1	/16
2	/16
3	/17
4	/19
5	/19
6	/13

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.191 Computation Structures
Spring 2023

Quiz #2

Name Solutions	*Athena login name*	*Score*

Recitation section

☐ WF 10, 34-302 (Alexandra)
☐ WF 11, 34-302 (Alexandra)
☐ WF 12, 34-302(Georgia)
☐ WF 1, 34-302 (Georgia)
☐ WF 2, 34-302 (Boom)
☐ WF 3, 34-302 (Boom)
☐ WF 12, 35-308 (Keshav)
☐ WF 1, 35-308 (Keshav)
☐ 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 (Stop-and-wait protocol) (16 points)

A Sender is trying to send a sequence of data to a Receiver by sending a signal through the air. Usually, data from the Sender reaches the Receiver in a finite amount of time. However, sometimes, the data might take longer to transmit or get completely lost in the air.

To ensure that the Receiver receives and processes all the data in-order, the Sender and Receiver communicate using a variant of a **Stop-and-wait protocol**.

The Sender sends a sequence of data packets. As shown in the **DataPacket** struct below, each data packet contains a 32-bit word of data, and an index, `idx`, that identifies the position of the packet in the sequence. The index begins at 0 and increments by 1 for every new data word sent. Upon receiving a DataPacket, the Receiver sends back an **AckPacket** to signal that they have successfully received the data. The structure of DataPacket and AckPacket is defined as follows:

```
typedef Bit#(32) Word;

typedef struct {
    Word idx;   // index of this data packet
    Word data;  // the content of the data
} DataPacket;

typedef struct {
    Word idx;   //  index of the data packet to acknowledge
} AckPacket;
```

(A) (8 points) Complete the implementation of a Sender module in Minispec, which sends a fixed sequence of 32 data words. These words are specified as a `Vector#(32, Word) data`. To send a message according to the stop-and-wait protocol, the Sender should:

- Iterate through the `data` vector from index 0 up until index 31.
- For each data index `i`,
    1. Construct a DataPacket that contains `data[i]` and whose index field is `i`.
    2. Send the DataPacket right away.
    3. Retransmit the same DataPacket every 3 cycles until it receives the AckPacket with **same index field** as the DataPacket index that the Sender sent.
    4. Once the Sender receives the AckPacket with the same index, it moves on to send DataPacket corresponding to the next index.
- After receiving an AckPacket for index 31, set `finished` to True to stop sending data.

Fill in the Minispec code for the Sender module below so that it conforms to the stop-and-wait protocol described above. You may use all Minispec operators, including +, -, *, /, and %.

```
// “data” stores all the data to be sent
module Sender(Vector#(32, Word) data);

    // AckPacket Received by the sender each cycle
    input Maybe#(AckPacket)  in_ACK default = Invalid;

    Reg#(Word) send_idx(0);
    Reg#(Word) timer(0);
    Reg#(Bool) finished(False);

    rule tick;
        if(isValid(in_ACK)) begin
            let ack_idx = _ fromMaybe(?, in_ACK).idx ______;

            if (___ack_idx == send_idx __________) begin

                send_idx <= ___ send_idx + 1_____________;

                finished <= __ send_idx >= 31;  or send_idx == 31;

                timer <= 0;
            end
        end else begin
            timer <= __(timer == 0)? 2: timer-1; or (timer + 1)% 3;
        end

    endrule

    // DataPacket to transmit each cycle
    method Maybe#(DataPacket) out_data;
        return (timer == 0 && !finished)? Valid(DataPacket{

            idx : ___ send_idx _____________________________,

            data : ___ data[send_idx]_____________________
        }) : Invalid;
    endmethod

endmodule
```

The Receiver module is defined with the following Minispec code. Note that while packets are sent in order, they are not necessarily received in order.

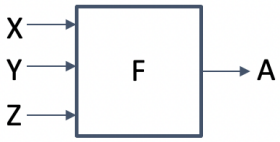
```
module Receiver;

    // DataPacket received by the receiver each cycle
    input Maybe#(DataPacket) in_data default = Invalid;

    Reg#(Maybe#(AckPacket)) to_send(Invalid);
    Reg#(Word) expected_idx(0);

    rule tick;
        Maybe#(AckPacket) packet_to_send = Invalid;
        if(isValid(in_data)) begin
            // Receiver receives a DataPacket
            let in_idx = fromMaybe(?, in_data).idx;
            if (in_idx <= expected_idx) begin
                packet_to_send = Valid(AckPacket{idx : in_idx});
                if (in_idx == expected_idx) begin
                    // process(fromMaybe(?, in_data)).data somehow
                    expected_idx <= expected_idx + 1;
                end
            end
        end
        to_send <= packet_to_send;
    endrule

    // AckPacket to transmit each cycle
    method Maybe#(AckPacket) out_ACK = to_send;
endmodule
```

(B) (8 points) Fill in the timing chart below with the register values and outputs for the first 9 cycles of the Receiver module. If the value is not known, write a "?" in the slot.

