1	/16
2	/16
3	/14
4	/18
5	/18
6	/18

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures

Spring 2024

Quiz #2

Name Solutions	*Athena login name*	*Score*

Recitation section

☐ WF 10, 34-302 (Wendy)	☐ WF 2, 34-302 (Catherine)	☐ opt-out
☐ WF 11, 34-302 (Wendy)	☐ WF 3, 34-302 (Catherine)	
☐ WF 12, 34-302 (Adrianna)	☐ WF 12, 35-308 (Shabnam)	
☐ WF 1, 34-302 (Adrianna)	☐ WF 1, 35-308 (Shabnam)	

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)

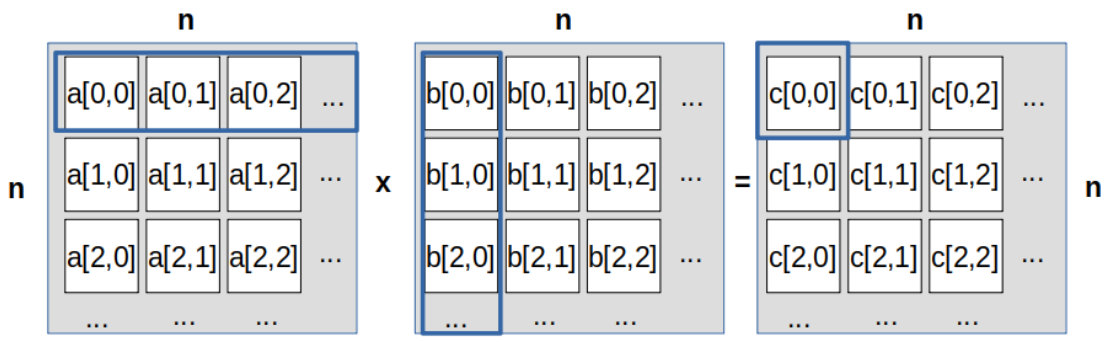
You are given the following Minispec sequential module:

```
typedef struct {Bit#(16) a; Bit#(16) b;} Args;

module Comp;
   Reg#(Bit#(16)) x(1);
   Reg#(Bit#(16)) y(0);

   input Maybe#(Args) in;

   rule step;
      if (isValid(in)) begin
         let args = fromMaybe(?, in);
         x <= args.a;
         y <= args.b;
      end else if (x != 0) begin
         if (x >= y) begin
            x <= x - y;
         end else begin
            x <= y;
            y <= x;
         end
      end
   endrule

   method Maybe#(Bit#(16)) result = (x == 0)? Valid(y) : Invalid;
endmodule
```

(A) (6 points) Fill in the table below with the value of each variable at each cycle.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
in	Invalid	Valid(Args{ a:6 ,b:4})	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
x	1	1	6	2	4	2	0	0
y	0	0	4	4	2	2	2	2
result	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Valid(2)	Valid(2)

Now that you have warmed up, you are asked to design a parameterized matrix multiplication module for matrix product AxB where A and B are square matrices, such that the number of columns and rows in each matrix is *n*. Matrix multiplication is defined as demonstrated in the following diagram:



$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} * b_{k,j}$$

Your module can accept 2 inputs at a time, `row` and `column`. `row` and `column` are `Maybe Vectors` of *n* 32-bit elements. When both `row` and `column` inputs are valid, then your module should compute the dot product of these two vectors and assign the result to the correct location of `temp_matrix` (which is indexed by `row_counter` and `col_counter`) in the code below. After your module computes all the matrix product elements, your module should output a valid `result`, the matrix product C. Assume that all addition (+) and multiplication (*) operations in your matrix multiply use 32-bit inputs and produce 32-bit outputs.

It is important to note that you cannot assume that both `row` and `column` inputs will be valid every cycle.

Your module will receive the pairs of inputs in the following order and you can assume when a valid input pair is received, it follows the following pattern:

