

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Spring 2025

1	/19
2	/16
3	/16
4	/18
5	/12
6	/19

Quiz #2

<i>Name</i>	<i>Athena login name</i>	<i>Score</i>
Solutions		
<i>Recitation section</i>		
o WF 10, 34-302 (Hilary)	o WF 2, 34-302 (Raymond)	o WF 12, 35-308 (Keshav)
o WF 11, 34-302 (Hilary)	o WF 3, 34-302 (Raymond)	o WF 1, 35-308 (Keshav)
o WF 12, 34-302 (Ezra)	o WF 10, 35-308 (Harry)	o WF 2, 8-205 (Vedantha)
o WF 1, 34-302 (Ezra)	o WF 11, 35-308 (Harry)	o WF 3, 8-205 (Vedantha)
		o 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.

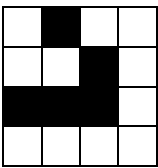
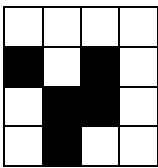
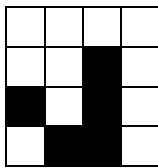
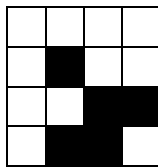
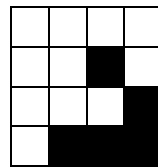
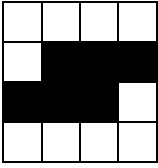
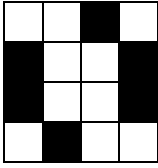
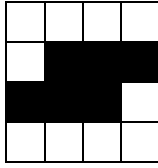
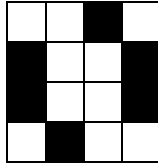
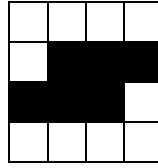
Problem 1. Sequential Logic in Minispec (19 points)

Ben Bitdiddle is building a sequential circuit that simulates Conway's Game of Life. This game consists of a 2D grid of cells. Each cell can be in one of two states, *dead* or *live*. The grid starts with some initial configuration of dead and live cells, and evolves in steps called generations. At each step, each cell evolves based on the state of its neighbors, using three simple rules:

1. Underpopulation: A live cell with less than 2 live neighbors becomes dead.
2. Reproduction: A dead cell with exactly 3 live neighbors becomes live.
3. Overpopulation: A live cell with more than 3 live neighbors becomes dead.

If none of these rules apply, a cell does not change state. Cells have up to 8 neighbors, those that are horizontally, vertically, or diagonally adjacent. We will simulate finite 2D grids, so cells at the edges of the grids can have fewer neighbors.

Although this game has simple rules, it can result in surprisingly complex patterns, as illustrated in the examples below (**live cells are shaded black, dead cells are white**). Each row is a separate example.

Initial state (generation 0)	Generation 1	Generation 2	Generation 3	Generation 4
				
				

(A) (2 points) Complete the combinational function below, which computes the next value of a cell based on its current value and the current count of neighbors that are live. We represent the state of each cell using a 1-bit value: 1 if live, 0 if dead.

```
function Bit#(1) evolveCell(Bit#(1) curValue, Bit#(4) liveNeighbors);
    return (liveNeighbors < 2 || liveNeighbors > 3)? 0 : (liveNeighbors ==
3)? 1 : curValue;
endfunction
```

Alternate solution:

```
function Bit#(1) evolveCell(Bit#(1) curValue, Bit#(4) liveNeighbors);
    if (curValue == 1 && (liveNeighbors < 2 || liveNeighbors > 3))
return 0;
    if (curValue == 0 && liveNeighbors == 3) return 1;
    return curValue;
endfunction
```

- (B) (6 points) Complete the sequential circuit on the next page, which simulates a tick of Game of Life each clock cycle, updating the state of all cells. The module simulates a square (n-by-n) 2D grid of cells. The start input provides the initial state of the game, and the readCells method outputs the state every cycle. You can call the `evolveCell` function from part A.

NOTE: Cells at the edges of the grid have fewer than 8 neighbors, so when indexing neighbors, watch out for out-of-bounds indices.

```
// Sums 8 1-bit values
function Bit#(4) sum8(Bit#(1) v1, Bit#(1) v2, Bit#(1) v3, Bit#(1) v4,
                    Bit#(1) v5, Bit#(1) v6, Bit#(1) v7, Bit#(1) v8);
    Bit#(2) s1 = {0, v1} + {0, v2} + {0, v3};
    Bit#(2) s2 = {0, v4} + {0, v5} + {0, v6};
    Bit#(2) s3 = {0, v7} + {0, v8};
    return {0, s1} + {0, s2} + {0, s3};
endfunction
```

```

module GameOfLife#(Integer n);
  Vector#(n, Vector#(n, RegU#(Bit#(1)))) cells;

  input Maybe#(Vector#(n, Vector#(n, Bit#(1)))) start default = Invalid;

  rule tick;
    if (__isValid(start)) _____) begin
      // Load new input
      for (Integer i = 0; i < n; i = i + 1) begin
        for (Integer j = 0; j < n; j = j + 1) begin

          cells[i][j] <= __fromMaybe(?, start)[i][j]_____;
        end
      end
    end else begin
      // Evolve all cells in the 2D grid
      for (Integer i = 0; i < n; i = i + 1) begin
        for (Integer j = 0; j < n; j = j + 1) begin
          Bit#(4) liveNeighbors = sum8(

            (i > 0 && j > 0)? cells[i-1][j-1] : 0,
            (i > 0)? cells[i-1][j] : 0,
            (i > 0 && j < n-1)? cells[i-1][j+1] : 0,
            (j > 0)? cells[i][j-1] : 0,
            (j < n-1)? cells[i][j+1] : 0,
            (i < n-1 && j > 0)? cells[i+1][j-1] : 0,
            (i < n-1)? cells[i+1][j] : 0,
            (i < n-1 && j < n-1)? cells[i+1][j+1] : 0

          );

          cells[i][j] <= __ evolveCell(cells[i][j], liveNeighbors)____;
        end
      end
    end
  endrule

  // map(map(readReg), cells) returns all the state in the correct format
  method Vector#(n, Vector#(n, Bit#(1))) readCells =
    map(map(readReg), cells);
endmodule

```

- (C) (8 points) Complete the sequential circuit below, which is a folded version of that in part B. This circuit updates *a single row* of the 2D grid each cycle. Follow the directions in the comments near each blank. You can assume that n is a power of 2. You can call the `evolveCell` function from part A.

