Problem 1. Sequential Logic Timing (13 points)

Consider the circuit shown below, which calculates the parity of two bits, with an optional data override.

(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>XOR</td>
<td>400ps</td>
<td>40ps</td>
</tr>
<tr>
<td>NOT</td>
<td>100ps</td>
<td>10ps</td>
</tr>
<tr>
<td>AND</td>
<td>200ps</td>
<td>20ps</td>
</tr>
<tr>
<td>OR</td>
<td>250ps</td>
<td>25ps</td>
</tr>
</tbody>
</table>

$t_{PD}$: ___________ ps

$t_{CD}$: ___________ ps
Now consider the sequential circuit below, which uses two different types of registers, REG A and REG B. Both are standard D flip flops; they differ only in their timing parameters. All registers share a common clock. The dashed region indicates a part of the circuit, which currently contains only a wire. The table below shows the timing specification of each component.

![Circuit Diagram]

<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>XOR</td>
<td>400ps</td>
<td>50ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NOT</td>
<td>100ps</td>
<td>10ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND</td>
<td>200ps</td>
<td>20ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR</td>
<td>250ps</td>
<td>25ps</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>REG A</td>
<td>600ps</td>
<td>30ps</td>
<td>120ps</td>
<td>40ps</td>
</tr>
<tr>
<td>REG B</td>
<td>640ps</td>
<td>50ps</td>
<td>220ps</td>
<td>0ps</td>
</tr>
</tbody>
</table>

(B) (5 points) What is the minimum clock period we can use for this circuit to function properly?

**Minimum clock period for correct operation (ps):**

(C) (4 points) Now suppose that we replace Reg B with a register with the same timing parameters except that its $t_{HOLD}$ is now **100ps**. We are considering adding NOT gates to the region in the circuit indicated by the dashed box. What is the smallest number of not gates (if any) that should be added in sequence (no parallel connections) within the dashed region in order for the circuit to function correctly without changing its functionality?

**Minimum number of NOT gates needed, or NONE if not needed:**

---

6.004 Fall 2019 - 2 of 16 - Quiz #2
Problem 2. Finite State Machines (14 Points)

The truth table and state-transition diagram below correspond to the same FSM. Both are only partially complete. This FSM is designed to be the control system for a gondola that transports people between four different stations at the Fine Ice Skate & Skis Resort. The FSM receives a single binary digit as input each cycle.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>B</td>
</tr>
</tbody>
</table>

(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 and label each transition with either 0 or 1 to indicate which input value causes the transition.

Add missing information to truth table and state-transition diagram above

(B) (2 points) The resort staff want to implement an override system that will force the gondola to return to station A at the end of the day or if the gondola requires maintenance. However, the command console is broken and can only remotely issue inputs to the gondola in groups of 3. If you know nothing about the current state of the FSM, is there a sequence of EXACTLY 3 input bits that will always result in the FSM ending up in state A, no matter what state it was in before? If there is such a sequence, please provide it below, and if not then write NONE.

3-bit override sequence or NONE: __________________
(C) (6 points) You have been charged with implementing the circuit for another FSM, which controls the lighting systems on the ski slopes. **This is a completely separate FSM from part (A).** Below is the truth table for this new FSM. \( S_1 \) and \( S_0 \) are the two bits of state associated with the FSM, and as such the FSM will be implemented using two registers. Based on the truth table, complete the partially implemented FSM circuit to control how \( S_1 \) and \( S_0 \) are updated below. **You may use AND, OR, NOT, and XOR gates** in your implementation. For full credit, your implementation should use only **3 gates**.

<table>
<thead>
<tr>
<th>( S_1^t )</th>
<th>( S_0^t )</th>
<th>Input</th>
<th>( S_1^{t+1} )</th>
<th>( S_0^{t+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram of FSM circuit](image-url)
Problem 3. Sequential Circuits in Minispec (18 points)

You join a startup building hardware to mine Dogecoins. In this cryptocurrency, mining coins requires repeatedly evaluating a function with two arguments, \( sc(x, y) \). \( x \) is given to you, and mining requires trying different values of \( y \) until you find a \( y \) for which \( sc(x, y) \) is below a threshold value. Finding such a \( y \) value yields several Dogecoins as a reward, which you can then exchange for cold hard cash.

