Problem 1. Sequential Logic Timing (16 points)

Consider the **toggle flip-flop (TFlop)**, whose symbol is shown on the right. The TFlop is a clocked device, as shown by the clock input indicated by the triangle on its lower-left edge. It stores one bit of state, given at the Q output. The input T (for **toggle**) controls whether the state will change at the next rising clock edge. If T=1 on the rising edge of the clock, the TFlop flips its state (the Q output) from 0 to 1 or vice versa. If T=0 on the rising edge of the clock, the state of the TFlop remains unchanged.

(A) (12 points) A TFlop may be implemented using a D flip-flop like the ones developed in lecture together with an XOR2 gate, as shown below.

Like for D flip-flops, the TFlop’s input T should be valid and stable for at least some setup time ($t_{\text{setup}}$) before the rising edge of the clock and for at least some hold time ($t_{\text{hold}}$) after the rising edge of the clock. Also like for D flip-flops, the TFlop’s propagation delay ($t_{\text{PD}}$) and contamination delay ($t_{\text{CD}}$) are measured from the rising edge of the clock to the output.

Using the timing specifications for the components shown in the table below, derive the timing specifications of this TFlop implementation.

<table>
<thead>
<tr>
<th>Component</th>
<th>$t_{\text{CD}}$</th>
<th>$t_{\text{PD}}$</th>
<th>$t_{\text{setup}}$</th>
<th>$t_{\text{hold}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR2</td>
<td>40ps</td>
<td>400ps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DREG</td>
<td>100ps</td>
<td>300ps</td>
<td>80ps</td>
<td>40ps</td>
</tr>
</tbody>
</table>

$t_{\text{CD}}$: ______100______ps

$t_{\text{PD}}$: ______300______ps

$t_{\text{setup}}$: ______480______ps

$t_{\text{hold}}$: ______0______ps
(B) (2 points) Suppose we connect the T input of a single TFlop to 1 (i.e., \( V_{DD} \)) and try to clock it at its maximum rate. What is the minimum clock period we can use and expect the TFlop to perform properly?

\[ \text{Minimum clock period for correct operation: } \underline{780} \text{ ps} \]

We next consider the use of four TFlops to make a 4-bit ripple-carry counter as shown to the right. Assume that the TFlops share a common clock input (not shown) with an appropriate period, and that all TFlops have an initial Q=0 state.

(C) (2 points) If the AND2 gates have \( t_{PD} = 200 \text{ ps} \) and \( t_{CD} = 40 \text{ ps} \), what is the minimum clock period we can use for the 4-bit counter?

\[ \text{Minimum clock period for correct operation: } \underline{1180} \text{ ps} \]
Problem 2. Finite State Machines (10 points)

Below is a state transition diagram for a 4-state FSM with a single binary input B. The FSM has a single output – a light which is “on” when the FSM is in states “E” or “S”. The starting state “W” is marked by the heavy circle.

A) (8 points) Please fill in as many entries as possible in the following truth table for the FSM. The light output should be 1 when the light is “on” and 0 when it’s “off.”

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>B</th>
<th>S1’</th>
<th>S0’</th>
<th>light</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

B) (2 points) Please label the states with their corresponding S1S0 values.

E = 00; S = 01; W = 11; N = 10;
Problem 3. Sequential Circuits (19 points)

The following code implements a simple sequential circuit as a module that computes a function over a series of steps.

```verilog
typedef enum {Idle, Busy, Done} State;
typedef Bit#(32) Word;

module FSMAdder;

    Reg#(Word) sum(0);
    Reg#(Bit#(3)) index(0);
    RegU#(Vector#(4, Word)) vec;
    Reg#(State) state(Initialize);

    input Maybe#(Vector#(4, Word)) in default = Invalid;

    rule tick;
        if (isValid(in) && state != Busy) begin
            sum <= 0;
            index <= 0;
            vec <= fromMaybe(? , in);
            state <= Busy;
        end else if (index < 4 && state == Busy) begin
            sum <= sum + vec[index];
            index <= index + 1;
        end else if (index == 4) begin
            state <= Done;
        end
    endrule

    method Bool isDone;
        return (state == Done);
    endmethod

    method Maybe#(Word) getResult;
        return (state == Done)? Valid(sum) : Invalid;
    endmethod

endmodule
```

---

6.004 Fall 2020 - 4 of 13 - Practice Quiz #2A
A) (6 points) At cycle 0, the FSMAdder module is in the Idle state. At cycle 0, the input to the FSMAdder module is set to a Valid vector [5, 5, 5, 5]. In successive cycles (i.e., from cycle 1 onwards), FSMAdder's input is set to Invalid.