Cycle	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
isValid(in_data)	False	True	False	True	True	True	False	True	True
fromMaybe(?,in_data).idx	?	0	?	0	1	0	?	2	1
expected_idx	0	0	1	1	1	2	2	2	3
isValid(to_send)	False	False	True	False	True	True	True	False	True
fromMaybe(?,to_send).idx	?	?	0	?	0	1	0	?	2

Problem 2. Arithmetic Pipelines (16 points)

You are given a module named "F." This module has three inputs, **X**, **Y** and **Z**, and one output, **A**. You are told that the circuit functions, but its throughput is too low. You decide to pipeline the circuit.

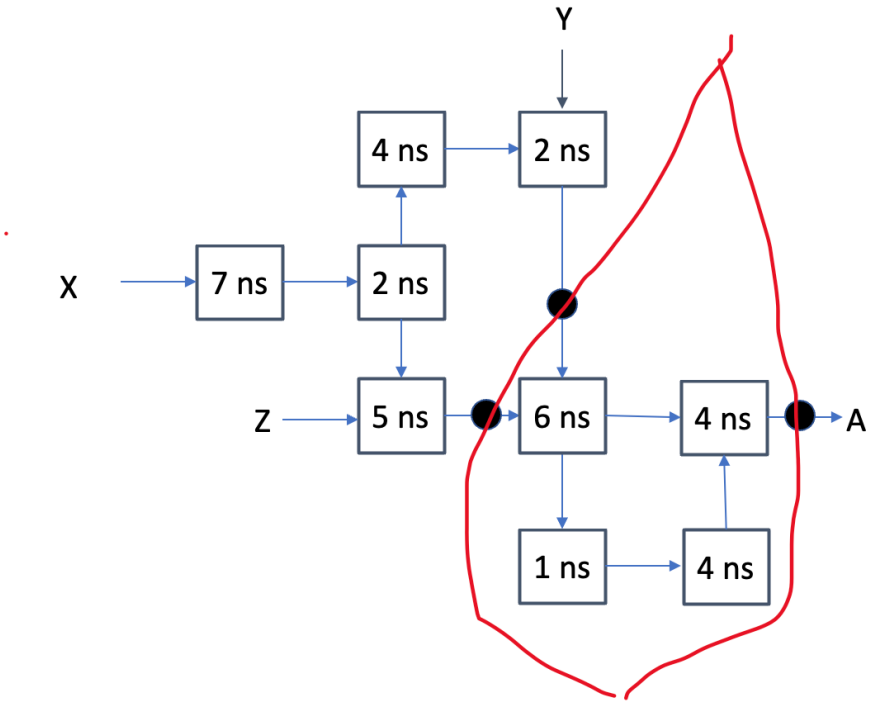


For each of the questions below, please create a valid K-stage pipeline of the given circuit. Each component in the circuit is annotated with its propagation delay in nanoseconds. **Show your pipelining contours** and place large black circles (●) on the signal arrows to **indicate the placement of pipeline 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 circuit as-is without pipelining?

t_PD (ns): ___30_____

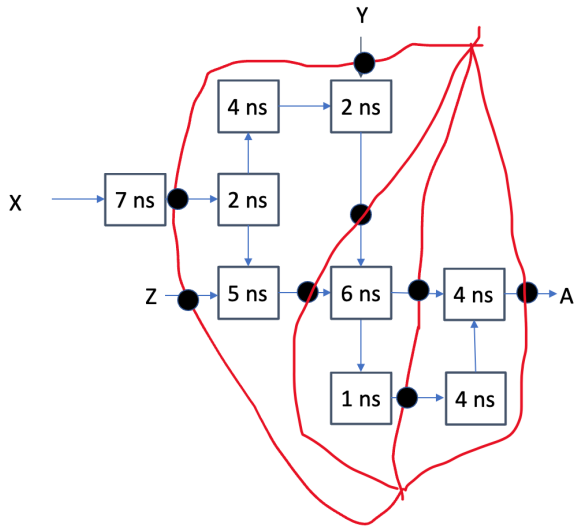
(B) (4 points) Show a **maximum-throughput 2-stage pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit? Pay close attention to the direction of each arrow.



Latency (ns): _____30_____

Throughput (ns^{-1}): _____1/15_____

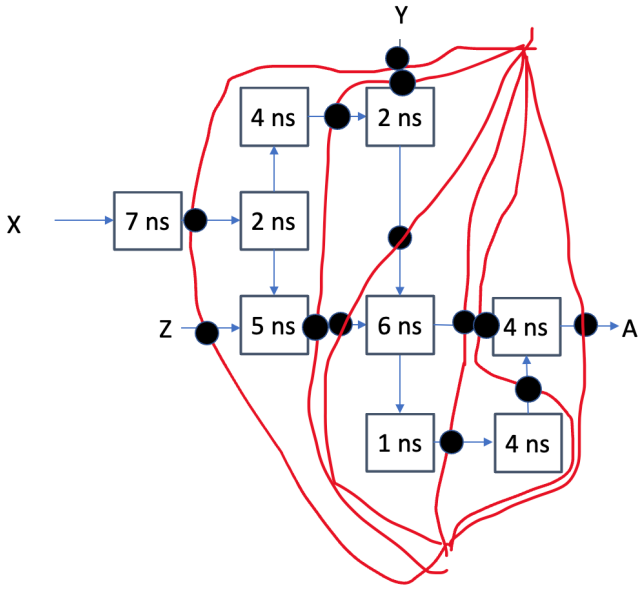
(C) (4 points) Show a **maximum-throughput 4-stage pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit?