NOTE: Beware that updating row i requires the values of rows $i-1$, i , and $i+1$ for the current generation. Make sure you don't mix cell values from different generations!

```

module GameOfLifeFolded#(Integer n);
  Vector#(n, Reg#(Bit#(n))) rowRegs(0);
  Reg#(Bit#(log2(n))) curRowIdx(0);

  // Define additional state elements here
  Reg#(Bit#(n)) lastRow(0);

  input Maybe#(Vector#(n, Bit#(n))) start default = Invalid;

  rule tick;
    if (___isValid(start)___) begin
      // Load new input
      for (Integer i = 0; i < n; i = i + 1) begin
        rowRegs[i] <= ___ fromMaybe(?, start)[i]___;
      end
      curRowIdx <= ___ 0 ___;
    end else begin
      // Compute new values for row of cells at index curRowIdx
      Bit#(n) rowAbove = ___(curRowIdx == 0)? 0 : lastRow ___;
      Bit#(n) curRow = ___ rowRegs[curRowIdx]___;
      Bit#(n) rowBelow = ___(curRowIdx == n-1)? 0 : rowRegs[curRowIdx+1];

      Bit#(n) newRow = curRow;
      for (Integer j = 0; j < n; j = j+1) begin
        Bit#(4) liveNeighbors = sum8(
          (rowAbove >> 1)[j], rowAbove[j], (rowAbove << 1)[j],
          (curRow >> 1)[j], (curRow << 1)[j],
          (rowBelow >> 1)[j], rowBelow[j], (rowBelow << 1)[j]);

        newRow[j] = ___ evolveCell(curRow[j], liveNeighbors)___;
      end
      rowRegs[curRowIdx] <= ___ newRow ___;
      curRowIdx <= ___(curRowIdx == n-1)? 0 : (curRowIdx + 1)___;

      // Update any additional state elements below
      lastRow <= curRow;
    end
  endrule

  // Should return a valid output ONLY when rowRegs is fully updated
  // (i.e., rowRegs values are all from the same generation)
  // map(readReg, rowRegs) returns all the state in the correct format
  method Maybe#(Vector#(n, Bit#(n))) readCells = ___ curRowIdx == 0 ___ ?
    Valid(map(readReg, rowRegs)) : Invalid;
endmodule

```

(D) (3 points) For the previous implementations in parts B and C, how do the area, clock cycle time, and throughput (in generations/second) grow with parameter n ? Use order-of notation. Assume propagation delay depends only on gate delays, ignoring fanout and wire delays.

	GameOfLife#(n)	GameOfLifeFolded#(n)
Area	$\Theta(\underline{\text{ } n^2 \text{ } })$	$\Theta(\underline{\text{ } n^2 \text{ } })$
t_{CLK}	$\Theta(\underline{\text{ } 1 \text{ } })$	$\Theta(\underline{\text{ } \log n \text{ or } 1 \text{ } })$
Throughput (generations/s)	$\Theta(\underline{\text{ } 1 \text{ } })$	$\Theta(\underline{1/n \log n (t_{\text{CLK}} - \log n) \text{ or } 1/n (t_{\text{CLK}} = 1)})$

Explanation:

- Area – Both GameOfLife#(n) and GameOfLifeFolded#(n) have n^2 1-bit registers, so their total area is $\Theta(n^2)$.
- $t_{\text{CLK}} - t_{\text{CLK}}$ for GameOfLife#(n) is $\Theta(1)$ because the combinational logic that updates each register checks a constant number of neighbors, so its delay does not grow with n .
 t_{CLK} for GameOfLifeFolded#(n) is $\Theta(\log n)$ because reading each row incurs $\Theta(\log n)$ delay (due to the n -to-1 mux in each read port). Since this can be easy to overlook, we also gave full credit for $\Theta(1)$.
 The non-folded version doesn't have this problem because there is no such muxing: there's much more logic, but each piece of logic is associated with a local neighborhood. You don't need to choose one of n rows to process. That n -to-1 downselect (as well as the decoder for the write-enable signals, which is also $\log n$) is what adds delay.
- Throughput – GameOfLife#(n) computes a new generation every cycle, so its throughput is $\Theta(1/t_{\text{CLK}}) = \Theta(1)$. GameOfLifeFolded#(n) computes a new generation every n cycles, so its throughput is $\Theta(1/(n \cdot t_{\text{CLK}})) = \Theta(1/n \log n)$. (or, $\Theta(1/n)$ if you answered $t_{\text{CLK}} = \Theta(1)$ above)

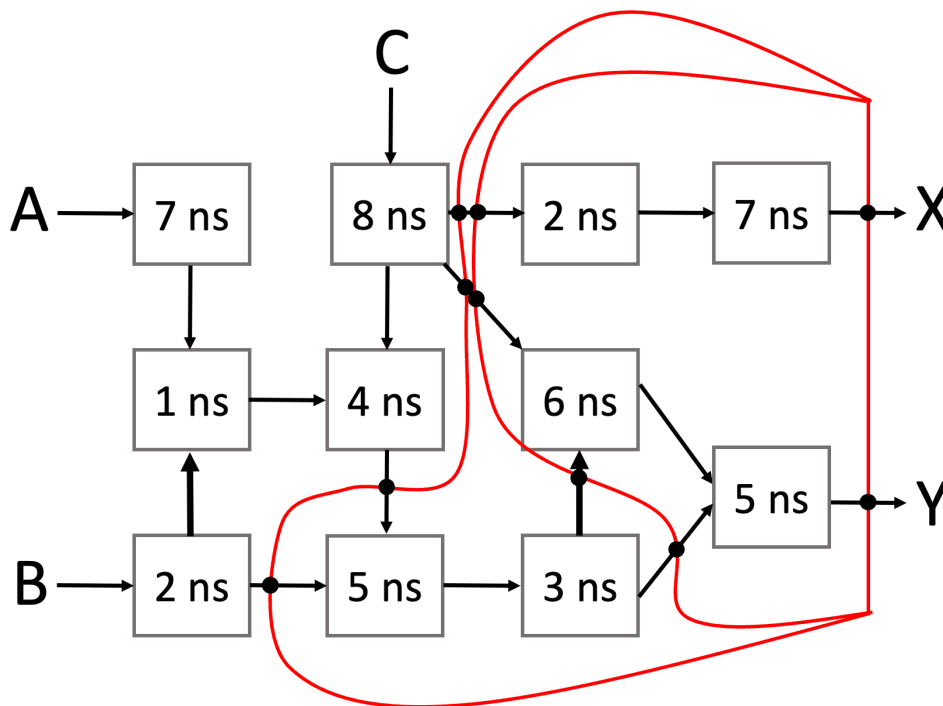
Problem 2. Convolved Arithmetic Pipelines (16 points)

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) (2 points) What is the propagation delay, t_{PD} , of the original combinational circuit shown in part (B), prior to pipelining?

