Please write your name and Kerberos on your solutions.

Each subproblem is labelled with (Label: X), (e.g., Label: 1A_tpd). Please make sure you write down the label of each subproblem together with its solution and circle each answer. So, for example, if your computed propagation delay for problem 1A_tpd is 100ps, you would write:

1A_tpd: 100ps
Problem 1. Combinational and Sequential Logic Timing (13 points)

Consider the circuit shown below:

(A) (4 points) Find the propagation delay \( t_{PD} \) and the contamination delay \( t_{CD} \) of the circuit, using the \( t_{PD} \) and \( t_{CD} \) information for the gate components shown in the following table.

<table>
<thead>
<tr>
<th>Component</th>
<th>( t_{PD} )</th>
<th>( t_{CD} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>100ps</td>
<td>10ps</td>
</tr>
<tr>
<td>AND</td>
<td>300ps</td>
<td>30ps</td>
</tr>
<tr>
<td>OR</td>
<td>350ps</td>
<td>35ps</td>
</tr>
</tbody>
</table>

(Label: 1A_tpd) \( t_{PD} \): _______________ps

(Label: 1A_tcd) \( t_{CD} \): _______________ps
Now consider the sequential circuit below, which includes standard D flip flops, in addition to basic logic gates. All registers share a common clock. The table below shows the timing specification of each component.

Now consider the sequential circuit below, which includes standard D flip flops, in addition to basic logic gates. All registers share a common clock. The table below shows the timing specification of each component.

![Diagram of sequential circuit](image)

<table>
<thead>
<tr>
<th>Component</th>
<th>$t_{PD}$</th>
<th>$t_{CD}$</th>
<th>$t_{SETUP}$</th>
<th>$t_{HOLD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>100ps</td>
<td>10ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND</td>
<td>300ps</td>
<td>30ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR</td>
<td>350ps</td>
<td>35ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>REG</td>
<td>1000ps</td>
<td>40ps</td>
<td>150ps</td>
<td>100ps</td>
</tr>
</tbody>
</table>

(B) (4 points) What is the minimum clock period we can use for this circuit to function properly?

(B) (4 points) What is the minimum clock period we can use for this circuit to function properly?

(Label: 1B) *Minimum clock period for correct operation (ps): ________*

(C) (5 points) We are looking to decrease the clock cycle required by this circuit and discover a selection of faster OR gates we can use instead. However, because they are very expensive, we can only replace ONE of the OR gates in our circuit with one of the faster OR gates to the right. Each OR gate has a label next to it (X or Y). Which OR gate should be replaced?

(C) (5 points) We are looking to decrease the clock cycle required by this circuit and discover a selection of faster OR gates we can use instead. However, because they are very expensive, we can only replace ONE of the OR gates in our circuit with one of the faster OR gates to the right. Each OR gate has a label next to it (X or Y). Which OR gate should be replaced?

<table>
<thead>
<tr>
<th>OR gate type</th>
<th>$t_{PD}$</th>
<th>$t_{CD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td>300ps</td>
<td>30ps</td>
</tr>
<tr>
<td>Model 2</td>
<td>250ps</td>
<td>25ps</td>
</tr>
<tr>
<td>Model 3</td>
<td>150ps</td>
<td>15ps</td>
</tr>
<tr>
<td>Model 4</td>
<td>50ps</td>
<td>5ps</td>
</tr>
</tbody>
</table>

(Label: 1C_1) OR gate to be replaced (select X or Y as the gate to replace): X Y

(Label: 1C_1) OR gate to be replaced (select X or Y as the gate to replace): X Y

Which gate should it be replaced with to minimize the minimum clock period required while ensuring that the circuit still functions properly? What is that minimum clock period?

(Label: 1C_2) OR gate to replace it with (write down the model number): Model ____

(Label: 1C_2) OR gate to replace it with (write down the model number): Model ____

(Label: 1C_3) After replacement, *minimum clock period for correct operation (ps): ______*
Problem 2. Finite State Machines (14 points)

Suppose Alan wants to read a stream of 6s and 0s and find each separate instance where he has a 6, followed by one or more 0s, and then followed by another 6. Each instance should have no shared numbers with each other. In the state machine Alan designs, it outputs 1 for one cycle after having read in a sequence of the form 6, followed by one or more 0s, and then followed by another 6. Otherwise, it outputs a 0. Additionally, the FSM outputs a 1 the cycle immediately following the cycle where the last 6 input was processed.