Fill in the table below indicating the values at the output of the isDone() and getResult() methods during cycles 0-7. For Maybe types, express the value as invalid or Valid(someValue).

<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>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDone() output</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>getResult() output</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Valid(20)</td>
<td>Valid(20)</td>
</tr>
</tbody>
</table>

B) Rewrite the getResult() method so that it produces the output one cycle earlier.

```plaintext
method Maybe#(Word) getResult;
    return (index == 4)? Valid(sum) : Invalid;
endmethod
```

C) Below is a block diagram of the circuit produced by this code for the index and state registers. Given this diagram’s implementation of this code, provide a Boolean expression that specifies when state_sel equals 0?

state_sel equals 0 when:

index < 4 && state == Busy
Now suppose that we modify FSMAdder’s rule as shown below. Changes are highlighted in **green**:

```
rule tick;
    if (isValid(in) && state != Busy) begin
        sum <= 0;
        index <= 0;
        vec <= fromMaybe(?; in);
        state <= Busy;
        end
    else if (index < 4 && state == Busy) begin
        sum <= sum + vec[index];
        index <= index + 1;
        end
    else if (index == 4) begin
        state <= Done;
        index <= index + 1;
        end
    else begin
        state <= Idle;
        end
endrule
```

D) (5 points) Using the updated code, assume again that at cycle 0, the FSMAdder module is in the **Idle** state. At cycle 0, the input to the FSMAdder module is set to a **Valid** vector `[5, 5, 5, 5]`. In successive cycles (i.e., from cycle 1 onwards), FSMAdder’s input is set to **Invalid**.

Fill in the table below indicating the values at the output of the `isDone()` and `getResult()` methods during cycles 0-11. For Maybe types, express the value as **Invalid** or **Valid(someValue)**.

<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>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDone() output</td>
<td><strong>False</strong></td>
<td><strong>False</strong></td>
<td><strong>False</strong></td>
<td><strong>False</strong></td>
<td><strong>False</strong></td>
<td><strong>False</strong></td>
<td><strong>True</strong></td>
<td><strong>False</strong></td>
</tr>
<tr>
<td>getResult() output</td>
<td><strong>Invalid</strong></td>
<td><strong>Invalid</strong></td>
<td><strong>Invalid</strong></td>
<td><strong>Invalid</strong></td>
<td><strong>Invalid</strong></td>
<td><strong>Invalid</strong></td>
<td><strong>Valid(20)</strong></td>
<td><strong>Invalid</strong></td>
</tr>
</tbody>
</table>
Problem 4. Pipelining Combinational Circuits (20 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\text{PD}=0, t\text{SETUP}=0). Remember that our convention is to place a pipeline register on each output.

If you need them, additional copies of these circuits are available at the end of the quiz.

(A) (2 points) Show the maximum-throughput 1-stage pipeline.

![Diagram](image)

Latency (ns): __9_____

Throughput (ns\(^{-1}\)): __1/9_____

(B) (4 points) Show a maximum-throughput 2-stage pipeline.

![Diagram](image)

Latency (ns): __12_____

Throughput (ns\(^{-1}\)): __1/6_____

6.004 Fall 2020 - 7 of 13 - Practice Quiz #2A
(C) (6 points) Show a maximum-throughput pipeline that uses the smallest possible number of pipeline stages.

![Diagram](image1)

Latency (ns): ___12_____  
Throughput (ns⁻¹): _1/4_____

(D) (8 points) You reimplement the 4ns combinational component in the previous circuit using two faster components connected in series, shown in grey below. You can choose the propagation delays of these two components, as long as their delays add to 4ns (e.g., they could be 3ns and 1ns, 2ns and 2ns, etc.)

Choose the propagation delays of both components in a way that lets you pipeline the circuit for maximum throughput while minimizing the number of pipeline stages. Then, find the maximum-throughput pipeline. For full credit, your solution should use the minimum possible number of pipeline stages.