t_{PD} (ns): 31

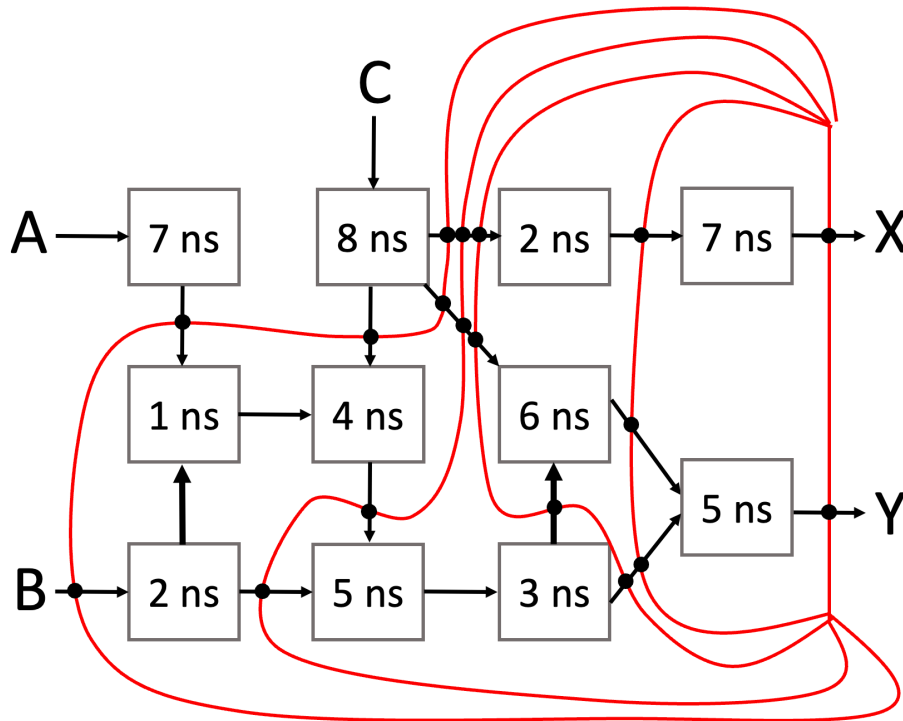
(B) (4 points) Show a **maximum-throughput 3-stage pipeline** using a minimal number of registers. **Invalid pipelines will earn 0 points.** What are the latency and throughput of the resulting circuit? *In case you need them, extra copies of the circuit are available at the end of the exam.*



Latency (ns): 36

Throughput (ns^{-1}): 1/12

- (C) (4 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): 40

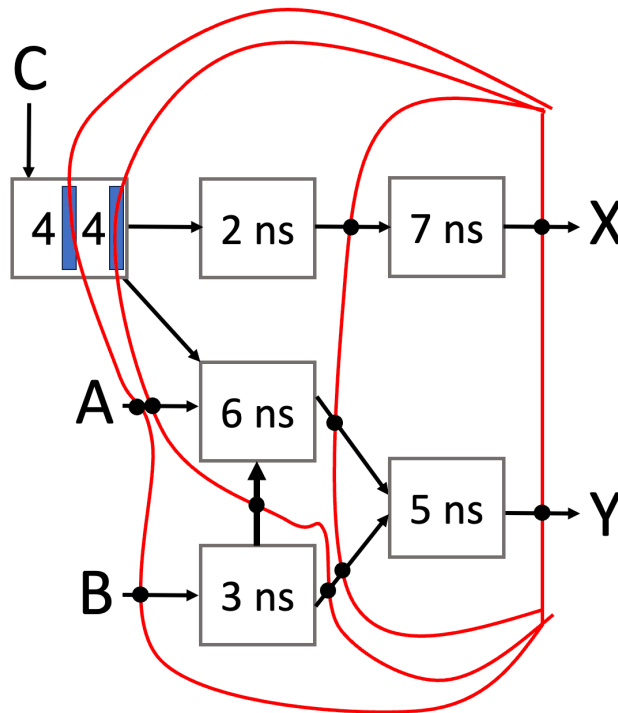
Throughput (ns^{-1}): 1/8

- (D) (2 points) How would the latency and throughput of your maximum throughput circuit change if instead of using ideal registers to pipeline your circuit, you only had registers with a ($t_{PD}=1$, $t_{SETUP}=1$) available. What would be the latency and throughput of the same max throughput circuit of part C using these non-ideal registers?

Latency (ns): 50

Throughput (ns^{-1}): 1/10

(E) (4 points) You are now tasked with pipelining a new circuit shown below. Note that this circuit contains an 8ns component, with input C, that has been replaced with a 2-stage pipelined version of that 8ns module, where each stage takes 4ns. The shaded rectangles are the registers that are already present in the 2-stage pipelined component. Show a maximum throughput pipeline using a minimal number of registers for this new circuit. Make sure to account for the two registers already present in the 8ns module in drawing your contours and labeling your 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.*