Because the \( sc \) function is expensive, it is implemented as a multi-cycle sequential module, called \( SC \). \( SC \) is given to you. Its implementation is irrelevant, and its interface, shown below, is the usual interface for multi-cycle modules: \( SC \) has a single input, \( in \), and a single method, \( getResult() \). To start a new computation, the module user sets \( in \) to a \( Valid \) \( Args \) struct containing arguments \( x \) and \( y \). Some cycles later, \( SC \) produces the result as a \( Valid \) output of its \( getResult() \) method. While \( SC \) is processing an input, the \( getResult() \) method returns \( Invalid \) and \( in \) should stay \( Invalid \).

```plaintext
module SC;
    input Maybe#(Args) in default = Invalid;
    method Maybe#(Bit#(32)) getResult();
        // unknown implementation
    endmethod
    // unknown rules
endmodule
```

You are asked to design the \( ArgFeeder \) module, which accepts an input \( x \), and feeds a sequence of inputs \((x, 0), (x, 1), (x, 2), ..., (x, y-1), (x, y)\) to the \( SC \) module. \( ArgFeeder \) keeps feeding values to \( SC \) until \( SC \)'s result is less than \( threshold \) (a parameter to your module). At that point, \( ArgFeeder \) should return the \( y \) such that \((x, y)\) meets this condition through its \( getResult() \) method. The diagram below sketches the implementation of \( ArgFeeder \). Like \( SC \), \( ArgFeeder \) follows the usual interface for a multi-cycle module.

Implement the \( ArgFeeder \) module by completing the implementation of the \( getResult() \) method and the \( tick \) rule. The rule considers three cases:

(i) a new input is provided to \( ArgFeeder \),
(ii) \( SC \) returns a \( Valid \) result, and it is less than the threshold value, and
(iii) \( SC \) returns a \( Valid \) result, but it is not less than the threshold value.

You may use any Minispec operator, including arithmetic (+, -, *, /). You will not need additional registers to complete this problem. Do not add additional rules, methods, or functions.
module ArgFeeder#(Integer threshold);
SC sc;

Reg#(Maybe#(Bit#(32))) out(Invalid);
RegU#(Bit#(32)) x;
RegU#(Bit#(32)) y;

input Maybe#(Bit#(32)) in_x default = Invalid;

method Maybe#(Bit#(32)) getResult();
    // implement the getResult() method
    return ;
endmethod

rule tick;
    if (isValid(in_x)) begin
        // case (i): received a new input; start a new sequence of (x, y) pairs
        sc.in = Valid(Args{x: , y: });
        out <= ;
        x <= ;
        y <= ;
    end else if (isValid(sc.getResult())) begin
        if (fromMaybe(? , sc.getResult()) < threshold) begin
            // case (ii): result satisfies threshold
            out <= ;
        end else begin
            // case (iii): result does not yet satisfy threshold
            //   send next (x, y) pair to SC
            sc.in = Valid(Args{x: , y: });
            y <= ;
        end
    end else begin
        // case (iii): result does not yet satisfy threshold
        //   send next (x, y) pair to SC
        sc.in = Valid(Args{x: , y: });
        y <= ;
    end
endrule
endmodule
Problem 4. 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. Additional pipeline diagrams are provided at the end of the quiz.

(A) (8 points) Show the maximum-throughput 2-stage pipeline using a minimal number of registers. What are the latency and throughput of the resulting circuit?

Latency (ns): __________  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?

