

Consider a memory that has been initialized to 0 at address 0x200, to 1 at address 0x204, 2 at address 0x208, 3 at address 0x20C, and so on, with each following data value increasing by 1 (as shown on the right). This pattern is repeated until address 0x300.

Given a direct mapped cache that can hold a total of 16 words and has a block size of 4, run the following code sequence and update the cache to indicate all changes made by executing this code. Assume that the cache was initially empty (i.e., all valid bits were 0). Assume that the same cache is used for instructions and data.

```
. = 0x1000
lw x2, 0x228(x0)
sb x2, 0x237(x0)
sh x2, 0x238(x0)
lw x3, 0x274(x0)
```

Address	Data
0x200	0x0
0x204	0x1
0x208	0x2
0x20C	0x3
0x210	0x4
0x214	0x5
0x218	0x6
0x21C	0x7
0x220	0x8
0x224	0x9
0x228	0xA
0x22C	0xB
0x230	0xC
0x234	0xD
0x238	0xE
0x23C	0xF
0x240	0x10
0x244	0x11
0x248	0x12
...	...

Main Memory

(A) (6 points) Update the direct mapped cache to indicate all known values that it holds after executing the code sequence. For instructions, you can just enter the instruction name (e.g., `lw`). Data should be shown in hexadecimal notation and should correspond to the data shown in memory manipulated by these instructions.

	Tag	V	D	Word 3	Word 2	Word 1	Word 0
0							
1							
2							
3							

(B) (4 points) How many instruction and data hits and misses did the execution of this code sequence produce? Assume that the code is running on a single cycle processor.

Instruction Hits: ____________________

Instruction Misses: ____________________

Data Hits: ____________________

Data Misses: ____________________

(C) (2 points) If any data had to be written back to memory during execution of this code, then provide the full list of affected word addresses and their updated data. (e.g., Mem[0x100] = 0x7)

List of memory word addresses and their data contents for all words that were written back from cache to main memory. Enter NONE for any extra rows.

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

Mem[__________**] =** ____________________

(D) (2 points) What are the values of the following registers after execution of this code?

x2 = ____________________

x3 = ____________________

Now consider running the following benchmark program on a **fully associative** cache that has a block size of 2 and can hold a total 16 words. The same cache is used for instructions and data. It uses an LRU replacement strategy. The benchmark runs through the loop 20 times and then executes the `unimp` instruction once and halts.

```
. = 0
 mv x4, x0        // byte index into array
loop:
 lw x2, 0x400(x4) // load next element of array
 slli x3, x2, 1   // perform computation
 sw x3, 0x700(x4) // store result in second array
 addi x4, x4, 4   // byte index of next array element
 slti x2, x4, 80  // process 20 entries
 bnez x2, loop
 unimp            // halt
```

(E) **(2 points)** How many instruction fetches does this program execute and what is the hit rate of the instruction fetches? Include one instruction fetch for the `unimp` instruction.

Number of instruction fetches: ____________________

Hit rate of instruction fetches: ____________________

(F) **(2 points)** How many data accesses does this program perform and what is the hit rate of the data accesses?

Number of data accesses: ____________________

Hit rate of data accesses: ____________________

Problem 5. Pipelined Processors (16 points)

The TAs are writing a data-processing kernel for a new embedded system. Their code needs to iterate through an array, update a 16-bit status flag in each entry of the array using the sh instruction, and then copy the entire 32-bit data word into the element of a different array.