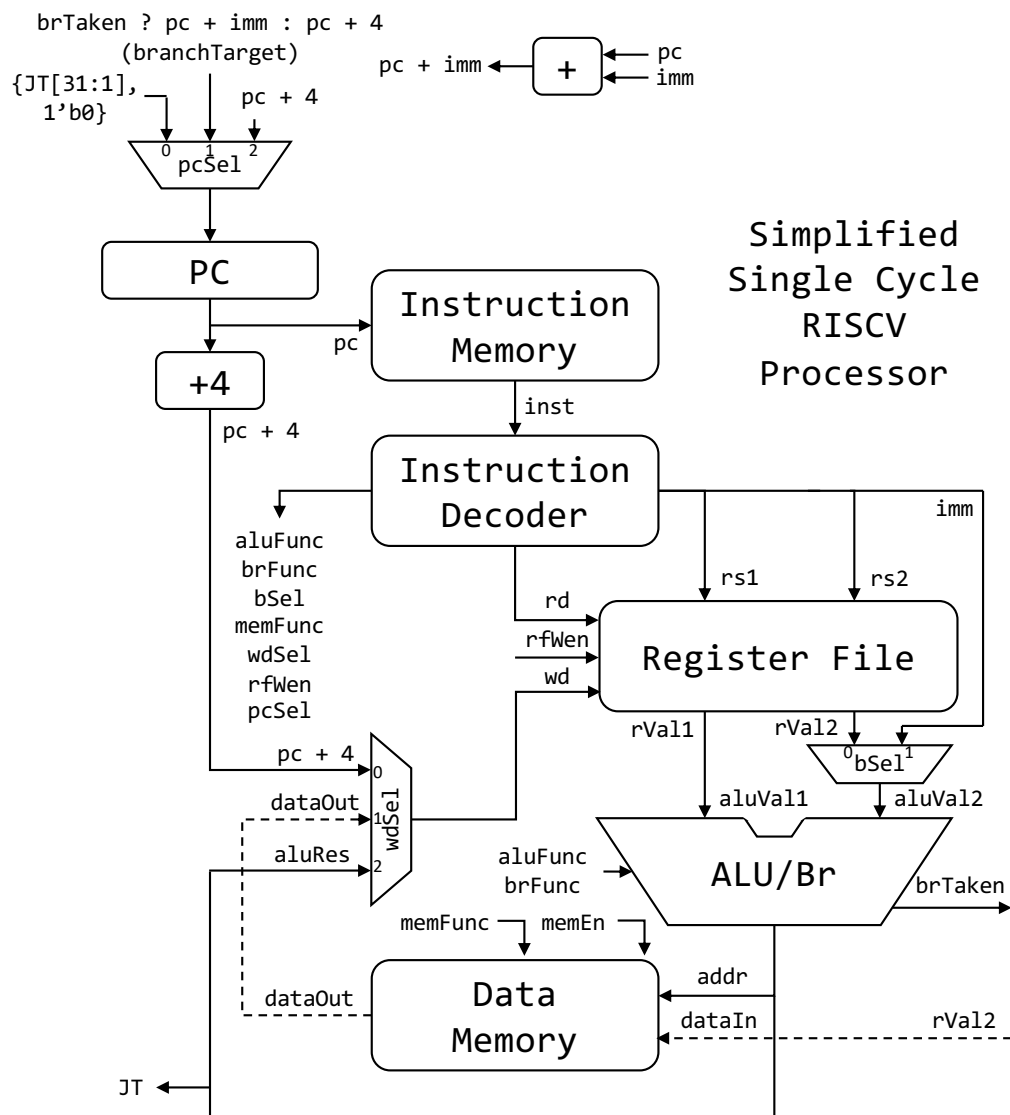
Latency (ns): 28

Throughput (ns⁻¹): 1/7

Problem 3: (not) sorry to interrupt! (Processor Implementation) (16 points)

The 6.004 RISC-V processors you have been building feel lonely running code all by themselves. They would like to be able to communicate with each other. Let us give them a very basic communication capability.

We start with this generic *single-cycle* RISC-V processor similar to the one you have seen in lecture.



As a reminder, for single-cycle per instruction operation without pipelining, **assume that the memory reads (both instruction and data) are combinational, as well as reads from the register file.** The instruction decoder decodes the instruction into the following fields:

Field	Description	Possible Values
imm	Immediate	Appropriate 32-bit constant (assume proper sign extension)
rs1, rs2	Source Registers	Integers between 0 and 31
rd	Destination Register	Integer between 0 and 31
aluFunc	ALU Function	Add, Sub, And, Or, Xor, Sll, Sra, Srl
brFunc	Branch Comp. Function	Eq, Neq, Lt, Leq, Gt, Geq
bSel	ALU/Br Operand 2 Select	0 (rVal2), 1 (imm), 2, 3, 4... (others, <i>if extended in later parts</i>)
memFunc	Data Memory Function	Lw, Sw, Lh, Lhu, Sh, Lb, Lbu, Sb
memEn	Memory Enable	True/False
wdSel	Write Data Select	0 (pc + 4), 1 (dataOut), 2 (aluRes), 3, 4, 5... (others, <i>if extended in later parts</i>)
rfWen	Register File Write Enable	True/False
pcSel	Next PC Select	0 (jumpTarget), 1 (branchTarget), 2 (pc + 4), 3, 4, 5... (others, <i>if extended in later parts</i>)

- (A) (2 points) As a warmup, complete the table on the right with what the decoded control signals should be for the `lui x15, 0xffff00` instruction shown. **Assume that the value to write into rd is computed by adding 0 to the decoded 32-bit immediate.** Use possible values from the above table. Write “?” for don’t-care values. The table on the left is provided as an example.

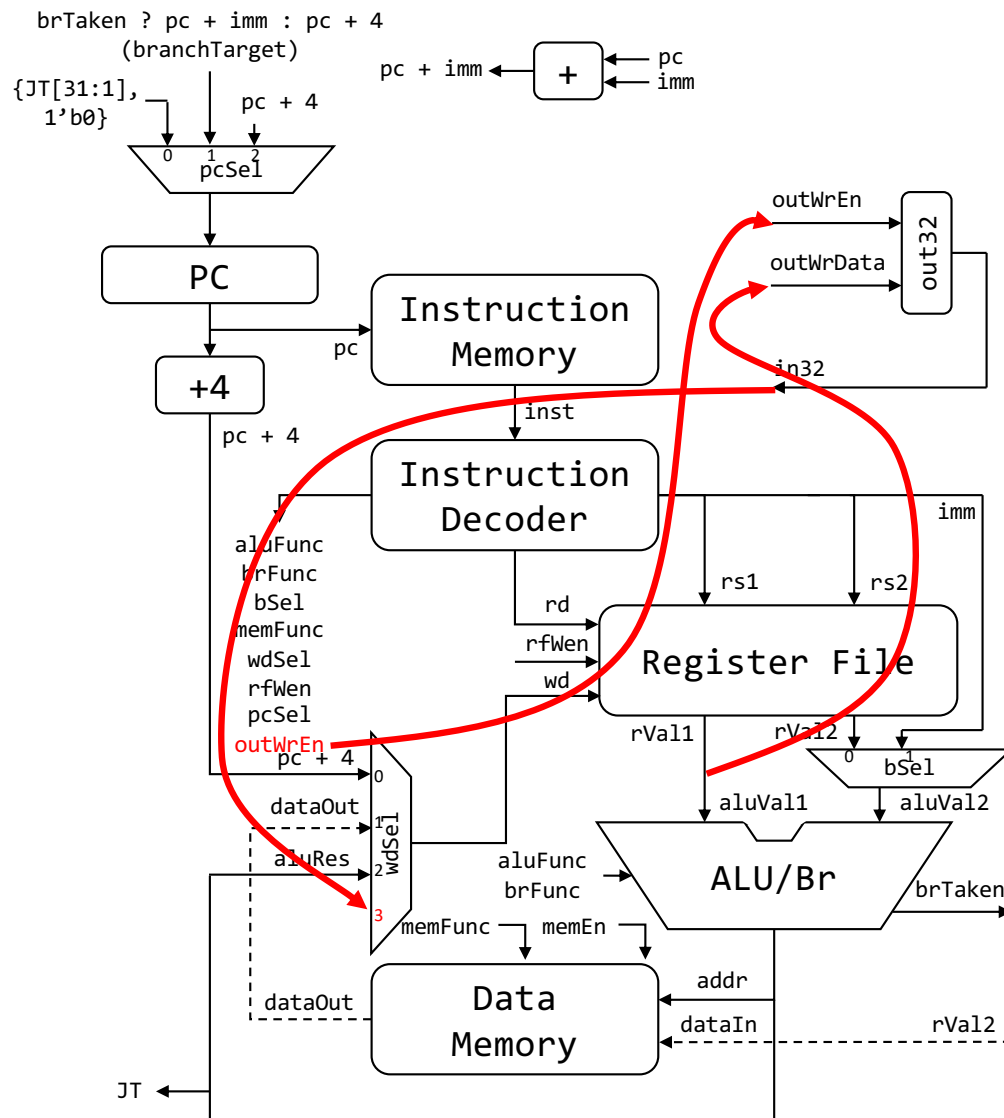
add x2, x19, x5	Field	Value
	imm	?
	rs1	19
	rs2	5
	rd	2
	aluFunc	Add
	brFunc	?
	bSel	0 (rVal2)
	memFunc	?
	memEn	False
	wdSel	2 (aluRes)
	rfWen	True
	pcSel	2 (pc + 4)
lui x15, 0xffff00	Field	Value
	imm	0xffff0000
	rs1	0
	rs2	?
	rd	15
	aluFunc	Add
	brFunc	?
	bSel	1 (imm)
	memFunc	?
	memEn	False
	wdSel	2 (aluRes)
	rfWen	True
	pcSel	2 (pc + 4)