Latency (ns): _____32_____

Throughput (ns^{-1}): _____1/8_____

(D) (4 points) Show a **maximum-throughput pipeline** that uses a minimum number of pipeline stages. What are the latency and throughput of the resulting circuit?



Latency (ns): _____42_____

Throughput (ns^{-1}): ____1/7_____

(E) (3 points) Now, we want to connect our max throughput Module F (from part D) to another module that does additional operations on A, the output of F. We have three separate pipelined implementations, J, K and L, that implement the same operations but with different throughputs and number of pipeline stages.

Module	Throughput (ns^{-1})	Number of Pipeline Stages
J	1/6	3
K	1/7	2
L	1/8	1

When connecting the Module F to one of Module J, K, or L, you are trying to maximize throughput. If two options have equal throughput, as a secondary objective you want to minimize latency. Which module would you append on the output of F, Module J, K, or L and why did you select that one? What would the throughput and total latency of the combined device be?

Module (circle one): **J (T = 1/6)** **K (T = 1/7)** **L (T = 1/8)**

Throughput (ns^{-1}): _____**1/7**_____

Total Latency (ns):_____ **56**_____

Why did you choose that module?

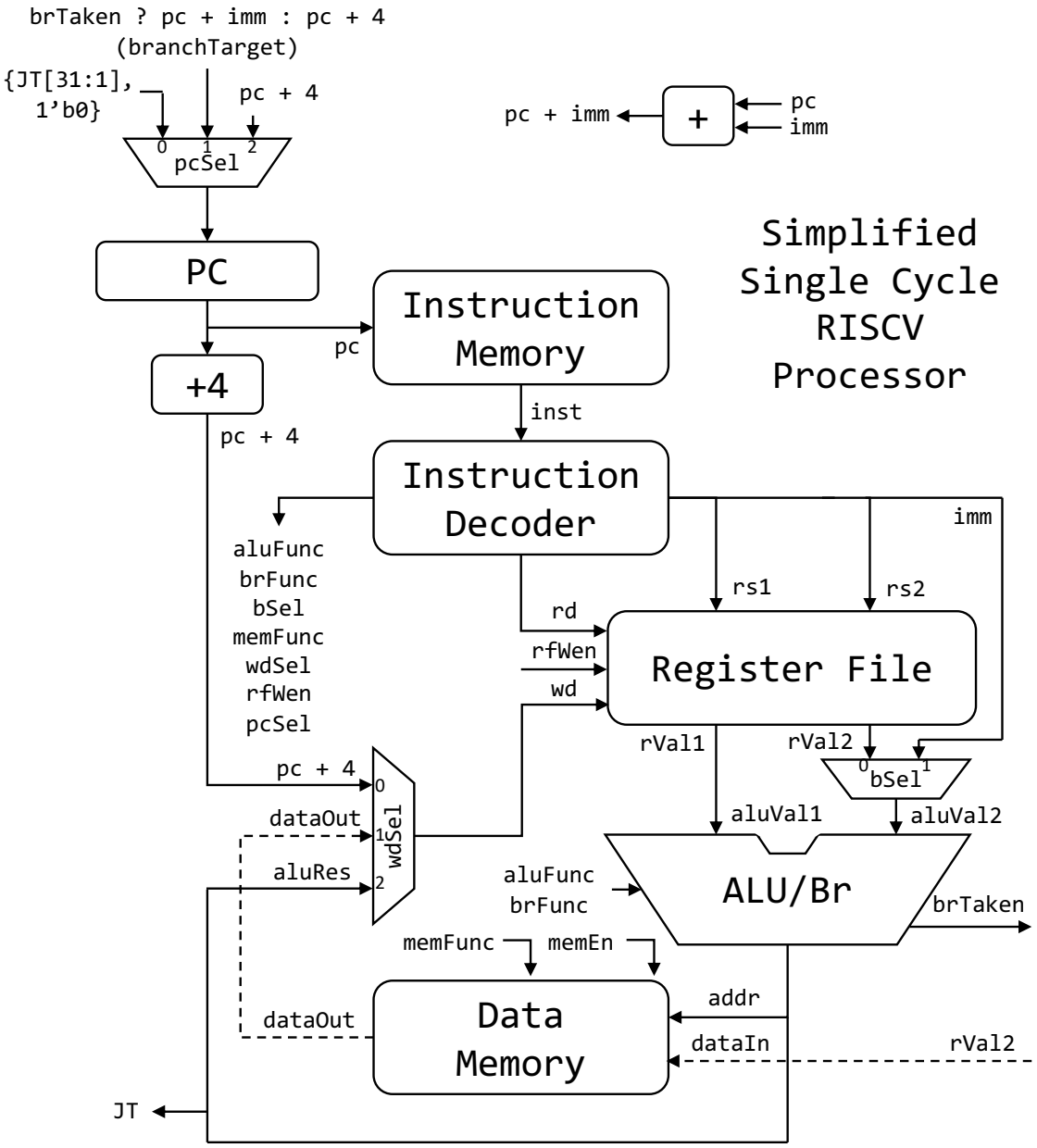
Explanation:

The highest possible throughput for Module F is 1/7 because we can't break up the block with propagation delay 7, which means that there is no point in picking Module J, which has the best throughput of the 3 modules. Instead, we want to pick Module K because it keeps the 1/7 throughput without using any unnecessary registers like picking Module J would. We do not pick Module L because that would worsen the throughput of the combined device.