Input Pair	Row	Column
0	0	0
1	0	1
2	0	2
…		
n-1	0	n-1
n	1	0
n+1	1	1
…		

(B) (8 points) Complete the Minispec skeleton below to implement matrix multiplication for *n by n* matrix inputs.

```
module MatrixMultiply#(Integer n);

    input Maybe#(Vector#(n, Bit#(32)) row;
    input Maybe#(Vector#(n, Bit#(32)) column;

    Reg#(Bit#(log2(n))) row_counter(0);
    Reg#(Bit#(log2(n))) col_counter(0);
    Reg#(Bit#(1)) done(0);

    Vector#(Vector#(n, Reg#(Bit#(32))) temp_matrix = unpack(0);

    rule tick;

          if (row_counter == (n-1) && col_counter == (n-1)
             && isValid(row) && isValid(column)) done <= 1;
          else done <= 0;

          if( isValid(row) && isValid(column) ) begin

                Vector#(n, Bit#(32)) row_in = ___fromMaybe(?, row)______;

                Vector#(n, Bit#(32)) col_in = ___fromMaybe(?, column)___;

                Bit#(32) total_sum = 0;
                for(Integer k = 0; k < n; k = k + 1) begin

                      __total_sum = total_sum + row_in[k]*col_in[k]_____;

                end

                temp_matrix[row_counter][col_counter] <= __total_sum_____;

                if (col_counter == n - 1) begin

                      col_counter <= ________0______________;

                      if (row_counter == n - 1) begin

                            row_counter <= ________0______________;

                      end else __row_counter <= row_counter + 1_____;

                end else __ col_counter <= col_counter + 1______;

          end
    endrule
```

(C) (2 points) Complete the `result` method which returns a valid matrix C once it has been fully computed and returns Invalid otherwise.

```
    method Maybe#(Vector#(n, Vector#(n, Bit#(32)))) result;
          Vector#(n, Vector#(n, Bit#(32))) ret = unpack(0);

          for (Integer i = 0; i < n; i = i + 1) begin

                 for(Integer j = 0; j < n; j = j + 1) begin

                        ret[i][j] = __temp_matrix[i][j]___;
                 end
          end

          if (__done == 1__) return ____Valid(ret)_____;

          else return ___Invalid_______;

    endmethod

endmodule
```

Problem 2. Arithmetic Pipelines (16 points)

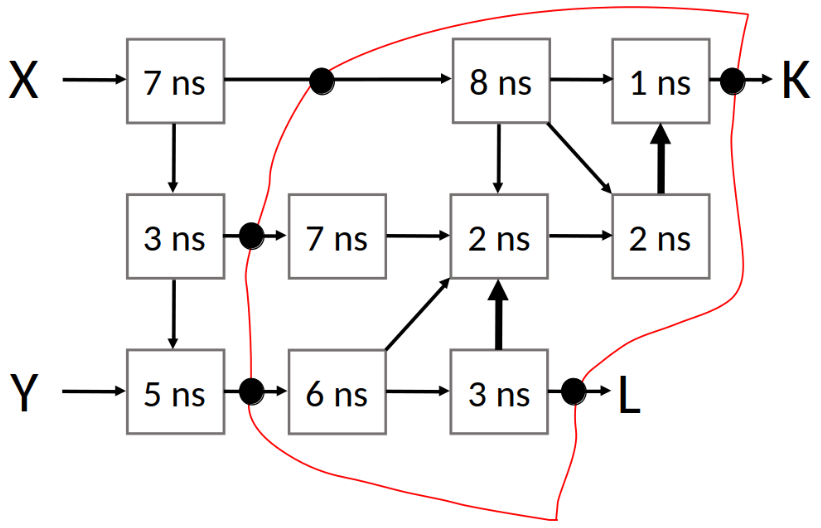
Octavian the octopus accidentally fried his sister's Kelp-o-Meter, but luckily, he was able to buy a working replacement. However, the new device's throughput is much lower than the original's, so Octavian asks you to help him pipeline it. A Kelp-o-Meter (KELP for short) has two inputs, **X** and **Y**, and two outputs, **K** and **L**.

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.

(A) (1 point) Based on the circuit shown in part (B), what is the propagation delay of the whole KELP circuit as-is, without pipelining?