Latency (ns): __________  Throughput (ns^{-1}): __________
Problem 5. Processor Implementation (20 points)

We want to add the following instruction to the RISC-V ISA:

```
cmvz rd, rs1, rs2
```

This new cmvz instruction is a conditional move: it checks the contents of source register rs1, and if it’s 0, copies the contents of source register rs2 into rd. In other words:

```
if (reg[rs1] == 0) begin
    reg[rd] <= reg[rs2];
end
```

We have chosen the following encoding for CMVZ:

<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>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0001011</td>
</tr>
</tbody>
</table>

(A) (2 points) Encode the following instruction as a 32-bit binary word:

```
cmvz x1, x2, x3
```

**Encoding in hexadecimal 0x: __________________________**

(B) (4 points) One of your teammates proposes modifying your processor from lecture and Lab 6 to implement this new instruction in a single cycle as follows:

1. From your register file, read the contents of all three registers rd, rs1, and rs2.
2. Add a comparator that compares the contents of rs1 against 0.
3. Add a mux that uses the result of the comparison above to select between the contents of rd and the contents of rs2, choosing the contents of rs2 if they are equal, and the contents of rd if they are not.
4. Write the result of the mux back into register rd of the register file.

Briefly explain why this implementation strategy does not work with the components in the RISC-V processor from lecture and Lab 6.
(C) (8 points) Instead of the above strategy, we want to implement cmvz without reading register rd, by doing the comparison to decide whether to write into rd in the execute function.