They write the following RISC-V assembly loop and plan to run it on a standard 5-stage processor that:

- has full-bypassing
- always predicts that branches are not taken (always fetch from PC + 4)
- makes branch decisions in the EXE stage
- annuls instructions following taken branches
- all memory instructions complete in 1 cycle

```
// a1 holds the array base location
// a3 holds the length of the data array
// a4 holds the loop counter
// a5 holds the 16-bit status flag

. = 0x0
loop:
    sh a5, 0(a1)      // Modify lower half of array[i]
    lw a6, 0(a1)      // Load all 32-bits of array[i]
    sw a6, 100(a1)    // Store loaded value to new_array[i] with offset 100
    addi a4, a4, 1    // Increment loop counter
    addi a1, a1, 4    // Increment pointer to point to next element of the array
    blt a4, a3, loop  // Branch if i < array length
    slli a7, a5, 2
    xor a7, a7, a6
```

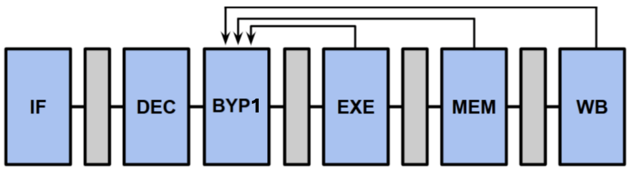
(A) (6 points) Fill in the following pipeline diagram. Assume that the loop has been running in steady-state and will continue to loop. Cycle 100 begins with fetching the sh instruction. Include any bypassing arrows.

	100	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**	**111**
IF	sh											
DEC	▨											
EXE	▨	▨										
MEM	▨	▨	▨									
WB	▨	▨	▨	▨								

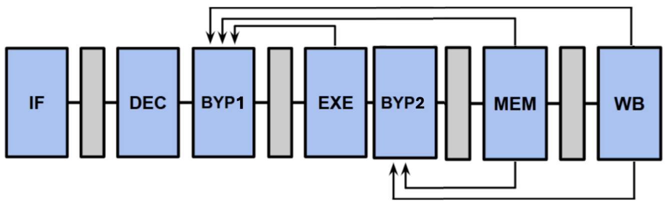
(B) (5 points) The TAs design an advanced processor, which has been significantly optimized. It has the same 5-stage pipeline, but features the following three major changes:

- Branch Resolution in DEC: Branch decisions and target address calculation are moved from the EXE stage to the DEC stage.
- Delayed Stalling: An instruction with a data dependency is allowed to proceed until it reaches the stage **immediately before** the stage where the value is needed. (i.e. an instruction that needs data in EXE must bypass in DEC, or an instruction that needs data in MEM must bypass in EXE)
- Extra Bypassing: This processor has additional bypass paths to support delayed stalling, including MEM -> EXE and WB -> EXE.

A block diagram of the original processor looks like a regular 5-stage pipeline with full bypassing as shown below:



A block diagram of the advanced processor has additional bypass paths as shown below:



Analyze the same code loop from Part A on this advanced processor by filling in the following diagram. Assume that the loop has been running in steady-state and will continue to loop. Include any bypassing arrows.

	100	101	102	103	104	105	106	107	108	109	110	111
IF	sh											
DEC	▩											
EXE	▩	▩										
MEM	▩	▩	▩									
WB	▩	▩	▩	▩								

(C) (2 points) In steady state how many cycles does the loop take for the original processor and how many for the advanced processor?

Cycles per loop iteration using original processor: ____________________

Cycles per loop iteration using advanced processor: ____________________

(D) (3 points) Moving the branch logic from the EXE stage to the DEC stage changes the combinational logic delays for those stages. This makes the DEC stage more complex and slower, while simplifying the EXE stage. Furthermore, the original processor only has a single BYP1 delay module to account for handling bypass paths in the DEC stage, whereas the advanced processor as a second, BYP2, module to account for the handling of the additional bypass paths. The delays for each pipeline stage in both processors are given below. Assume that the pipeline registers are ideal so they have a t_{PD} and t_{SETUP} of 0.