t_{PD} (ns): ___29_____

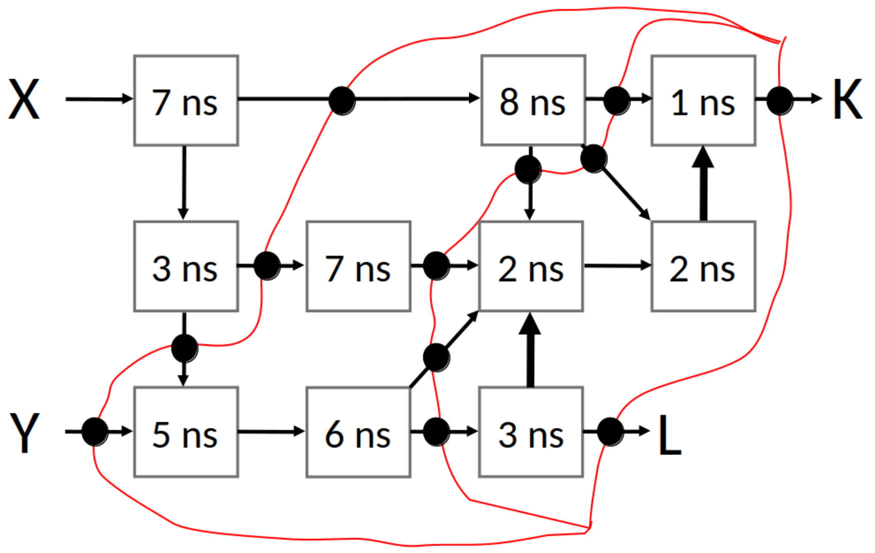
(B) (3 points) Show the **maximum-throughput 2-stage pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit? **Pay close attention to the direction of each arrow.** Show your pipeline contours and each pipeline register. In case you need them, extra copies of the circuit are available at the end of the quiz.



Latency (ns): ____30______

Throughput (ns^{-1}): ____1/15______

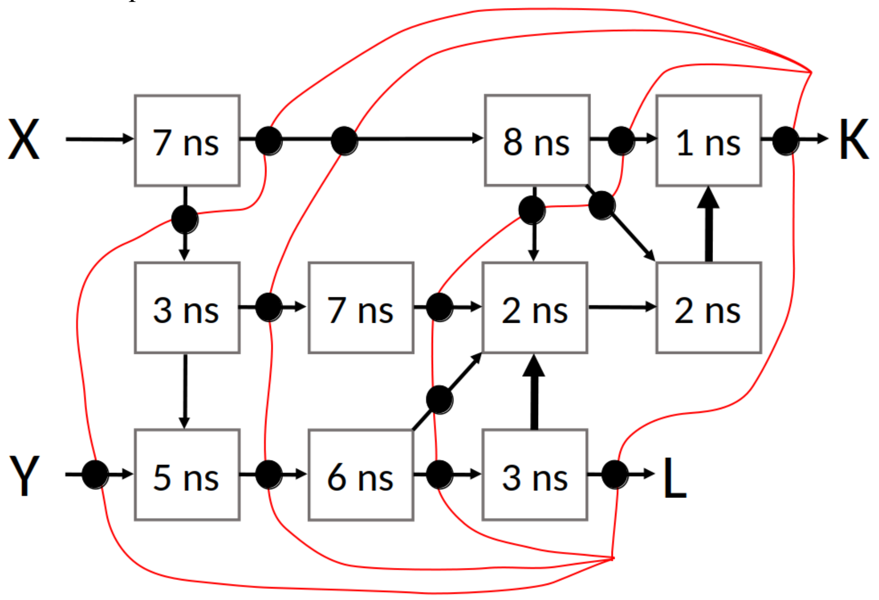
(C) (4 points) Show the **maximum-throughput 3-stage pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit? Show your pipeline contours and each pipeline register. In case you need them, extra copies of the circuit are available at the end of the quiz.



Latency (ns): ___33_______

Throughput (ns^{-1}): ___1/11_______

(D) (4 points) Show the **maximum-throughput pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit? Show your pipeline contours and each pipeline register. In case you need them, extra copies of the circuit are available at the end of the quiz.



Latency (ns): ___32_______

Throughput (ns^{-1}): ___1/8_______

(E) Octavian came up with a brilliant birthday present for his sister: an add-on module that connects to the outputs of her new Kelp-o-Meter. He is considering three different models, CRAB, STAR, and FISH, 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
CRAB	1/6	4
STAR	1/8	2
FISH	1/12	1

(i) (2 points) Octavian thinks maximizing throughput is most important. Which versions of the pipelined KELP module and add-on module should Octavian choose, and what are the resulting latency and throughput? If two combinations have identical throughput, choose the one with better latency.

Module KELP (circle one):

2-stage pipeline **3-stage pipeline** [✓ **Maximum-throughput pipeline**]

Add-on Module (circle one):

CRAB (T = 1/6) [✓ **STAR (T = 1/8)**] **FISH (T = 1/12)**

Latency (ns): ____48______

Throughput (ns^{-1}): ____1/8______

(ii) (2 points) After thinking for a bit, Octavian starts worrying about the latency of the combined modules. To minimize latency, which versions of the pipelined KELP module and add-on module should Octavian choose, and what are the resulting latency and throughput? If two combinations have identical latency, choose the one with better throughput.