Alan encodes each input digit into a digital value in the following way: Let input = 0 be when reading a 0, and input = 1 be when reading a 6.

He expects his FSM to undergo the following behavior:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Input</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Note that the input at cycle 5 will update the state at the beginning of the cycle 6, so the input at cycle 5 will have an effect at cycle 6. The – input in the table means that no valid input is given, and the state of FSM will not be updated. Also note that the output is only dependent on the current state, not the input.

(A) (6 points) Using the provided partial information, complete both the truth table and the state-transition diagram. In the diagram, fill in the missing state labels with its corresponding output (U) and label each transition with either 0 or 1 to indicate which input value causes the transition. The state shown in bold is the initial state.

(Label: 2A) Copy and complete the FSM diagram and the state transition table on your answer sheet
(B) (3 points) Given the following inputs, determine what the next state will be (A, B, C, or D) after the last input is processed:

(i) 60066

(Label: 2B_1) Next state: _________________

(ii) 660606

(Label: 2B_2) Next state: _________________

(iii) 60000000660000000006

(Label: 2B_3) Next state: _________________

(C) (5 points) Below is the truth table for a completely separate FSM from Parts (A & B). S₁ and S₀ are the two bits of state associated with this FSM, and as such the FSM will be implemented using two registers. Based on the truth table, complete the partially implemented FSM circuit. You may use NOT gates and 2-input AND, OR, and XOR gates in your implementation. For full credit, your implementation will consist of 5 gates or fewer.

<table>
<thead>
<tr>
<th>S₁&lt;sup&gt;t&lt;/sup&gt;</th>
<th>S₀&lt;sup&gt;t&lt;/sup&gt;</th>
<th>Input</th>
<th>S₁&lt;sup&gt;t+1&lt;/sup&gt;</th>
<th>S₀&lt;sup&gt;t+1&lt;/sup&gt;</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(LABEL: 2C) Draw your complete FSM circuit on your solution sheet.
Problem 3. Combinational Minispec (16 points)

(A) (8 points) Complete the truth table for the following Minispec function.

```
function Bit#(3) h(Bit#(1) a, Bit#(2) b);
  Bit#(3) ret = 3'b110;
  case ({a, b[1]})
    0: ret = {1'b0, zeroExtend(a) & b};
    1: ret = zeroExtend(a) + signExtend(b);
    3: ret = {a, ~b};
    default: ret = 3'b010;
  endcase
  return ret;
endfunction
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Label: 3A) Copy the truth table and fill in all the missing blanks.
(B) (8 points) The following incomplete Minispec function recursively implements an n-bit ripple-carry adder (rca) that returns the carry_out bit in addition to the sum (for a total of n + 1 bits on the output). The rca function uses the fullAdder function which has a hidden implementation of a full adder. The fullAdder function signature is also shown below.

```minispec
// fullAdder returns the sum bit in index 0 and the carry-out bit in index 1 of its output
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) carryIn);
// Implementation not given
endfunction

function Bit#(n + 1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) carryIn);
  if (n == 1)
    return __________ [Part B1] _______;
  else begin
    // Recursively sum everything except the MSBs of a and b
    Bit#(n) partialSum = rca#(__[Part B2]___)(_______[Part B2]_______);
    // Sum the MSBs
    Bit#(2) msbs = fullAdder(a[n-1], b[n-1], partialSum[n-1]);
    return {msbs, __________[Part B3]_______};
  end
endfunction
```

There are blanks in the code above labeled [Part B#]. Fill in the missing code, by copying each of the lines below and filling in the blanks corresponding to parts B1, B2, and B3 such that the completed code implements the rca function.

**Label 3B_1 B1:** return ________________________________;

**Label 3B_2 B2:** Bit#(n) partialSum = rca#(____)(____, _____, _____);

**Label 3B_3 B3:** return {msbs, ________________};
**Problem 4. Sequential Minispec (16 points)**

The incomplete Minispec module, `FindLongestBitRun`, below counts the length of the longest string of 1’s in a 32-bit word. The algorithm works by repeatedly performing a bitwise AND of the word with a version of itself that has been left-shifted by one. This repeats until the word is 0. The number of iterations required is the longest string of 1’s in the word. This works because each iteration converts the last 1 in any string of 1’s into a 0. The word will not equal zero until its longest string of 1’s has all been converted into 0’s.

The circuit should start a new computation when a Valid input is given and `bitString` is 0. The `bitString` register should be initialized to the input argument, and register `n` should hold the output. When the computation is finished, the `result` method should return a Valid result; while the computation is ongoing, `result` should return Invalid.