Problem 3: All right then. Keep your secrets. (17 points)

Cafecafe likes to be updated on all the latest gossip going around in her high school. However, she trusts no one with her secrets and would like to build a processor that will only give out her secret value to a user that knows her secret passcode. She figures she needs a register that holds her secret value, and some way to transfer the contents of that secret register into another, user-accessible general-purpose register when needed. *Let us help her build such a processor.*

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 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, *if extended in later parts*)
`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, *if extended in later parts*)
`rfWen`	Register File Write Enable	True/False
`pcSel`	Next PC Select	0 (jumpTarget), 1 (branchTarget), 2 (pc + 4), 3, 4, 5… (others, *if extended in later parts*)

(A) (2 points) As a warmup, complete the table on the right with what the decoded control signals should be for the `jalr x7, 0x10(x9)` instruction shown. 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)

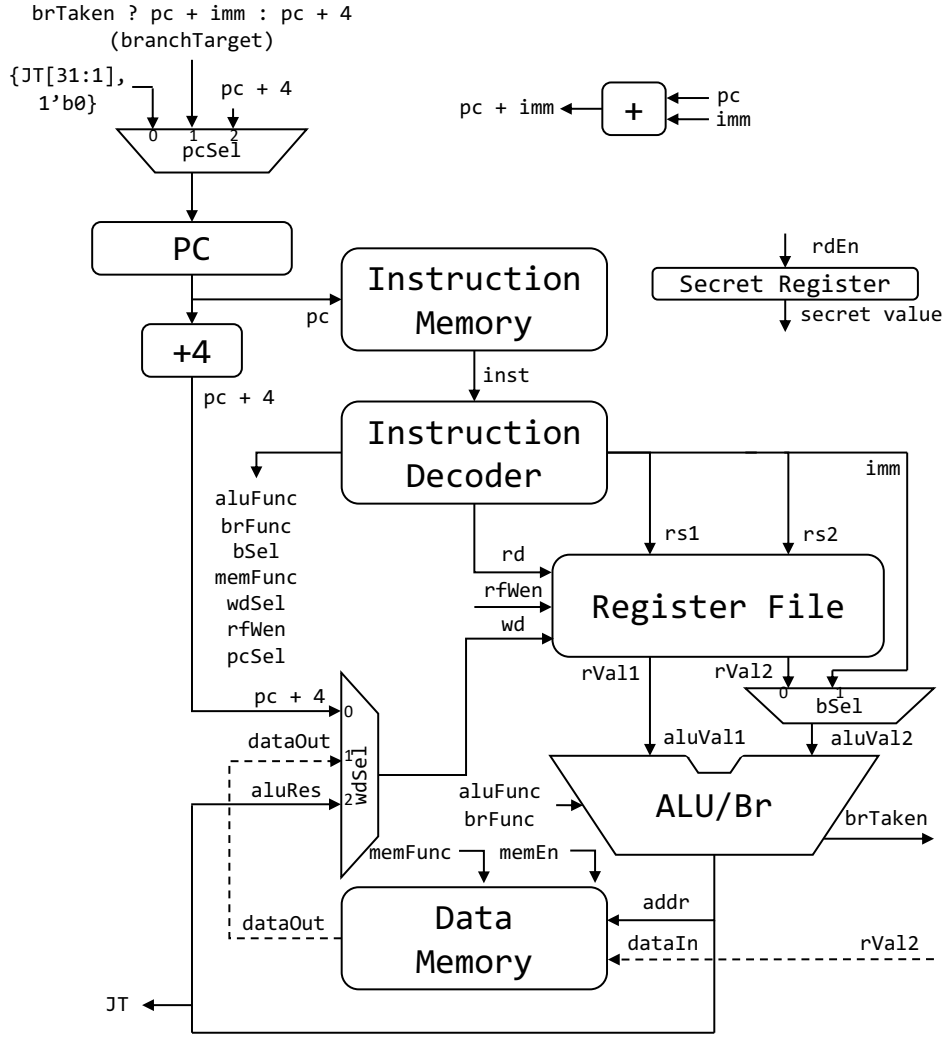
`jalr x7, 0x10(x9)`

Field	Value
`imm`	0x10
`rs1`	9
`rs2`	?
`rd`	7
`aluFunc`	Add
`brFunc`	?
`bSel`	1 (imm)
`memFunc`	?
`memEn`	False
`wdSel`	0 (pc + 4)
`rfWen`	True
`pcSel`	0 (jumpTarget)

(B) (3 points) Next, let us add support to be able to read the secret register that Cafecafe has added to the processor. Her criteria for being able to access this register is simple. One uses the following instruction (load secret) in order to access it.

```
lsecret rd, rs1; // reg[rd] <= (reg[rs1] == 0xcafecafe) ? secret value : 0
```

If the register `rs1` contains the value `0xcafecafe` (this being the secret passcode) when this instruction is executed, then the register `rd` will get the secret value written to it. Otherwise, a zero is written to it. The secret register that Cafecafe has added to her processor works as follows: if its `rdEn` input is asserted, then it outputs the secret value, otherwise it outputs a zero.