Module KELP (circle one):

[✓ **2-stage pipeline**] **3-stage pipeline** **Maximum-throughput pipeline**

Add-on Module (circle one):

CRAB (T = 1/6) **STAR (T = 1/8)** [✓ **FISH (T = 1/12)**]

Latency (ns): ____45______

Throughput (ns^{-1}): ____1/15______

Problem 3. Processor Implementation (14 points)

Reggie Ster has written a program in RISC-V assembly that repeatedly calculates an address and jumps to that address. Reggie's program has this code pattern repeated many times:

```
slli x3, x3, 2
add x2, x4, x3
jalr x1, 0(x2)
```

Reggie notices that the offset for the `jalr` instruction is **always 0** for his program. To make his code more efficient, Reggie decides to combine these 3 instructions into a single *calculate and jump* instruction that can be executed in one cycle. This is the new instruction Reggie wants to add to the RISC-V ISA:

```
calcj rd, rs1, rs2
```

The `calcj` instruction computes an address based on the values in registers `rs1` and `rs2`, then jumps to that address. The instruction also stores `pc+4` to the `rd` register. The behavior of this new instruction is summarized in the code below:

```
reg[rd] <= pc + 4

JT = reg[rs1] + (reg[rs2]<<2)
pc <= {JT[31:1], 1'b0}
```

Reggie has chosen to encode the `calcj` instruction in the following way:

31...25	24...20	19...15	14...12	11...7	6...0
0000001	rs2	rs1	101	rd	0111011

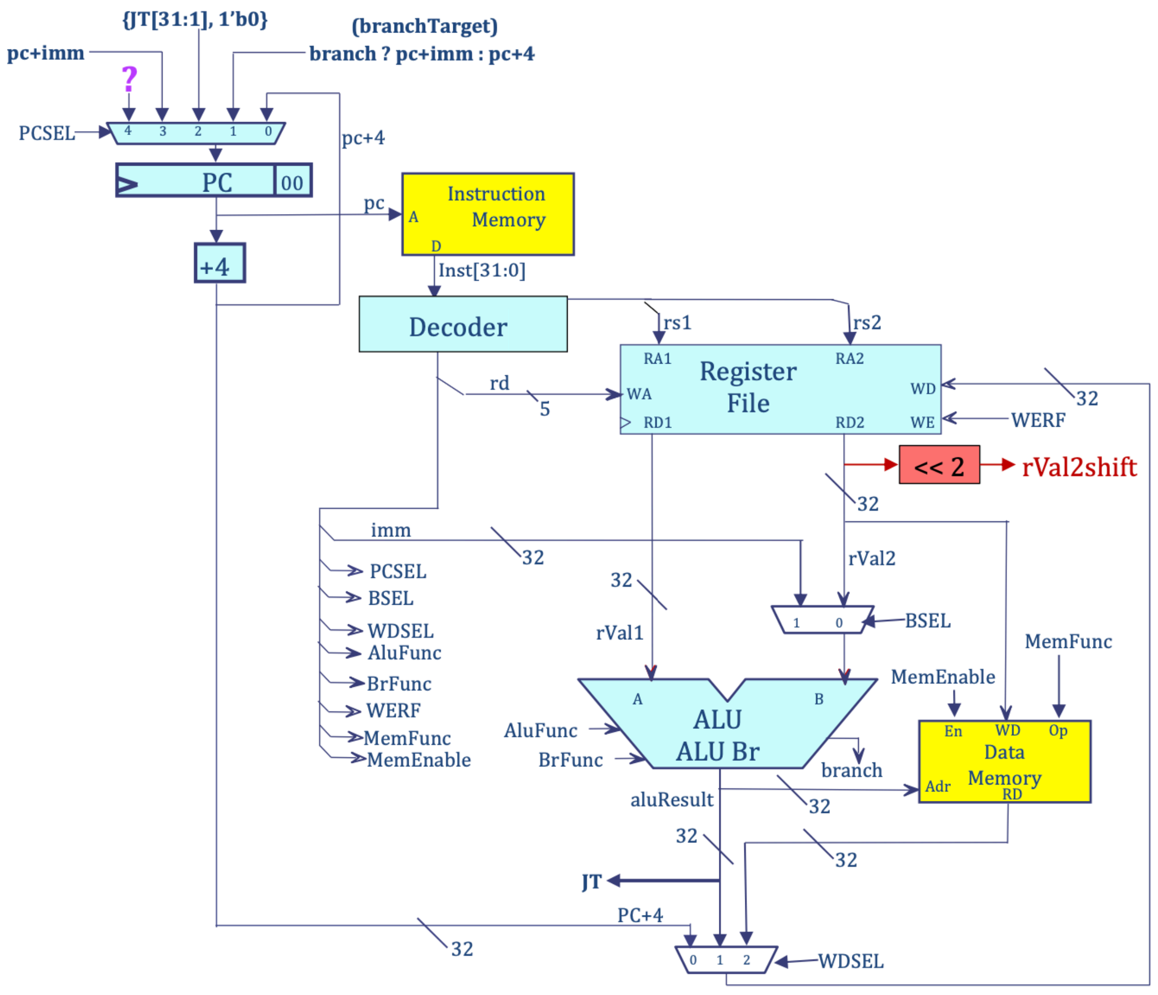
(A) (1 point) Encode the following instruction as a 32-bit binary word. Provide your encoding in hexadecimal notation.

```
calcj x1, x4, x3

0000|001 0|0011 |0010|0 101 |0000|1 011|1011
0x023250BB
```