```plaintext
typedef Bit#(32) Word;

module FindLongestBitRun;
    Reg#(Bool) initialized(False);
    Reg#(Bit#(6)) n(0);
    Reg#(Word) bitString(0);

    input Maybe#(Word) in default = Invalid;

    method Maybe#(Bit#(6)) result;
        return (initialized && bitString == 0) ? [Part A1] : [Part A1];
    endmethod

    rule tick;
        if (isValid(in) && bitString == 0) begin
            n <= 0;
            bitString <= [Part A2];
            initialized <= True;
        end else if (initialized && (bitString != 0)) begin
            n <= n + 1;
            bitString <= [Part A3];
        end
    endrule
endmodule
```

(A) (8 points) There are blanks in the code above labeled [Part A#]. #. Fill in the missing code, by copying each of the lines below and filling in the blanks corresponding to parts A1, A2, and A3.

You may use any Minispec operators, built-in functions, and literals. You will not need additional registers to complete this problem. Do not add additional rules, methods, or functions.

(Label: 4A_1) A1: return (initialized & bitString == 0) ? _____ : _____;

(Label: 4A_2) A2: bitString <= ____________________________;

(Label: 4A_3) A3: bitString <= ____________________________;
(B) (8 points) At cycle 0, the input is set to $\text{Valid}(32'b0111)$. Copy and fill in the table below to indicate the values at the output of the $\text{result()}$ method, the value in register $n$, and the value in the $\text{bitString}$ register. Write “Invalid” if a value is invalid, “?” if a value is unknown, and just a number to indicate a valid value (i.e. you do not need to write “Valid(5)”; just write “5”). “0b” indicates that the number after it is a binary value.

(Label: 4B) Copy and fill in the table below

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0b0111</td>
<td>Invalid</td>
<td>0b1111</td>
<td>Invalid</td>
<td>0b0001</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>$\text{result()}$ output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value in register $n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value in $\text{bitString}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 5. Arithmetic Pipelines (14 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.

(A) (6 points) Show the **maximum-throughput 2-stage pipeline** using a minimal number of registers. What are the latency and throughput of the resulting circuit?

![Diagram of the circuit](image)

(copy diagram and add pipeline contours and registers for a maximum throughput 2-stage pipeline. (Note that the arrow between the middle 2 and the top left 4 points UP.))

Then provide the latency and throughput for this circuit.

(Label: 5A_latency) Latency (ns): _________

(Label: 5A_throughput) Throughput (ns\(^{-1}\))): _________
(B) (8 points) Show the maximum-throughput pipeline using a minimal number of registers. What are the latency and throughput of the resulting circuit?

(Label: 5B_1) Copy the diagram above and add pipeline contours and registers for a maximum throughput pipeline using a minimal number of registers. (Note that the arrow between the middle 2 and the top left 4 points UP.)

Then provide the latency and throughput for this circuit.

(Label: 5B_latency) Latency (ns): __________

(Label: 5B_throughput) Throughput (ns⁻¹): __________
Problem 6. RISC-V Processor (12 points)

**Note:** This question uses and extends a processor diagram that differs slightly from the one we have seen in lecture and recitation, specifically in the Decoder signals. This was the diagram used in Spring 2020, which we have simplified over time. This term, if the quiz had a question of this type, it would use the diagram from this term.

Giuseppe is writing a large RISC-V assembly program, and is tired of getting confused by how to properly maintain the stack pointer sp. He’s heard that other ISAs have push and pop instructions, which handle both allocating/freeing the space for a word on the stack, and storing it from/loading it to a register. Giuseppe decides that he’d like to implement them in the RISC-V ISA. Their syntaxes are as follows:

```
push rs2
pop rd
```

And the following is a Python-like description of how each works:

**push:**
```
reg[sp] = reg[sp] – 4
Mem[reg[sp]] = reg[rs2]
```

**pop:**
```
reg[rd] = Mem[reg[sp]]
reg[sp] = reg[sp] + 4
```

(A) (4 points) Giuseppe hopes to be able to simply add some new signals as inputs to the selection muxes in the existing RISC-V processor diagram. For each instruction being added, can this be done with the existing processor components as described in lecture and shown on the following page? If not, describe the limitation that prevents it.

(Label: 6A_push) Can push be implemented with existing components?

(Label: 6A_pop) Can pop be implemented with existing components?

Giuseppe’s friend Jeffrey tells him that the push and pop instructions can cause issues when pipelining his processor later on. Giuseppe, knowing nothing about processor pipelining, decides to take his word for it and find a different solution to his problem.

His goal now is to remove a large amount of the uses of stack, so that he doesn’t have as many opportunities to become confused. The main limitation that causes him to use the stack so much is that he’s running out of registers to hold temporary values. In order to overcome this, he wants to combine multiple steps into a single instruction. In particular, he wants to combine the ANDing of two registers and XORing the result with an immediate into a single instruction. All immediates he uses are 10 bits or less, so he comes up with the following instruction syntax, definition, and encoding:
andxori $rd$, $rs1$, $rs2$, $imm$

andxori:
\[
\text{reg}[rd] = ((\text{reg}[rs1] \& \text{reg}[rs2]) \^ \text{signExtend}(imm))
\]

<table>
<thead>
<tr>
<th>31...25</th>
<th>24...20</th>
<th>19...15</th>
<th>14...12</th>
<th>11...7</th>
<th>6...0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[9:3]</td>
<td>$rs2$</td>
<td>$rs1$</td>
<td>imm[2:0]</td>
<td>rd</td>
<td>0110100</td>
</tr>
</tbody>
</table>

(B) (2 points) Encode the following instructions as 32-bit binary words:

andxori $x7$, $x1$, $x16$, $0x1BC$

**(Label: 6B) Encoding in hexadecimal 0x: _________**

In the diagram below, Giuseppe has decoded and sign extended the immediate used for the andxori instruction, labelling this signal immA. He also added an additional dedicated XOR module used in the instruction’s computation. The output of this new XOR module is labelled axRes.
(C) (3 points) For each of the following signals, does the mux being controlled by that signal need 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.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Needs new input?</th>
<th>Input/Control signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSEL</td>
<td>YES</td>
<td>New input/Existing control signal: __________</td>
</tr>
<tr>
<td>WDSEL</td>
<td>YES</td>
<td>New input/Existing control signal: __________</td>
</tr>
<tr>
<td>WERFSEL</td>
<td>YES</td>
<td>New input/Existing control signal: __________</td>
</tr>
</tbody>
</table>

(D) (3 points) Additionally, decide for each of the following control signals what their values should be when executing the andxori 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
- MWR: Read, Write

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AluFunc</td>
<td>______________</td>
</tr>
<tr>
<td>BrFunc</td>
<td>______________</td>
</tr>
<tr>
<td>MWR</td>
<td>______________</td>
</tr>
</tbody>
</table>
Problem 7. Caches (15 points)

Anne and Ben just learned about caches in 6.004 and decided to design their own.

(A) (2 points) They would like to design a cache with an AMAT (average memory access time) of 3 cycles. Accessing the cache should take 1 cycle, and on a miss, it should take an additional 16 cycles to retrieve the data from main memory, update the cache, and return the requested word to the processor. What should their hit ratio be in order to achieve the target AMAT?

(Label: 7A) Hit ratio: ________

Ben suggests implementing a 2-way set-associative cache with a block size of 4 (i.e. 4 words per line). The number of sets in the cache is 8. Assume that addresses and data words are 32 bits wide.