- (B) (4 points) First, let us add input/output capabilities to our processor. Your processor now has a 32-bit input (called `in32`), the value of which comes in from a 32-bit register (called `out32`). You can assume that other devices in the external world are also able to change the value of `out32` but you do not need to worry about how that happens. We will implement an instruction called **Exchange Value**, which stores the value of source register `rs1` (in the processor’s register file) into `out32` and stores the value of `in32` into the destination register

rd (in the processor's register file). This has the effect of swapping the value currently stored in out32 with a value in our processor.

```
xchgv rd, rs1; // reg[rd] <= in32; out32 <= reg[rs1];
```

Connect the inputs and outputs of the out32 register to appropriate spots in the following diagram to make the xchgv instruction work. Feel free to add additional outputs from the decoder if needed, and additional pcSel and wdSel and bSel mux values if needed. *For full credit, use minimum such additions.*



(C) (4 points) What does the decoder need to output for the following fields when it sees an `xchgv` instruction shown below? Write “?” for don’t-care values. If needed, you should list any additional decoder outputs and additional `pcSel`, `wdSel`, and `bSel` mux values below. *For full credit, use minimum such additions.*

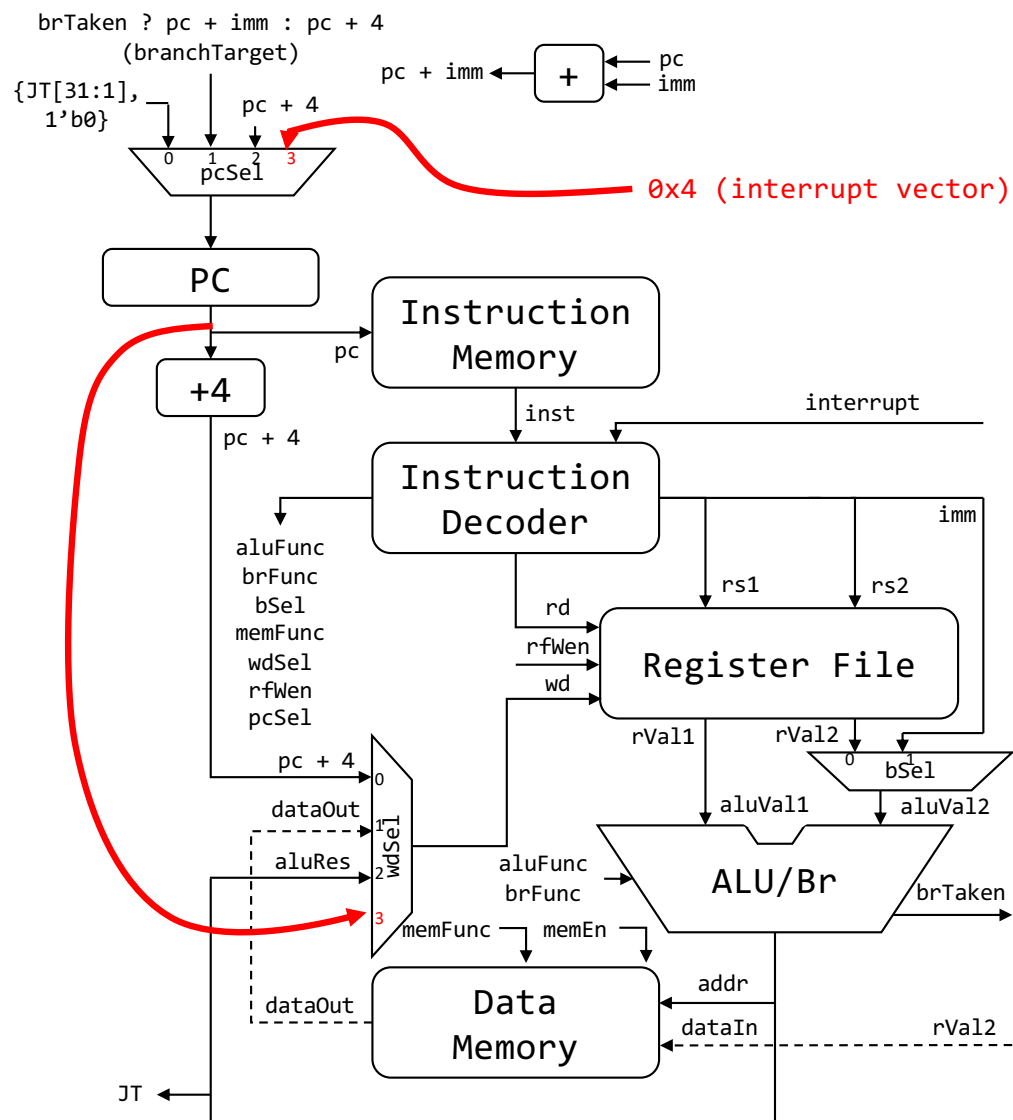
xchgv x6, x9	Field	Value
	imm	?
	rs1	9
	rs2	?
	rd	6
	aluFunc	?
	brFunc	?
	bSel	?
	memFunc	?
	memEn	False
	wdSel	3 (in32)
	rfWen	True
	pcSel	2 (pc + 4)
	outWrEn	True

(D) (6 points) Next, let us implement a way for the external world to let the processor know that there's something that needs to be looked at. We have a single bit wire coming in from the external world, called an interrupt. This wire usually stays low, but when it is asserted high by the external world, the processor should abandon the currently executing instruction, and instead jump to a special function called the interrupt handler. In order to achieve this, the processor should:

- Store the **current** pc value (**NOT** $pc + 4$) in register **x31**, so the interrupt handler can resume execution from the current instruction after it finishes executing.
- Jump to a special address called the interrupt vector (for simplicity, assume this is $0x4$). In other words, set $pc = 0x4$.

On the following page, please wire the processor so that it supports interrupts. Then complete the table below with what the decoder should output when the interrupt line is asserted? Assume that this implementation is independent of parts B and C (i.e., the **xchg** is not being implemented). If needed, you should list any additional decoder outputs and additional **pcSel**, **wdSel**, and **bSel** mux values below. *For full credit, use minimum such additions.*

(interrupt asserted)	Field	Value
	imm	?
	rs1	?
	rs2	?
	rd	31
	aluFunc	?
	brFunc	?
	bSel	?
	memFunc	?
	memEn	False
	wdSel	3 (pc)
	rfWen	True
	pcSel	3 (int. vect.)



Problem 4: Caches (18 points)

Frodo and Sam are trying to get to Mount Doom as quickly as possible, so they decide to use a cache to speed up their access to critical information.