Connect the wires connected to Secret Register to appropriate places on the above diagram. You cannot add additional hardware, other than wires and constants, to this processor. Thus, you need to find a way to use the existing comparator to perform the comparison with `0xcafecafe`. The multiplexers have been widened for your convenience. If you choose to add additional inputs to multiplexers, indicate so clearly in part (C) if they are to be selected by the multiplexer.
Connect brTaken to rdEn, connect secret value to input 3 of wdSel, and make a constant input to bSel 0xcafecafe.

(C) (3 points) What does the decoder need to output for the following fields when it sees an `lsecret` instruction shown below?

`lsecret x6, x9`

Field	Value
imm	?
rs1	9
rs2	?
rd	6
aluFunc	?
brFunc	Eq
bSel	2 (constant 0xcafecafe)
memFunc	?
memEn	False
wdSel	3 (secret value)
rfWen	True
pcSel	2 (pc + 4)

(D) (6 points) Uh oh! Cafecafe's gossip secrets are already outdated. She wants to modify her processor so that she can update the secret value by supplying the correct password. She intends to use the following instruction (store secret) to update the secret value.

```
ssecret rd, rs1, rs2;

// authenticated = (reg[rs1] == 0xcafecafe);
// if (authenticated) secret value <= reg[rs2];
// red[rd] <= authenticated;
```

If the register `rs1` contains the value `0xcafecafe`, `ssecret` updates the secret value to the value of register `rs2`. Register `rd` stores the result of this operation: 1 if successful, 0 if failed. The secret register now works as follows: if its rdEn input is asserted, then it outputs the secret value, otherwise it outputs a zero (as before). If wrEn is asserted, the register updates its secret value to the `secret value to write` input.

Do we need to output an additional field/signal from the instruction decoder? If yes, why did lsecret not need this additional field? If no, how can the processor stop undesirable values from being written to the secret value register when the instruction being executed isn't ssecret? *You may want to implement the necessary hardware changes before answering these questions.*

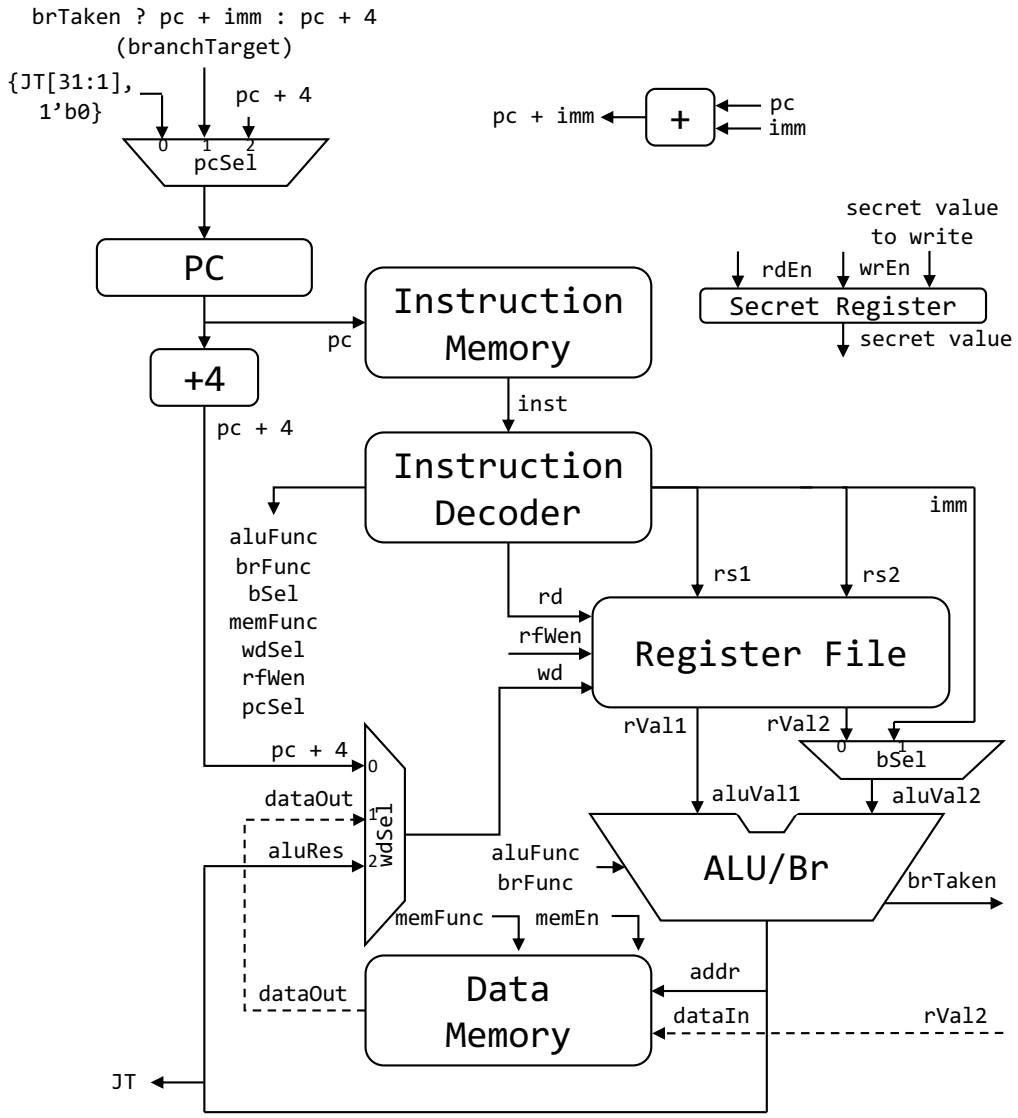
Do we need an additional field? Circle: YES NO