	Original Processor	**Advanced Processor**
IF	200 ns	200 ns
DEC	180 ns	300 ns
BYP1	100 ns	100 ns
EXE	250 ns	210 ns
BYP2	NA	80 ns
MEM	240 ns	240 ns
WB	150 ns	150 ns

Minimum clock period of original processor (in ns): ____________________

Total time/loop iteration for original processor (in ns): ____________________

Minimum clock period of advanced processor (in ns): ____________________

Total time/loop iteration for advanced processor (in ns): ____________________

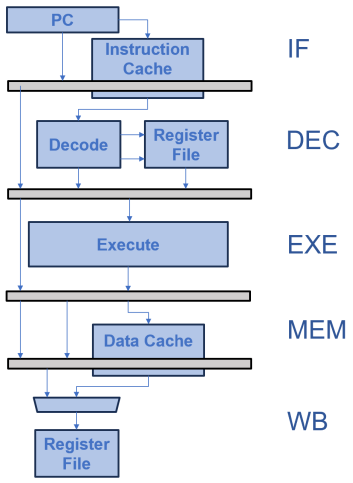
Which processor is actually faster for this loop? ____________________

Problem 6. Pipelined Processor Performance (16 points)

Having completed 6.1910, Ben Bitdiddle is designing his own highly performant 5-stage pipelined processor. He is using the following RISC-V assembly code to benchmark his processor design. The benchmark iterates over 16 elements in an array which begins at address `0x0`.

```
// a0 contains the starting address of the array: 0x0
// a1 contains the length of the array: 16
// a2 contains the current index of iteration, starting at 0
// a3 contains a running sum, starting at 0

. = 0x100
L1:
    slli a4, a2, 2          // store index as a byte count in a4
    add a4, a4, a0          // add array base to a4
    lw a5, 0(a4)            // load arr[i] into a5
    add a3, a3, a5          // add a5 to the running sum
    addi a2, a2, 1          // increment i
    blt a2, a1, L1          // loop while i < 16
```

Ben begins with a standard 5-stage pipelined RISC-V processor with branch annulment and full bypassing that predicts that branches are not taken. Remembering the 6.191 lectures on caches, Ben decides to add a cache to his processor hoping to exploit locality. He finds two direct mapped caches each with a single line and four words total capacity, so he decides to configure his processor to use one cache as a dedicated data cache and the other as a dedicated instruction cache.
With his new caches in place, he measures that **accessing memory from his processor takes a single cycle in the case of a cache hit, two cycles in the case of a clean miss, and three cycles in the case of a dirty miss for both caches**.
A diagram of the processor is given below. Notice that the processor uses realistic, cached memory with clocked reads that can have cache misses as well as that the results of cache accesses are returned in DEC and WB.



(A) (10 points) Fill in the pipeline diagram for the first 16 cycles of the benchmark's execution, making sure to account for stalls due to cache misses. You may assume that instruction fetches to cache completely restart every cycle that the instruction is in the IF stage, i.e. fetching an instruction multiple times while stalling in IF will not prevent it from causing a cache miss. Draw in arrows for bypassed values and if the instruction is unknown write ?. Then give the average cycles per instruction for the entire benchmark along with an explanation for how you arrived at that CPI. *Hint: Don't forget to account for accessing all 16 elements of the array in the CPI.*

Note: We have provided several blank pipeline diagrams at the end of this problem should you wish to use them.

	0	1	2	3	4	5	6	7
IF								
DEC	▨							
EXE	▨	▨						
MEM	▨	▨	▨					
WB	▨	▨	▨	▨				

	8	9	10	11	12	13	14	15
IF								
DEC								
EXE								
MEM								
WB								

Average CPI for benchmark: ____________________

Explanation:

Thinking his processors are production ready, Ben has them fabricated. He receives a box full of them but, in his excitment, drops the box and breaks the processors. Devastated, Ben tries to see how bad the damage is. He notices that after dropping all the processors, any cache access from his processors are now always cache hits, as well as that various functionalities of the processors are broken. In general, he notices that any given processor in the box fits into one of the two following categories:

- **Processor A**: A broken version of the processor from Part A with no cache misses and with broken load-to-use bypassing: the results of loads are bypassed from the EXE stage instead of the WB stage using the value in the register file, i.e. `lw a0, 0(x0)` would return the current value of `a0` in the register file.
- **Processor B**: A broken version of the processor from Part A with no cache misses, broken `nextPc` speculation, and bypassing disabled: the processor stalls instead of speculating the `nextPc` until it can figure out `nextPc` and always stalls on data hazards.