Suppose you are given implementations of the decode and execute functions that are fully functional prior to the addition of this new instruction. We have added a new IType label, CMVZ, for this instruction. **Fill in the blanks below** in the decode and execute functions in order to implement this new instruction. Assume that the rest of the hardware implementation is the same as what you have seen in lecture and lab.

```cpp
typedef enum { OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE, AUIPC, CMVZ, Unsupported } IType;
Bit#(7) opCMVZ = 7'b0001011;
Bit#(3) fnCMVZ = 3'b100;
// ...
function DecodedInst decode(Bit#(32) inst);
  Bit#(7) opcode = inst[6:0];
  Bit#(3) funct3 = inst[14:12];
  Bit#(7) funct7 = inst[31:25];
  RIndx dst = inst[11:7];
  RIndx src1 = inst[19:15];
  RIndx src2 = inst[24:20];
  Maybe#(RIndx) validDst = Valid(dst);
  Maybe#(RIndx) dDst = Invalid;  // default value
  RIndx dSrc = 5'b0;  // default value
  Word immD = signExtend(1'b0);  // default value
  // ...
  DecodedInst dInst = unpack(0);
  dInst.iType = Unsupported;
  // ...
  case (opcode)
    // ...
    opStore: begin
      if (funct3 == fnSW) begin
        dInst = DecodedInst {iType: STORE, dst: dDst, src1: src1, src2: src2, imm: immS, brFunc: Dbr, aluFunc: ?};
      end
    end
    opCMVZ: begin
      if (_____________________________________________)
        dInst = DecodedInst {iType: ___, dst: ___,
          src1: ___, src2: ___, imm: ___,
          brFunc: ___, aluFunc: ___};
    end
  endcase
  return dInst;
endfunction
```

```cpp```
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2,  
        Word pc);

    Word imm = dInst.imm;
    BrFunc brFunc = dInst.brFunc;
    AluFunc aluFunc = dInst.aluFunc;
    Maybe#(RIndx) dst = dInst.dst;

    Word data = case (dInst.iType)
        AUIPC: pc + imm;
        // ...
        CMVZ: 
        default: 0;
    endcase;

    Word nextPc = // ...
    Word addr = // ...

    if (dInst.iType == CMVZ &&                               ) begin
        // don’t perform the move
        dst =                               ;
    end

    return ExecInst { iType: dInst.iType, dst: dst, data: data,  
                        addr: addr, nextPc: nextPc };

endfunction
Now we would like to update the hardware diagram of the single-cycle processor to support this new instruction. **Unlike in part (C), we would like to avoid changing the ALU or adding new comparisons** to minimize the amount of hardware added. Instead, we would like to reuse the existing **branch ALU comparison logic** to check if $rs1$ is $0$. In the simplified diagram below, we have added extra inputs to the BSEL and WDSEL muxes to support reusing the branch ALU in this way, and we have changed the WERF output from the decoder into a WERFSEL signal.

Assume that when the decoder decodes CMVZ, it outputs $BrFunc = Eq$, $BSEL = 3$, $WDSEL = 3$, and $WERFSEL = 2$. It also outputs $PCSEL = 0$ so that the next value of $PC$ will be $PC + 4$.

(D) (3 points) For each of the missing mux inputs $X$, $Y$, and $Z$, write down either a constant or the name of a different wire or port in the processor hardware diagram that should be connected to each input, so that the processor executes CMVZ correctly.

$X$: ____________

$Y$: ____________

$Z$: ____________

**Summary of some of the relevant processor components in the above diagram:**

- One register in the register file can be written to each cycle by providing the register index in $WA$ and the data to write in $WD$. The data is only written if the write enable signal, $WE$, is $1$.
- The ALU receives two inputs, $A$ and $B$. It computes an arithmetic operation on them specified by $AluFunc$, outputting the 32-bit result to $aluResult$, and a comparison specified by $BrFunc$, outputting the 1-bit result to branch.
We would like to check that our processor can still decode and execute the old RISC-V instructions correctly. Suppose the processor is decoding an OPIMM instruction, such as ADDI. For each of the select signals we changed (shown in **bold** in the diagram), write down the correct signals that the decoder should produce for such an instruction. Another identical copy of the diagram is provided below for ease of reference.

Decoded value of BSEL: _____
Decoded value of WDSEL: _____
Decoded value of WERFSEL: _____

Summary of some of the relevant processor components in the above diagram:
- One register in the register file can be written to each cycle by providing the register index in WA and the data to write in WD. The data is only written if the write enable signal, WE, is 1.
- The ALU receives two inputs, A and B. It computes an arithmetic operation on them specified by AluFunc, outputting the 32-bit result to aluResult, and a comparison specified by BrFunc, outputting the 1-bit result to branch.
Problem 6. Caches (19 points)

(A) (2 points) We would like to design a cache with an AMAT (average memory access time) of 1.5 cycles. Accessing our cache takes 1 cycle, and on a miss, it takes an additional 10 cycles to retrieve the data from main memory and update the cache. What does our hit ratio need to be in order to achieve the target AMAT?

Hit ratio = __________

We choose to implement 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 4. Assume that addresses and data words are 32 bits.

(B) (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[ _____ : _____ ]
Address bits used for cache index: A[ _____ : _____ ]
Address bits used for tag field: A[ _____ : _____ ]

(C) (2 points) Assuming the cache uses a writeback policy, what is the total number of bits per cache line? Please show your work for partial credit.

______________ bits
We want to analyze the performance of this cache on the following assembly program, which iterates through a 1000-word array A and sets each element to \( A[i] = -A[i] \). The base address of array A is \( 0x3000 \).

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

// Assume the following registers are initialized:
// x1=0 (loop index)
// x2=1000 (number of array elements)
// x3=0x3000 (base address of array A)

loop:
    slli x5, x1, 2 // x5 = byte offset of the ith element
    add x6, x5, x3 // x6 = address of A[i]
    lw x7, 0(x6)   // x7 = A[i]
    sub x7, x0, x7 // x7 = -A[i]
    sw x7, 0(x6)   // store A[i] = -A[i]
    addi x1, x1, 1 // increment i
    blt x1, x2, loop // continue looping

(D) (6 points) Below is the cache state the first time the program is about to enter the loop at loop. Assume that the cache uses a least-recently used (LRU) replacement policy, and that all cache lines in Way 1 are currently the least-recently used. Mark up the cache below to indicate the state of the cache immediately after one loop iteration (after executing the blt instruction for the first time). You do not need to specify the value of data words, but do specify the values of D (dirty bit), V (valid bit), and Tag.

<table>
<thead>
<tr>
<th>Way 0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Way 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>V</td>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Way 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Way 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>V</td>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(E) (1 point) How many instruction fetches and data accesses occur per iteration of the loop?

Number of instruction fetches: _________

Number of data accesses: _________

(F) (5 points) After the program has been running for many loop iterations, what is the steady-state hit ratio for instruction fetches and data accesses?

Steady-state hit ratio for instruction fetches: _________

Steady-state hit ratio for data accesses: _________

END OF QUIZ 2!
Additional pipeline diagrams for Problem 4