(A) (1 point) Frodo and Sam would like to design a cache with an AMAT (average memory access time) of 3 cycles. Accessing their cache takes 1 cycle, and on a miss, it takes an additional 16 cycles to retrieve the data from main memory and update the cache. What does their hit ratio need to be in order to achieve the target AMAT?

$$3 = 1 + \text{miss ratio} * 16$$

$$2 = \text{miss ratio} * 16$$

$$\text{Hit ratio} = 1 - \text{miss ratio} = 1 - 1/8 = 7/8$$

$$\text{Hit ratio} = \underline{\quad 7/8 \quad}$$

(B) (2 points) Frodo proposes using a **2-way set associative cache** with a block size of 2 words. Frodo's cache stores a total of 8 words. Assume that addresses and words are 32 bits wide. What address bits should be used for the word offset, byte offset, cache index, and tag field? Write X:X if no bits are required for some field.

Tag field: A [31 : 4]

Word offset: A [2 : 2]

Byte offset: A [1 : 0]

Cache index: A [3 : 3]

(C) (1 point) Sam's cache also stores a total of 8 words. He proposes using a **direct mapped cache** with a block size of 2 words. Assume that addresses and words are 32 bits wide. What address bits should be used for the word offset, byte offset, cache index, and tag field? Write X:X if no bits are required for some field.

Tag field: A [31 : 5]

Word offset: A [2 : 2]

Byte offset: A [1 : 0]

Cache index: A [4 : 3]

(D) (1 point) Assuming that Sam's direct mapped cache uses a writeback policy, and includes valid bits, what is the total number of bits in his cache?

For each line in the cache, we have:

1 valid bit + 1 dirty bit + 27 tag bits + 64 bits of data = 93 bits

We have 4 lines in our cache, for a total of 372 bits

372 bits

(E) (3 points) We want to analyze the performance of Frodo's cache and Sam's cache on the following assembly program. **Assume there is a separate data and instruction cache**, and that all words in data memory start off as 0x00000000. What is the steady state hit ratio of Frodo's 2-way set associative data cache?

```

    mv x1, x0          // byte index into array
    li x2, 1000         // set bound for end of array
loop:
    lw x3, 0x0(x1)      // access elements from array
    lw x4, 0x4(x1)
    lw x5, 0x14(x1)
    lw x6, 0x18(x1)
    lw x7, 0x0(x1)
    lw x8, 0x4(x1)
    lw x9, 0x14(x1)
    lw x10, 0x18(x1)
    addi x1, x1, 8
    ble x1, x2, loop    // process entries until we reach end of array

```

Below, we show the end state of the cache at the end of each iteration assuming way 0 was considered the LRU initially.

Iteration 1:

We are accessing elements A[0], A[1], A[5], A[6], A[0], A[1], A[5], A[6]

Way 0	
Word 1	Word 0
A[1]	A[0]
A[7]	A[6]

Way 1	
Word 1	Word 0
A[5]	A[4]

5/8 hits: A[0]: M, A[1]: H, A[5]: M, A[6]: M, A[0]: H, A[1]: H, A[5]: H, A[6]: H

Iteration 2:

We are accessing elements A[2], A[3], A[7], A[8], A[2], A[3], A[7], A[8]

Way 0	
Word 1	Word 0
A[9]	A[8]
A[7]	A[6]

Way 1	
Word 1	Word 0
A[5]	A[4]
A[3]	A[2]

6/8 hits: A[2]: M, A[3]: H, A[7]: H, A[8]: M, A[2]: H, A[3]: H, A[7]: H, A[8]: H

Iteration 3:

We are accessing elements A[4], A[5], A[9], A[10], A[4], A[5], A[9], A[10]

Way 0	
Word 1	Word 0
A[9]	A[8]
A[7]	A[6]

Way 1	
Word 1	Word 0
A[5]	A[4]
A[11]	A[10]

7/8 hits: A[4]: H, A[5]: H, A[9]: H, A[10]: M, A[4]: H, A[5]: H, A[9]: H, A[10]: H

Iteration 4:

We are accessing elements A[6], A[7], A[11], A[12], A[6], A[7], A[11], A[12]

Way 0	
Word 1	Word 0
A[9]	A[8]
A[7]	A[6]

Way 1	
Word 1	Word 0
A[13]	A[12]
A[11]	A[10]

7/8 hits: A[6]: H, A[7]: H, A[11]: H, A[12]: M, A[6]: H, A[7]: H, A[11]: H, A[12]: H

Iteration 5:

We are accessing elements A[8], A[9], A[13], A[14], A[8], A[9], A[13], A[14]

Way 0	
Word 1	Word 0
A[9]	A[8]
A[15]	A[14]

Way 1	
Word 1	Word 0
A[13]	A[12]
A[11]	A[10]

7/8 hits: A[8]: H, A[9]: H, A[13]: H, A[14]: M, A[8]: H, A[9]: H, A[13]: H, A[14]: H

From then on, this pattern repeats with only one miss per loop iteration, so the steady state hit rate is 7/8.

Steady state hit ratio for Frodo's 2-way set associative cache = 7/8

(F) (3 points) What is the steady state hit ratio of Sam's direct mapped data cache on the assembly program from above?

Below, we show the end state of the cache at the end of each iteration.

Iteration 1:

We are accessing elements A[0], A[1], A[5], A[6], A[0], A[1], A[5], A[6]

Word 1	Word 0
A[1]	A[0]
A[5]	A[4]
A[7]	A[6]

5/8 hits: A[0]: M, A[1]: H, A[5]: M, A[6]: M, A[0]: H, A[1]: H, A[5]: H, A[6]: H

Iteration 2:

We are accessing elements A[2], A[3], A[7], A[8], A[2], A[3], A[7], A[8]

Word 1	Word 0
A[9]	A[8]
A[3]	A[2]
A[5]	A[4]
A[7]	A[6]

6/8 hits: A[2]: M, A[3]: H, A[7]: H, A[8]: M, A[2]: H, A[3]: H, A[7]: H, A[8]: H

Iteration 3:

We are accessing elements A[4], A[5], A[9], A[10], A[4], A[5], A[9], A[10]

Word 1	Word 0
A[9]	A[8]
A[11]	A[10]
A[5]	A[4]
A[7]	A[6]

7/8 hits: A[4]: H, A[5]: H, A[9]: H, A[10]: M, A[4]: H, A[5]: H, A[9]: H, A[10]: H

Iteration 4:

We are accessing elements A[6], A[7], A[11], A[12], A[6], A[7], A[11], A[12]

Word 1	Word 0
A[9]	A[8]
A[11]	A[10]
A[13]	A[12]
A[7]	A[6]

7/8 hits: A[6]: H, A[7]: H, A[11]: H, A[12]: M, A[6]: H, A[7]: H, A[11]: H, A[12]: H

Iteration 5:

We are accessing elements A[8], A[9], A[13], A[14], A[8], A[9], A[13], A[14]

Word 1	Word 0
A[9]	A[8]
A[11]	A[10]
A[13]	A[12]
A[15]	A[14]

7/8 hits: A[8]: H, A[9]: H, A[13]: H, A[14]: M, A[8]: H, A[9]: H, A[13]: H, A[14]: H

From then on, this pattern repeats with only one miss per loop iteration, so the steady state hit rate is 7/8.