Encoding in hexadecimal 0x: ____023250BB________

Reggie introduces a new `<<2` module (highlighted below) to the processor diagram from lecture. This module left-shifts the second register value by two, with the shifted output labeled as `rVal2shift`.

Please also note that the **logic for computing branches and jumps** is provided in bold in the diagram below. Help Reggie modify the processor implementation below to support the new *calculate and jump* instruction.



(B) (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 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** is required to make the instruction work properly.

BSEL: **Needs new input (circle one)?** [✓ **YES**] **NO**

New input/Existing control signal: __ rVal2shift ___

WDSEL: **Needs new input (circle one)?** **YES** [✓ **NO**]

New input/Existing control signal: __PC+4 (0)__

(C) (3 points) To support the `calcj` instruction, Reggie decides he wants to add an input 4 to the `PCSEL` mux and connect `aluResult` to this new input signal (i.e., Reggie replaces the question mark, at input 4 of the PCSEL mux, with `aluResult`). The decoder will set `PCSEL = 4` for the `calcj` instruction.

Is there a problem with Reggie's approach? **If yes,** explain the issue and suggest what Reggie should do instead to update the `PC` register correctly. **If no,** explain why Reggie's approach works.

Is Reggie's approach problematic (circle one)? [✓ YES] NO

Explanation:
If the result of the addition is odd, then aluResult will give a PC that is not half-word aligned. Instead of creating a new input and connecting aluResult, Reggie should use the existing input 2 {JT[31:1], 1'b0} (which is essentially aluResult with the last bit cleared)

(D) (5 points) Decide for each of the following control signals what their values should be when executing the `calcj` instruction. If the value of the signal doesn't matter, then put **N/A**. The possible values for each signal are provided below.

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

MemEnable: True, False

WERF: True, False

AluFunc: _____Add__________

BrFunc: _____N/A__________

MemFunc: ___N/A____ **OR** **MemFunc:** __**Lw, Lh, Lhu, Lb, Lbu**_

MemEnable: __**False**__ **MemEnable:** ________**True**________

WERF: _____**True**____

(E) (3 points) Reggie modifies his program slightly so that the repeated code block in his program becomes:

```
slli x3, x3, 2
add x2, x4, x3
jalr x1, 4(x2)
```

Reggie notices that the offset for the `jalr` instruction is now **always 4** for his program. He wants to make another *calculate and jump four* instruction, which would have the following code implementation:

```
reg[rd] <= pc + 4

JT = (reg[rs1] + (reg[rs2]<<2)) + 4
pc <= {JT[31:1], 1'b0}
```

Could we modify **only the control signals** of the processor above (with the additional shifter module) to support this instruction? **Explain your answer**.

Could we implement a calculate and jump four instruction (circle one)? **YES** [✓ **NO**]

Explanation:
The processor with an additional shifter module allows us to support instructions that require a 2-bit left shift and one additional ALU operation. The calculate and jump four instruction would require a 2-bit left shift and 2 additions. This is not possible to implement in our processor without an additional adder module.

Problem 4. Caches (18 points)

Assume that addresses and data words are 32 bits. Consider a **4-way set associative cache** with **64 sets** and a **block size of 4**.

(A) (2 points) Which address bits should the cache use for the cache index, tag field, and block offset. Write [X : X] if no bits are used.