If yes, provide a descriptive name for the field:____ secretEn_____

Explanation:
Yes, we do need an additional field because the wrEn signal needs to be (secretEn && brTaken) to stop the secret value from being updated if the instruction is not ssecret or if the comparison failed. lsecret did not need this field because the decoder can simply set wdSel to something else or rfWen to False to stop the secret value from being written to the register file.

Next, using no more than one 2-input (1-bit wide) additional AND gate, wire up the two new inputs of the secret register (wrEn and secret value to write) into the processor diagram. **Make sure to also show all your changes from part (B).** If you add any additional inputs to a multiplexer, make sure to indicate them clearly in part (E) if they are to be selected by the multiplexer.

In addition to the previous connections, also need to connect wrEn to (secretEn && brTaken). Connect brTaken to yet another input of wdSel, input 4. Connect rVal2 to secret value to write.



(E) (3 points) Finally, provide the field values that the decoder needs to generate when it encounters the following instruction. **Write your new field name and field value in the blank row if you chose yes for part (D).**

`ssecret x14, x12, x8`

Field	Value
`imm`	?
`rs1`	12
`rs2`	8
`rd`	14
`aluFunc`	?
`brFunc`	Eq
`bSel`	2 (constant 0xcafecafe)
`memFunc`	?
`memEn`	False
`wdSel`	4 (brTaken)
`rfWen`	True
`pcSel`	2 (pc + 4)
`secretEn`	True

Problem 4: Fun with Caches (19 points)

Consider a 2-way set associative cache with 4 sets and a block size of 4. Assume that addresses and data words are 32 bits.

(A) (3 points) To ensure the best cache performance, which address bits should be used for the block offset, the cache index, and the tag field?

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

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

Address bits used for cache index: A[5 : 4]

Address bits used for tag field: A[[31 : 6]

We want to analyze the performance of this 2-way set associative cache on the following assembly program which computes the average of two arrays, `A` and `B`, and stores the results in a third array, `C` so that `C[i] = (A[i] + B[i])/2`. Each array has a total of 8 elements. The base address of array `A` is `0x1000`, of array `B` is `0x2040`, and of array `C` is `0x3020`.

```
. = 0x100 // The following code starts at address 0x100

// Assume the following registers are initialized:
// x1=0 (loop index)
// x2=8 (number of array elements)
// x3=0x1000 (base address of array A)
// x4=0x2040 (base address of array B)
// x5=0x3020 (base address of array C)

loop:
   lw x6, 0(x3)     // x6 = A[i]
   lw x7, 0(x4)     // x7 = B[i]
   add x8, x6, x7   // x8 = A[i] + B[i]
   srli x8, x8, 1   // x8 = (A[i] + B[i])/2
   sw x8, 0(x5)     // C[i] = (A[i] + B[i])/2
   addi x3, x3, 4   // x3 = address of next element of A
   addi x4, x4, 4   // x4 = address of next element of B
   addi x5, x5, 4   // x5 = address of next element of C
   addi x1, x1, 1   // increment loop index
   blt x1, x2, loop // repeat loop until all elements are averaged
```

(B) (8 points) 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., B[2]).

The loop code is repeated here for your convenience:

```
. = 0x100 // The following code starts at address 0x100
loop:
   lw x6, 0(x3)     // x6 = A[i]
   lw x7, 0(x4)     // x7 = B[i]
   add x8, x6, x7   // x8 = A[i] + B[i]
   srli x8, x8, 1   // x8 = (A[i] + B[i])/2
   sw x8, 0(x5)     // C[i] = (A[i] + B[i])/2
   addi x3, x3, 4   // x3 = address of next element of A
   addi x4, x4, 4   // x4 = address of next element of B
   addi x5, x5, 4   // x5 = address of next element of C
   addi x1, x1, 1   // increment loop index
   blt x1, x2, loop // repeat loop until all elements are averaged
```

LRU
1
1
0
0

Way 0

D	V	Tag	Word 3	Word 2	Word 1	Word 0
0	1	0x4	srli	add	lw	lw
0	1	0x4	addi	addi	addi	sw
1	1	0xC0	C[3]	C[2]	C[1]	C[0]
	0					

Way 1

D	V	Tag	Word 3	Word 2	Word 1	Word 0
0	1	0x81	B[3]	B[2]	B[1]	B[0]
	0					
0	1	0x4			blt	addi
	0					

(C) (4 points) During the execution of the **first iteration of the loop**, how many instruction hits and instruction misses occurred, and how many data hits and data misses occurred?