(B) (2 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__]

(Label: 7B_block) Address bits used for block offset: A[____ : _____]

(Label: 7B_index) Address bits used for cache index: A[____ : _____]

(Label: 7B_tag) Address bits used for tag field: A[____ : _____]

(C) (2 points) Anne agrees with the appropriateness of a 2-way set-associative implementation, but she suspects that a larger block size might result in a higher hit rate. Suppose the block size of the cache is doubled to 8. If the total number of data words in the cache remains unchanged, how would the number of cache lines change?

(Label: 7C) Change in # of cache lines (select one of the choices below):

UNCHANGED ... +1 ... -1 ... 2x ... 0.5x ... CAN’T TELL
Ultimately, they choose to implement Ben’s 2-way, 4-block cache. Below is a snapshot of the cache during the execution of some unknown code. The column labeled Word \( x \) corresponds to the \( x \)th word of the block. The V bit specifies whether or not the line is valid, and the D bit specifies whether or not the line is dirty.

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0x32</td>
<td>0x0A</td>
<td>0x1A</td>
<td>0x2A</td>
<td>0x3A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x32</td>
<td>0x4B</td>
<td>0x5B</td>
<td>0x6B</td>
<td>0x7B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x32</td>
<td>0x3C</td>
<td>0x2C</td>
<td>0x1C</td>
<td>0x0C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x32</td>
<td>0x7D</td>
<td>0x6D</td>
<td>0x5D</td>
<td>0x4D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x50</td>
<td>0x33</td>
<td>0x23</td>
<td>0x13</td>
<td>0x03</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x50</td>
<td>0x44</td>
<td>0x34</td>
<td>0x24</td>
<td>0x14</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0x43</td>
<td>0x55</td>
<td>0x65</td>
<td>0x75</td>
<td>0x85</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x66</td>
<td>0x66</td>
<td>0x76</td>
<td>0x86</td>
<td>0x96</td>
</tr>
</tbody>
</table>

(D) (3 points) Would a load request to address 0x193C result in a hit or a miss? If it results in a hit, specify what value is returned; if it is a miss, write N/A.

(Label: 7D_1) Hit / Miss : ________

(Label: 7D_2) Returned value if hit or N/A if miss: ________
Anne and Ben want to analyze the performance of this cache on the following assembly program, which calculates the first 256 terms in the Fibonacci sequence and stores them in an array A. The base address of array A is $0x3000$.

// Assume the following registers are initialized:
// x1 = 0 (initial loop index)
// x2 = 256 - 2 = 254 (number of Fibonacci elements to calculate)
// x3 = 0x3000 (base address of array A)

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

fibonacci:
  li x4, 1         // x4 = 1 (second element in sequence)
  sw x0, 0(x3)     // A[0] = 0
  sw x4, 4(x3)     // A[1] = 1

loop:
  slli x4, x1, 2   // x4 = byte offset of the ith element
  add x5, x4, x3   // x5 = address of A[i]
  lw x6, 0(x5)     // x6 = A[i]
  lw x7, 4(x5)     // x7 = A[i+1]
  add x6, x6, x7   // x6 = A[i] + A[i+1]
  sw x6, 8(x5)     // A[i+2] = x6
  addi x1, x1, 1   // increment i
  blt x1, x2, loop // continue looping

Answer the following questions about the behavior of the cache during execution of the above code. Assume that the cache uses a least recently used (LRU) replacement policy, that the cache is initially empty, and that all cache lines in Way 0 are currently the least-recently used.

(E) (2 points) How many instruction fetches and data accesses occur per iteration of the loop?

(Label: 7E_instr) Number of instruction fetches: ________

(Label: 7E_data) Number of data accesses: ________

(F) (4 points) After the program has been running for many loop iterations, what is the steady-state hit ratio for instruction fetches and data accesses?  

Hint: Note that in steady state each array element is accessed in multiple loop iterations.

(Label: 7F_instr) Steady-state hit ratio for instruction fetches: ________

(Label: 7F_data) Steady-state hit ratio for data accesses: ________

END OF QUIZ 2!