Address bits for byte offset: A[1 : 0]

Address bits for cache index: A[9 : 4]

Address bits for tag field: A[31 : 10]

Address bits for block offset: A[3 : 2]

Now consider a **2-way set associative** cache with **4 sets** and a **block size of 4**. You will use this architecture for parts (B) – (E).

(B) (3 points) How will the following cache parameters change in this new cache relative to the cache in part (A)? **Please circle the best answer. If 'Other', write the change.**

# of cache index bits:

UNCHANGED … +1 … -1 … +2 … -2 … CAN'T TELL… [✓ Other:___-4___]

# of tag field bits:

UNCHANGED … +1 … -1 … +2 … -2 … CAN'T TELL… [✓ Other___4___]

# of block offset bits:

[✓ UNCHANGED] … +1 … -1 … +2 … -2 … CAN'T TELL….Other:________

(C) (8 points) Now analyze the performance of this cache (2-way set associative, 4 sets, block size of 4 words) using the following assembly program, which iterates through array `A` (base address `0x40`) and stores the result of `4*A[i]` into array `B` (base address `0x80`).

```
// x1 = 0 (loop index i)
// x2 = 4 (number of elements in array A)
. = 0x0  // The following code starts at address 0x0
loop:
    slli x3, x1, 2          // convert to byte offset
    lw x4, 0x40(x3)         // load value from A[i]
    slli x4, x4, 2          // x4 = 4 * A[i]
    sw x4, 0x80(x3)         // store 4 * A[i] into B[i]
    addi x1, x1, 1          // increment index
    blt x1, x2, loop        // loop 4 times

    unimp                   // halt, all done
```

Assume the cache is empty before execution of this code (i.e., all valid bits are 0). Assume that the cache uses a least-recently used (LRU) replacement policy, and that **all cache lines in Way 0 are currently the least-recently used**. Fill in, or update, all the known values of the LRU bit, the dirty bit (D), the valid bit (V), the tag, and the data words **after one loop iteration** (after executing the `blt` instruction for the first time). For word fields, fill them in with the opcode if they are instructions (e.g., `blt`) or fill them in with the array element if they are data (e.g., `A[0]`). *You may assume that if V = 0 then D = 0.*

```
// x1 = 0 (loop index i)
// x2 = 4 (number of elements in array A)
. = 0x0     // The following code starts at address 0x0
loop:
    slli x3, x1, 2   // convert to byte offset
    lw x4, 0x40(x3)  // load value from A[i]
    slli x4, x4, 2   // x4 = 4 * A[i]
    sw x4, 0x80(x3)  // store 4 * A[i] into B[i]
    addi x1, x1, 1   // increment index
    blt x1, x2, loop // loop 4 times

    unimp            // halt, all done
```

Index	LRU after one loop iteration
0	0
1	1
2	0
3	0

Way 0 (After one loop iteration)

Index	V	D	Tag	Word 3	Word 2	Word 1	Word 0
0	1	0	0x0	sw	slli	lw	slli
1	1	0	0x0		unimp	blt	addi
2	0	0					
3	0	0					

Way 1 (After one loop iteration)

Index	V	D	Tag	Word 3	Word 2	Word 1	Word 0
0	1	1	0x2	B[3]	B[2]	B[1]	B[0]
1	0	0					
2	0	0					
3	0	0					

(D) (4 points) Fill out the table below with the number of instruction hits, instruction misses, data hits, and data misses for each of the four iterations of the loop.

	Instructions		**Data**	
	Hits	**Misses**	**Hits**	**Misses**
First loop iteration	4	2	0	2
Second loop iteration	6	0	0	2
Third loop iteration	6	0	0	2
Fourth loop iteration	6	0	0	2

(E) (1 point) What is the hit ratio for the execution of the **four loop iterations** (Note: do not include execution of the `unimp` instruction)? You may leave your answer as a fraction.

Hit Ratio: ____22/32____

Problem 5. Pipelined Processors (18 points)

Ben Bitdiddle writes the following loop in RISC-V assembly to multiply elements of an array by 3. The array is of size `n` and is stored in memory beginning at address `0x500`.

```
    // a0 = n = 100
    // a1 = 0 = loop index i
loop:
    slli a2, a1, 2          // multiply index by 4
    lw a3, 0x500(a2)
    slli a4, a3, 1
    add a4, a4, a3
    sw a4, 0x500(a2)        // a[i] = 3 * a[i]
    addi a1, a1, 1          // increment loop index i
    blt a1, a0, loop
    ori a0, a2, 2           // some instructions following the loop
    xori a2, a0, 4
    and a3, a2, a1
```

Ben runs this on 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**.
- Branches and jumps are resolved in the EXE/MEM stage.
- Full bypassing is implemented.