Number of Instruction Hits in First Loop Iteration:_____ **7**_____

Number of Instruction Misses in First Loop Iteration:_____ **3**_____

Number of Data Hits in First Loop Iteration:_____ **0**_____

Number of Data Misses in First Loop Iteration:_____ **3**_____

(D) (4 points) During the execution of the **second iteration of the loop**, how many instruction hits and instruction misses occurred, and how many data hits and data misses occurred?

Number of Instruction Hits in Second Loop Iteration:_____ **10**_____

Number of Instruction Misses in Second Loop Iteration:_____ **0**_____

Number of Data Hits in Second Loop Iteration:____ **1**_____

Number of Data Misses in Second Loop Iteration:____ **2**_____

Problem 5. Fake it till you write back (19 points)

Ben Bitdiddle writes the following loop in RISC-V assembly to sum the elements of an array:

```
loop: lw a1, 0(a2)
      add a0, a0, a1
      addi a1, a1, 4
      blt a1, a3, loop
      slli a0, a0, 2 # some instructions following the loop
      xori a0, a0, 4
      and a0, a0, a1
```

Ben runs this on a standard 5-stage pipeline (IF, DEC, EXE, MEM, WB). In this pipeline:

- Branches are predicted not taken.
- Branches and jumps are resolved in the EXE stage.
- Bypassing is done to the end of the DEC stage. Full bypassing is implemented.
- The data memory returns the result of a load in the WB stage.

(A) (4 points) Fill in the pipeline diagram below for cycles 100-109, assuming that at cycle 100 the `lw a1, 0(a2)` instruction is fetched. Assume the loop runs for many iterations, and **`blt` is taken**. **Draw arrows indicating each use of bypassing.** Ignore cells shaded in gray.

	100	101	102	103	104	105	106	107	108	109
IF	`lw`	`add`	`addi`	`addi`	`addi`	`blt`	`slli`	`xori`	`lw`	`add`
DEC		`lw`	`add`	*`add`*	*`add`*	`addi`	`blt`	`slli`	**NOP**	`lw`
EXE			`lw`	**NOP**	**NOP**	`add`	`addi`	`blt`	**NOP**	**NOP**
MEM				`lw`	**NOP**	**NOP**	`add`	`addi`	`blt`	**NOP**
WB					`lw`	**NOP**	**NOP**	`add`	`addi`	`blt`

(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: 8

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

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

Ben studies his program and finds that most values in the array are 0. He modifies his 5-stage pipeline to perform **load value prediction** to improve performance. In this pipeline:

- If an instruction uses the result of a load, but the load has not yet produced the result, the pipeline predicts that the load will produce a 0. The bypass mux feeds value 0 to the consumer instruction instead of stalling it. This instruction continues execution speculatively, since it is using a predicted value.
- When a load reaches the WB stage, it checks whether the loaded value is 0. If it is not, and if a later instruction speculatively used a 0 value, then we guessed incorrectly. To preserve correct behavior, the pipeline flushes all instructions following the load. In the following cycle, the pipeline restarts execution of the instruction following the load at the IF stage.

(C) (10 points) Fill in the pipeline diagrams below for cycles 100-109 for this new pipeline. As before, assume that at cycle 100 `lw a1, 0(a2)` is fetched, and `blt` is **taken**. **Draw arrows indicating each use of bypassing *and load value prediction*.** Ignore cells shaded in gray.

First, assume that `lw a1, 0(a2)` **loads value 0 from memory.**

	100	101	102	103	104	105	106	107	108	109
IF	`lw`	`add`	`addi`	`blt`	`slli`	`xori`	`lw`	`add`	`addi`	`blt`
DEC		`lw`	`add`	`addi`	`blt`	`slli`	`NOP`	`lw`	`add`	`addi`
EXE			`lw`	`add`	`addi`	`blt`	`NOP`	`NOP`	`lw`	`add`
MEM				`lw`	`add`	`addi`	`blt`	`NOP`	`NOP`	`lw`
WB					`lw`	`add`	`addi`	`blt`	`NOP`	`NOP`

Second, assume that `lw a1, 0(a2)` **loads value 1 from memory.**

	100	101	102	103	104	105	106	107	108	109
IF	`lw`	`add`	`addi`	`blt`	`slli`	`add`	`addi`	`blt`	`slli`	`xori`
DEC		`lw`	`add`	`addi`	`blt`	`NOP`	`add`	`addi`	`blt`	`slli`
EXE			`lw`	`add`	`addi`	`NOP`	`NOP`	`add`	`addi`	`blt`
MEM				`lw`	`add`	`NOP`	`NOP`	`NOP`	`add`	`addi`
WB					`lw`	`NOP`	`NOP`	`NOP`	`NOP`	`add`

(D) (3 points) How many cycles does the execution of the loop take when load value prediction succeeds vs. when it fails? What percentage of loaded values must be zero for this technique to improve performance?