Steady state hit ratio for Sam's direct mapped cache = 7/8

(G) (3 points) Now consider a slightly different access pattern (note that the array offsets have changed).

```

mv x1, x0          // byte index into array
li x2, 1000        // set bound for end of array
loop:
  lw x3, 0x0(x1)    // access elements from array
  lw x4, 0x4(x1)
  lw x5, 0x20(x1)
  lw x6, 0x24(x1)
  lw x7, 0x0(x1)
  lw x8, 0x4(x1)
  lw x9, 0x20(x1)
  lw x10, 0x24(x1)
  addi x1, x1, 8
  ble x1, x2, loop  // process entries until we reach end of array

```

What is the steady state hit ratio of Frodo's 2-way set associative data cache on this new access pattern?

Below, we show the end state of the cache at the end of each iteration assuming way 0 was considered the LRU initially.

Iteration 1:

We are accessing elements A[0], A[1], A[8], A[9], A[0], A[1], A[8], A[9]

Way 0	
Word 1	Word 0
A[1]	A[0]

Way 1	
Word 1	Word 0
A[9]	A[8]

6/8 hits: A[0]: M, A[1]: H, A[8]: M, A[9]: H, A[0]: H, A[1]: H, A[5]: H, A[6]: H

Iteration 2:

We are accessing elements A[2], A[3], A[10], A[11], A[2], A[3], A[10], A[11]

Way 0	
Word 1	Word 0
A[1]	A[0]
A[3]	A[2]

Way 1	
Word 1	Word 0
A[9]	A[8]
A[11]	A[10]

6/8 hits: A[2]: M, A[3]: H, A[10]: M, A[11]: H, A[2]: H, A[3]: H, A[10]: H, A[11]: H

Iteration 3:

We are accessing elements A[4], A[5], A[12], A[13], A[4], A[5], A[12], A[13]

Way 0	
Word 1	Word 0
A[5]	A[4]
A[3]	A[2]

Way 1	
Word 1	Word 0
A[13]	A[12]
A[11]	A[10]

6/8 hits: A[4]: H, A[5]: H, A[12]: M, A[13]: H, A[4]: H, A[5]: H, A[12]: H, A[13]: H

We notice that in each iteration of the loop, we have two cold misses and six cache hits. This gives us a hit ratio of $\frac{3}{4}$.

Steady state hit ratio = 3/4

(H) (4 points) Frodo and Sam get some advice from Gandalf and decide to use his cache, which is a 2-way set associative cache that begins with the initial state shown below.

Way 1						Way 0						
LRU		V	D	tag	Word 1	Word 0		V	D	tag	Word 1	Word 0
0	0	0	0	0xA3	0x64A9	0x87B1	0	1	0	0x0F	0x99C7	0xB69B
1	1	1	0	0x27	0xE57B	0x961C	1	1	1	0x6B	0x614D	0x2448

For each of the following memory accesses, determine if it results in a hit or a miss. Consider each request independently.

Instruction	Data Returned' If hit, what data is returned? If miss, enter NA.	Address to Update If hit, enter NA. If miss, list all addresses that need to be updated in memory, or enter NONE of no updates are necessary.
lw x1, 0x61C(x0)	NA	NONE (LRU is 1 and it has D = 0)
lw x1, 0xA34(x0)	NA (valid bit is 0)	NONE (D = 0)

Problem 5: Software can fix everything. (12 points)

Assume our five-stage pipelined processor has no stall logic or bypass logic added, but has speculation logic, i.e., $nextPC = PC + 4$, implemented, as well as annulling logic on mispredictions for correct functionality. Also, assume that branches are resolved in the EXE stage. For each of the code snippets below, insert NOPs so the following code run on this pipelined processor produces the same results as when run on an unpipelined processor. **For full credit, minimize** the number of NOPs **executed**.

Write all NOPs between any two instructions on a single line separated by semicolons. If no NOPs are required, write “No NOPs required.”

(A) (2 points)

```
loop:  addi x15, x10, 2
      sub x13, x16, x10
      bne x10, x0, loop
      lw x17, 0(x10)
      addi x18, x10, 7
```

No NOPs required.

(B) (2 points)

```
loop:  addi x15, x10, 2
      NOP; NOP; NOP
      sub x13, x15, x10
      bne x10, x0, loop
      lw x17, 0(x10)
      addi x18, x10, 7
```

(C) (2 points)

```
loop:  addi x15, x10, 2
      sub x13, x16, x10
      bne x10, x0, loop
      lw x17, 0(x10)
```

```
NOP; NOP; NOP
addi x18, x17, 7
```

(D) (2 points)

```
loop: addi x15, x10, 2

      sub x13, x16, x10
      NOP; NOP; NOP
      bne x13, x0, loop

      lw x17, 0(x13)

      addi x18, x10, 7
```

(E) (2 points)

```
loop: addi x15, x10, 2

      sub x13, x16, x10

      bne x10, x0, loop
      NOP
      lw x17, 0(x15)

      addi x18, x10, 7
```

Should not put a **NOP** in the loop, e.g., after addi or sub, since it is less performant.

(F) (2 points)

```
loop: addi x15, x10, 2

      sub x13, x16, x10
      NOP; NOP;
      lw x17, 0(x15)

      bne x13, x0, loop
      NOP; NOP;
      addi x18, x17, 7
```

Problem 6. Pipelined Processor Performance (19 points)

Ben Bitdiddle decided to use his newfound RISC-V skills to calculate Fibonacci numbers. He writes the following loop in RISC-V assembly that calculates the 25th Fibonacci number and stores it in register `a0`. Assume registers `t1` and `t2` each contain writable memory addresses.

```

    addi t0, x0, 24    // t0 = 24
    addi a0, x0, 0     // fib(0) = 0
    sw a0, 0(t1)
    addi a1, x0, 1     // fib(1) = 1
    sw a1, 0(t2)

fib: lw a0, 0(t1)
     lw a1, 0(t2)
     add a2, a0, a1    // fib(n) = fib(n-1) + fib(n-2)
     sw a1, 0(t1)      // update fib(n-2) to fib(n-1)
     sw a2, 0(t2)      // update fib(n-1) to fib(n)
     addi t0, t0, -1
     bnez t0, fib

     lw a0, 0(t2)      // a0 <- fib(25)
     xori a4, a2, 2
     ret

```

Ben runs this code on a standard 5-stage RISC-V processor with full bypassing and branch annulment. Assume that branches are always predicted not taken (i.e., we speculate that the branch is not taken) and that branch decisions are made in the EXE stage. Assume that the loop repeats many times and it's currently in the middle of its execution.