(A) (7 points) Fill in the pipeline diagram below for cycles 100-112, assuming that at cycle 100 the `slli a2, a1, 2` instruction is fetched. Assume the loop runs for many iterations. **Draw arrows indicating each use of bypassing.** Ignore cells shaded in gray.

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

(B) (2 points) How many cycles does each iteration of the loop take? 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__________

(C) (4 points) Suppose Ben now modifies his processor so that branches are always **predicted to be taken**. Assume that everything else about the processor remains unchanged. Fill in the pipeline diagrams below assuming that at cycle 200 the `addi a1, a1, 1` is fetched, and the branch is taken. **Draw arrows indicating each use of bypassing.** Ignore cells shaded in gray.

	200	201	202	203	204	205	206	207	208	209
IF	addi	blt	slli	lw	slli	add	add	sw	addi	blt
DEC	▨	addi	blt	slli	lw	slli	slli	add	sw	addi
EXE/MEM	▨	▨	addi ↑	blt	slli ↑	lw	NOP	slli ↑	add ↑	sw
WB	▨	▨	▨	addi ↑	blt	slli	lw ↑	NOP	slli	add

(D) (2 points) How many cycles does the execution of the loop take when branches are predicted to be taken? 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: _________8_________

Number of cycles per loop iteration wasted due to stalls: _________1_________

Number of cycles per loop iteration wasted due to annulments: _________0_________

(E) (3 points) Ben decides to continue using the version of his processor that predicts that branches are always taken. However, for cost saving reasons, he needs to remove one of the bypass paths from his processor. Should he choose to remove the EXE/MEM to DEC bypass or the WB to DEC bypass? With the selected bypass path removed, how many cycles does each iteration of the loop take? For each loop iteration, how many cycles are wasted due to stalls? How many are wasted due to annulments?

Bypass path to remove (circle one): **EXE/MEM to DEC** [✓ **WB to DEC**]

Num cycles per loop iteration after removing bypass path:
_________10_________

Num cycles per loop iteration wasted due to stalls after removing bypass path:
_________3_________

Num cycles per loop iteration wasted due to annulments after removing bypass path:
_________0_________

Problem 6. Pipelined Processor Performance (18 points)

The following loop sums up the elements of an array:

```
loop:   lw a1, 0(a2)
        add a0, a1, a0
        addi a2, a2, 4
        blt a2, a3, loop
        xor a1, a4, a5          // some code after the loop
        sub sp, sp, a6
        ret
```

(A) (3 points) Assume a standard 5-stage RISC-V pipelined processor with **full bypassing**. In steady state, how many cycles does each iteration of the loop take? Note that all branches are **predicted not taken**.

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.

Instructions per loop iteration: ______4______

Cycles per loop iteration lost to stalls: ______2______

Cycles per loop iteration lost to annulments: ______2______

Cycles per loop iteration: ______8______

(B) (2 points) Reorder the instructions in the loop to improve performance. How many cycles per iteration does your code achieve?

```
loop:   ___ lw a1, 0(a2)_______________

        ____ addi a2, a2, 4_____________

        ____ add a0, a1, a0_____________

        blt a2, a3, loop
```

(There are also 6-cycle solutions if you reorder across loop iterations, but the code for those is more complex.)

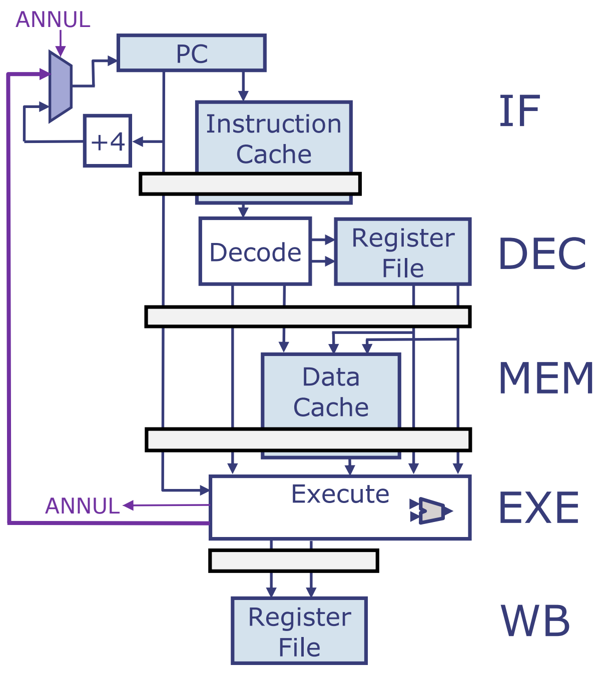
Cycles per loop iteration with reoredered code: ______7______