Cycles per loop iteration when load value prediction succeeds: _____6_____

Cycles per loop iteration when load value prediction fails: _____10_____

Percentage of 0s for load value prediction to help: _____>50%_____

***Read after the quiz**: It may seem that load value prediction is not very useful, because values are hard to predict, zeros are not that common, and we're not saving that many cycles per correct prediction. But in more complex pipelines, more cycles can be saved per prediction, so this idea sometimes makes sense! If you're interested, see "Value Locality and Load Value Prediction", by Lipasti, Wilkerson, and Shen, ASPLOS 1996.*

Problem 6. Pipelined Processor Performance (13 points)

You were hired as a systems engineer by a processor manufacturing company that specializes in the performance on map operations. Specifically, the company's processors work best for applying a function to each element of an array. Your boss thinks the processor with the best performance is the one with the minimum number of cycles and your first task on your first day is to design a processor with the best performance on the following benchmarking assembly code (the function here is an `addi` instruction):

```
                    // Assume that the starting address of the array
                    // already exists in register x12 and the end
                    // address of the array is in register x16.
loop:               lw   x10, 0(x12)
                    addi x10, x10, 2
                    sw   x10, 0(x12)
                    addi x12, x12, 4
                    ble  x12, x16, loop

some_other_code:    add x3, x4, x5
                    sub x5, x4, x3
```

Assume that the branch will always be taken. Branch decisions are resolved in the EXE stage.

(A) (4 points) As a proud 6.191 graduate, you know that you can build the beloved 5-stage pipelined processor from lecture with stages IF, DEC, EXE, MEM, and WB. Your first attempt is to build the baseline 5-stage pipelined processor **without bypassing**. Fill out the pipeline diagram below for the first 10 cycles. Fill in any stalled/annulled stages with NOPs. Calculate the number of cycles the processor takes to execute one iteration of the above loop. Also, specify the number of cycles wasted per loop iteration due to stalls and annulments. Additional pipeline diagrams are available at the end of the exam if needed.

Cycle	1	2	3	4	5	6	7	8	9	10
IF	lw	addi	sw	sw	sw	sw	addi	addi	addi	addi
DEC		lw	addi	addi	addi	addi	sw	sw	sw	sw
EXE			lw	NOP	NOP	NOP	addi	NOP	NOP	NOP
MEM				lw	NOP	NOP	NOP	addi	NOP	NOP
WB					lw	NOP	NOP	NOP	addi	NOP

Number of cycles per loop iteration: _____16_____

Number of cycles per loop iteration wasted due to stalls:_____9_____

Number of cycles per loop iteration wasted due to annulments:_____2_____

(B) (5 points) Your boss was not impressed with the baseline 5-stage pipelined processor. So you now decide to implement the same 5-stage pipelined processor but with **full bypassing** this time. Fill out the pipeline diagram below for the first 10 cycles. Fill in any stalled/annulled stages with NOPs and clearly show all your bypass arrows. Calculate the number of cycles the processor takes to execute one iteration of the loop. Also, specify the number of cycles wasted per loop iteration due to stalls and annulments. For your convenience, the assembly code is repeated below:

```
                    // Assume that the starting address of the array
                    // already exists in register x12 and the end
                    // address of the array is in register x16.
loop:               lw   x10, 0(x12)
                    addi x10, x10, 2
                    sw   x10, 0(x12)
                    addi x12, x12, 4
                    ble  x12, x16, loop

some_other_code:    add x3, x4, x5
                    sub x5, x4, x3
```

Again, assume that the branch will always be taken.

Cycle	1	2	3	4	5	6	7	8	9	10
IF	lw	addi	sw	sw	sw	addi	ble	add	sub	lw
DEC		lw	addi	addi	addi	sw	addi	ble	add	NOP
EXE			lw	NOP	NOP	addi	sw	addi	ble	NOP
MEM				lw	NOP	NOP	addi	sw	addi	ble
WB					lw	NOP	NOP	addi	sw	addi

Number of cycles per loop iteration: _____9_____

Number of cycles per loop iteration wasted due to stalls:_____2_____

Number of cycles per loop iteration wasted due to annulments:_____2_____

(C) (4 points) Your boss is impressed by your pipelined processor with full bypassing. However, he tells you that you can still get better performance with your processor from part (B) by modifying and/or reordering some of the instructions in the assembly code. If you think your boss is right, please provide the modified assembly code below and calculate the number of cycles per loop iteration. If you think your boss is wrong, argue why this is impossible below.

We have provided a pipeline diagram below that you can use to help answer the question, but *you do not need to fill the pipeline diagram*.

Is your boss correct? Circle one: **YES** **NO**

If yes, modified loop assembly code:

```
loop:
        lw   x10, 0(x12)
        addi x12, x12, 4
        addi x10, x10, 2
        sw   x10, -4(x12)
        ble  x12, x16, loop

some_other_code:    add x3, x4, x5
                    sub x5, x4, x3
```

If yes, number of cycles per loop iteration: _____8_____

If no, explanation: _____----------_____

Cycle	1	2	3	4	5	6	7	8	9	10
IF	lw	addi	addi	sw	sw	ble	add	sub	lw	addi
DEC		lw	addi	addi	addi	sw	ble	add	NOP	lw
EXE			lw	addi	NOP	addi	sw	ble	NOP	NOP
MEM				lw	addi	NOP	addi	sw	ble	NOP
WB					lw	addi	NOP	addi	sw	ble

END OF QUIZ 2!