(A) (7 points) Fill in the 5-stage pipeline diagram below for cycles 100-112, assuming that at cycle 100 the `lw, a0, 0(t1)` instruction is fetched. Assume the loop runs for many iterations. Draw arrows indicating each use of bypassing. Ignore cells shaded in gray.

Cycle	100	101	102	103	104	105	106	107	108	109	110	111	112
IF	lw	lw	add	sw	sw	sw	sw	addi	bnez	lw	xori	lw	lw
DEC		lw	lw	add	add	add	sw	sw	addi	bnez	lw	NOP	lw
EXE			lw	lw	NOP	NOP	add	sw	sw	addi	bnez	NOP	NOP
MEM				lw	lw	NOP	NOP	add	sw	sw	addi	bnez	NOP
WB					lw	lw	NOP	NOP	add	sw	sw	addi	bnez

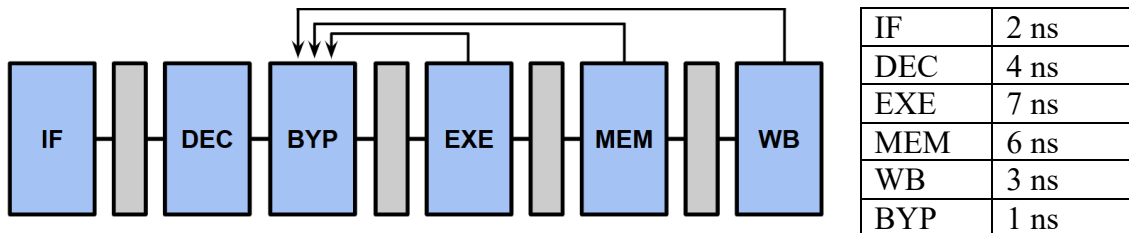
How many cycles does each iteration of the loop take in steady state? For each loop iteration, how many cycles are wasted due to stalls? How many are wasted due to annulments?

Number of cycles per loop iteration: 11

Number of cycles per loop iteration wasted due to stalls: 2

Number of cycles per loop iteration wasted due to annulments: 2

Ben's 5-stage full-bypassing processor has the following propagation delays for each piece of combinational logic:



What is the minimum clock period for this processor assuming ideal registers with ($t_{PD}=0$, $t_{SETUP}=0$)? How long does 1 iteration of the loop take?

Processor minimum clock period (ns): 8 ns (EXE -> BYP)

Time to execute 1 iteration of the loop (ns): 11(8 ns) = 88 ns

Alice wants to help Ben speed up his pipeline. She notices that memory operations are a significant component of the above code, so she tells Ben that merging the EXE and MEM stages will make each loop execute faster because fewer cycles per loop iteration will be used. Ben is skeptical and decides to check this for himself.

Ben creates a 4-stage pipeline (IF, DEC, EXE/MEM, WB), merging the EXE and MEM stages into one pipeline stage. Assume that branches are always predicted not taken (i.e., we speculate that the branch is not taken) and that branch decisions are made in the EXE/MEM stage. Assume that the loop repeats many times and it's currently in the middle of its execution.

(B) (7 points) Fill in the 4-stage pipeline diagram below for cycles 100-112, assuming that at cycle 100 the `lw, a0, 0(t1)` instruction is fetched. Assume the loop runs for many iterations. Draw arrows indicating each use of bypassing. Ignore cells shaded in gray.

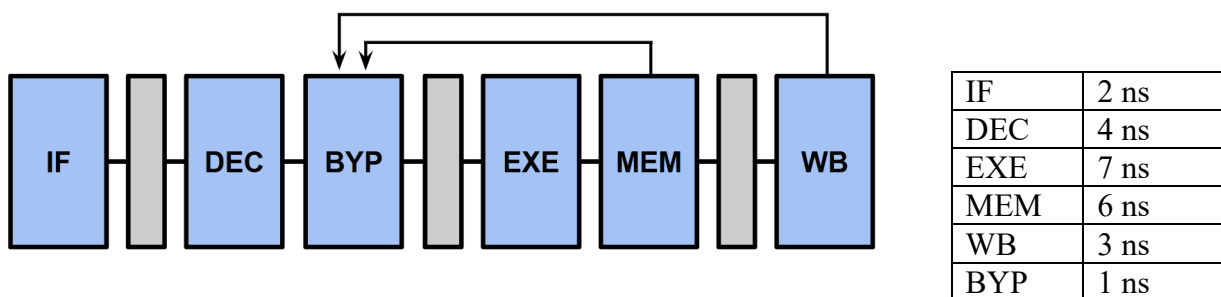
Cycle	100	101	102	103	104	105	106	107	108	109	110	111	112
IF	lw	lw	add	sw	sw	sw	addi	bnez	lw	xori	lw	lw	add
DEC		lw	lw	add	add	sw	sw	addi	bnez	lw	NOP	lw	lw
EXE/ MEM			lw	lw	NOP	add	sw	sw	addi	bnez	NOP	NOP	lw
WB				lw	lw	NOP	add	sw	sw	addi	bnez	NOP	NOP

How many cycles does each iteration of the loop take in steady state? For each loop iteration, how many cycles are wasted due to stalls? How many are wasted due to annulments?

Number of cycles per loop iteration: 10

Number of cycles per loop iteration wasted due to stalls: 1

Number of cycles per loop iteration wasted due to annulments: 2



Was Alice correct? What is the minimum clock period for this processor assuming ideal registers with ($t_{PD}=0$, $t_{SETUP}=0$)? How long does 1 iteration of the loop take?

Processor minimum clock period (ns): 14 ns (EXE+MEM + BYP)

Time to execute 1 iteration of the loop (ns): 10(14 ns) = 140 ns

While testing Alice's theory, Ben noticed that the Fib loop could be rewritten without using the `sw` and `lw` instructions. Thinking this will greatly speed up his pipeline, Ben decides to implement this change.

(C) (5 points) To improve performance, rewrite the Fib code without using `sw` and `lw` instructions. For full credit, minimize the number of cycles per loop iteration.

```
addi t0, zero, 24    // t0 = 24
addi a0, zero, 0      // fib(0) = 0
addi a1, zero, 1      // fib(1) = 1
```

```
fib: _add a2, a0, a1_____
     _mv a0, a1_____
     _mv a1, a2_____
     _addi t0, t0, -1_____
     _bnez t0, fib_____
_____
_____
```

```
mv a0, __a1_____ // a0 <- fib(25)
xori a4, a2, 2      // Unrelated future instructions
ret
```

Ben runs this modified code on the same **5-stage RISC-V processor from part A**. How many cycles per iteration in steady state does the modified code achieve? How long does 1 iteration of the loop take?

Note: You do not need to fill in a pipeline diagram to answer this question, but if you need one, there are blank diagrams at the end of the quiz.

Number of cycles per loop iteration: _____ **7** _____

Time to execute 1 iteration of the loop: _____ **7(8 ns) = 56 ns** _____

End of Quiz 2