In order to help classify the broken processors, Ben writes the following code and runs it on each processor. The program loads two words from memory, one at address `0x0` with value `0x04030201` and another at address `0x4` with value `0x80706050`, **multiplies the first value by 2**, and then adds them.

```
// all registers start as 0
    lw a0, 0(x0)        // a0 = Mem[0]
    beq a0, x0, L2      // if (a0 != 0)
    slli a0, a0, 1      //      a0 *= 2
    mv a1, a0           //      a1 = a0
L2: lw a0, 4(x0)        // a0 = Mem[4]
    add a1, a1, a0      // a1 += a0
```

(B) (6 points) Ben has several pipeline diagrams but can't figure out which ones correspond to his broken processors. Help Ben figure out how badly broken the processors are by matching processor A and B to their respective pipeline diagrams out of the four shown below. Additionally, give the value of the sum in `a1` at the end of the program for both processors. **Remember that all memory accesses in this part are cache hits and assume that all registers start as 0.**

Diagram Corresponding to Processor A: I II III IV

Processor A Calculated Sum: ______________________

Diagram Corresponding to Processor B: I II III IV

Processor B Calculated Sum: ______________________

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	lw	beq	slli	mv	lw	add										
DEC	▨	lw	beq	slli	NOP	lw	add									
EXE	▨	▨	lw	beq	NOP	NOP	lw	add								
MEM	▨	▨	▨	lw	beq	NOP	NOP	lw	add							
WB	▨	▨	▨	▨	lw	beq	NOP	NOP	lw	add						

Table 1: Pipeline Diagram I

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	lw	beq	slli	mv	mv	mv	lw	add								
DEC	▨	lw	beq	slli	slli	slli	mv	lw	add							
EXE	▨	▨	lw	beq	NOP	NOP	slli	mv	lw	add						
MEM	▨	▨	▨	lw	beq	NOP	NOP	slli	mv	lw	add					
WB	▨	▨	▨	▨	lw	beq	NOP	NOP	slli	mv	lw	add				

Table 2: Pipeline Diagram II

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	lw	beq	slli	slli	slli	mv	lw	lw	lw	add						
DEC	▨	lw	beq	beq	beq	slli	mv	mv	mv	lw	add	add	add			
EXE	▨	▨	lw	NOP	NOP	beq	slli	NOP	NOP	mv	lw	NOP	NOP	add		
MEM	▨	▨	▨	lw	NOP	NOP	beq	slli	NOP	NOP	mv	lw	NOP	NOP	add	
WB	▨	▨	▨	▨	lw	NOP	NOP	beq	slli	NOP	NOP	mv	lw	NOP	NOP	add

Table 3: Pipeline Diagram III

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	lw	NOP	beq	NOP	NOP	NOP	NOP	slli	NOP	mv	NOP	lw	lw	NOP	add	
DEC	▨	lw	NOP	beq	beq	beq	NOP	NOP	slli	NOP	mv	mv	mv	lw	NOP	add
EXE	▨	▨	lw	NOP	NOP	NOP	beq	NOP	NOP	slli	NOP	NOP	NOP	mv	lw	NOP
MEM	▨	▨	▨	lw	NOP	NOP	NOP	beq	NOP	NOP	slli	NOP	NOP	NOP	mv	lw
WB	▨	▨	▨	▨	lw	NOP	NOP	NOP	beq	NOP	NOP	slli	NOP	NOP	NOP	mv

Table 4: Pipeline Diagram IV

Extra pipeline diagrams for problem 6A:

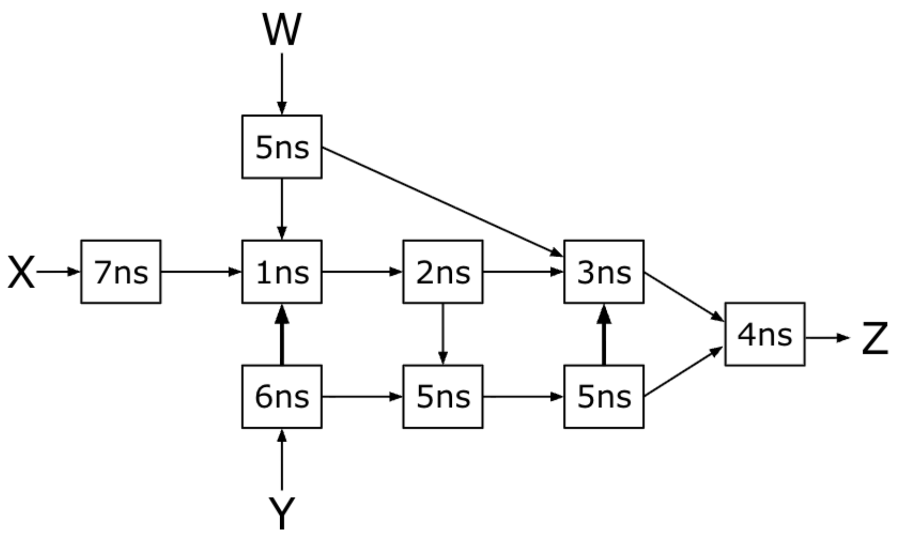
	0	1	2	3	4	5	6	7
IF								
DEC	▨							
EXE	▨	▨						
MEM	▨	▨	▨					
WB	▨	▨	▨	▨				

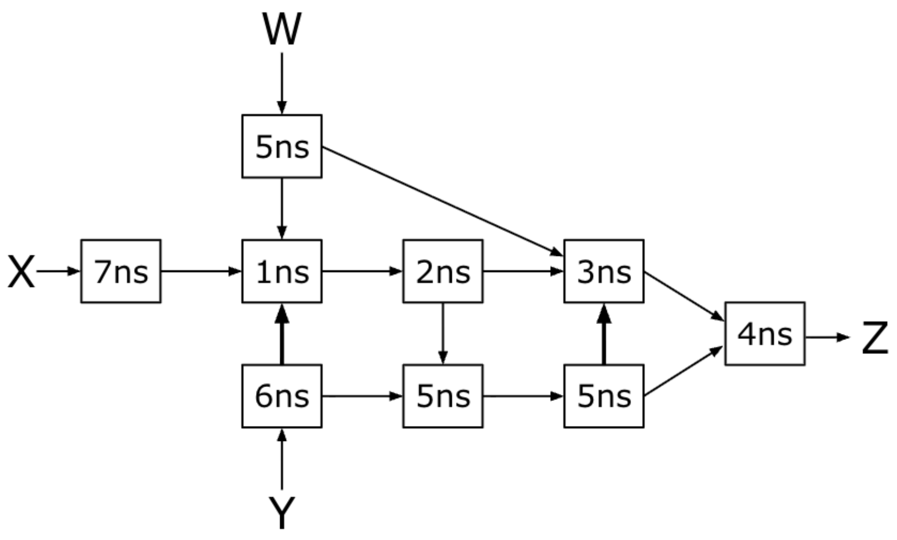
	8	9	10	11	12	13	14	15
IF								
DEC								
EXE								
MEM								
WB								

	0	1	2	3	4	5	6	7
IF								
DEC	▨							
EXE	▨	▨						
MEM	▨	▨	▨					
WB	▨	▨	▨	▨				

	8	9	10	11	12	13	14	15
IF								
DEC								
EXE								
MEM								
WB								

Extra diagrams for problem 2:





END OF QUIZ 2!