Ben Bitdiddle notices that it's common for code to have load instructions for values that are used by a single ALU instruction, like the `lw` and `add` instruction pair in the previous loop. He proposes to change the RISC-V ISA to support ALU instructions where the first operand comes from memory instead of a register. These instructions have the form:

```
    op rd, (rs1), rs2:    Reg[rd] ← Mem[Reg[rs1]] op Reg[rs2]
```

With this ISA change, the previous loop can be rewritten as follows, saving one instruction:

```
loop:   add a0, (a2), a0
        addi a2, a2, 4
        blt a2, a3, loop
        xor a1, a4, a5
        sub sp, sp, a6
        ret
```



To support this new instruction type, Ben implements the 5-stage pipeline shown here. The key difference with the standard 5-stage pipeline is that the MEM stage comes before the EXE stage, not after. This allows these new ALU instructions to load one of their operands from memory.

As in the classic 5-stage pipeline, branches and jumps are resolved in the EXE stage (but note that this is now the fourth stage in the pipeline, not the third one).

(C) (6 points) Assume that this pipeline implements the same bypass paths as the classic 5-stage pipeline: values can be bypassed **from the MEM, EXE, and WB stages to the DEC stage**. Analyze the performance of the loop above. Fill out the pipeline diagram below for the first 10 cycles and calculate the number of cycles this processor takes to execute one iteration of the above loop. Fill in any stalled/annulled stages with NOPs and clearly show all uses of the bypass paths using arrows. Ignore cells shaded in gray.

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
IF	add	addi	blt	xor	xor	sub	ret	add	addi	blt
DEC		add	addi	blt	blt	xor	sub	NOP	add	addi
MEM			add	addi	NOP	blt	xor	NOP	NOP	add
EXE				add	addi	NOP	blt	NOP	NOP	NOP
WB					add	addi	NOP	blt	NOP	NOP

Cycles per loop iteration: ______7_______

(D) (4 points) Let's consider adding even more bypass paths: assume that values can be bypassed **from the MEM, EXE, and WB stages to the DEC stage (like before), and from the EXE stage to the MEM stage**. Bypassing from EXE to MEM lets us have back-to-back dependences among ALU instructions without stalls, like in the classic pipeline, even though this pipeline has an extra stage (MEM) between DEC and EXE. The bypass paths to MEM work as follows: if the instruction in DEC reads a value that will be produced by an ALU instruction that is currently in MEM, the instruction in DEC does not stall and instead relies on the EXE→MEM bypass path to provide this value on the next cycle.

Analyze the performance of the loop above. Fill out the pipeline diagram below for the first 10 cycles and calculate the number of cycles this processor takes to execute one iteration of the loop. Fill in any stalled/annulled stages with NOPs and clearly show all uses of the bypass paths using arrows. Ignore cells shaded in gray.

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
IF	`add`	`addi`	`blt`	`xor`	`sub`	`ret`	`add`	`addi`	`blt`	`xor`
DEC		`add`	`addi`	`blt`	`xor`	`sub`	`NOP`	`add`	`addi`	`blt`
MEM			`add`	`addi`	`blt` $\uparrow$	`xor`	`NOP`	`NOP`	`add`	`addi`
EXE				`add`	`addi`	`blt`	`NOP`	`NOP`	`NOP`	`add`
WB					`add`	`addi`	`blt`	`NOP`	`NOP`	`NOP`

Cycles per loop iteration: ______6_______

(E) (3 points) In this pipeline, branches and jumps are resolved in a later stage than memory accesses. Is it safe to use speculation to resolve control hazards? If so, explain why. Otherwise, give a code sequence that produces incorrect behavior on mis-speculation and explain why the code sequence misbehaves. Include any assumptions you are making in your explanation.

This pipeline still produces correct execution, but by a hair. The key constraint is that instructions cannot modify architectural state until they are non-speculative. Because memory is placed before branch resolution, the worst-case sequence is a branch or jump followed by a store instruction. In this case, the store instruction can be annulled while it is in MEM and before it writes to the data cache. If EXE happened one cycle later, this would not be possible.

Also accepting solutions that say it doesn't work if you assume that the sw can alter state in the MEM stage before being annulled.

END OF QUIZ 2!