![Diagram](image2)

Latency (ns): ___9______  
Throughput (ns⁻¹): _1/3______
Problem 5. Processor Implementation (12 points)

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

\[ \text{lwr rd, rs1, rs2} \]

The behavior of this new lwr instruction is that it adds the contents of source registers rs1 and rs2 and uses the result as the address for your load. In other words:

\[
\text{reg[rd]} \leftarrow \text{mem[reg[rs1] + reg[rs2]]} \\
\text{pc} \leftarrow \text{pc} + 4
\]

Suppose you were given a decoder that was fully functioning prior to the addition of this new instruction, and you were told that a new type IType \text{LWR} has been added for this instruction. Fill in the required changes to the Decode and Execute functions in order to support adding this instruction. Assume that the opcode for this new instruction is \text{7'b0001011}. We have included relevant portions of these functions below to simplify this task. Assume that the rest of the hardware implementation is the same as what you have seen in lecture and lab.

(A) (12 points) Fill in the required additions to the code below.

typedef enum { OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE, GCD, AUIPC, LWR, Unsupported} IType;

function DecodedInst decode(Bit#(32) inst);
    ...
    Bit#(7) opLWR = 7'b0001011; // or other available opcode
    ...
    Word immD = signExtend(1'b0); // default value
    
    DecodedInst dInst = unpack(0);
    dInst.iType = Unsupported;
    case(opcode)
        ...
        opLoad: begin
            case (funct3)
                default: dInst.iType = Unsupported;
            endcase
        end
        opLWR: dInst = DecodedInst {iType: LWR, dst: validDst, src1: src1, src2: src2, imm: immD, brFunc: Dbr, aluFunc: ?};
    endcase
    return dInst;
endfunction
Write your code below. Please add arrows to indicate where your code fits in the `decode` function.

```plaintext
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2, Word pc);
    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;

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

    Word nextPc = case (dInst.iType)
        JALR: (rVal1 + imm) & ~1; // "& ~1" clears the bottom bit.
        ...
        LWR: pc + 4;
        default: pc + 4;
    endcase;

    Word addr = case (dInst.iType)
        ...
        LWR: rVal1 + rVal2;
        default: 0;
    endcase;

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

Write your code below. Please add arrows to indicate where your code fits in the `execute` function.

6.004 Fall 2020 - 10 of 13 - Practice Quiz #2A
(B) (3 points) Could a store operation that calculated the address in the same manner as `lwrd` be added without major changes to the RISC-V ISA?

Yes  No

(C) (3 points) If you answered Yes to B then provide a brief summary of the required changes. If you answered No to B then explain why not.

A store instruction already makes use of rs1, and rs2 because it needs rs2 to hold the data to be written to memory. So this kind of address calculation would not be possible for a store operation.
Problem 6. Caches (17 points)

Consider a direct mapped cache shown above. It has 32 rows with a line size of 2 (2 words per line). Assume that data and addresses are 32 bits. Also assume that the cache has a valid bit and a dirty bit per row.

(A) (3 points) To ensure the best cache performance, which address bits should be used for the line offset (to select a word within the line)? Which bits should be used for the cache index? Which bits should be used for the tag field?

address bits used for line offset: \[ A[2:2] \]
address bits used for cache index: \[ A[7:3] \]
address bits used for tag field: \[ A[31:8] \]

(B) (2 points) Explain why your selection of bits in part (A) should result in the best cache performance.

To take advantage of locality you want contiguous memory locations to be able to live in the cache at the same time either in the same line or by mapping to a different index.

(C) (4 points) Could the four 32-bit addresses 0x10, 0x18, 0x88, and 0x90 all be resident in the cache at the same time? Explain why you chose your answer.

0x10 maps to index 2.
0x18 maps to index 3.
0x88 maps to index 17.
0x90 maps to index 18.
Since each address maps to a different index, all four addresses can be resident at the same time.

All four addresses can be resident at the same time: YES NO
(D) (3 points) Assume that checking the cache on each read takes 2 cycles and that refilling the cache line on a miss takes an additional 12 cycles. If we wanted the \textit{average} access time over many reads to be 2.4 cycles, what is the minimum hit ratio the cache must achieve during that period of time? You don’t need to simplify your answer.

\begin{align*}
a. \quad & = 2 + 12(1-\text{HR}) \\
\text{HR} & = \frac{11.6}{12}
\end{align*}

\textbf{minimum hit ratio for 2.4 cycle average access time:______} \textbf{11.6/12__________}

(E) (5 points) Now consider an 8 row direct mapped cache with a line size of 2 (2 words per line) as shown below. Suppose the cache is initially empty and the following sequence of addresses is repeatedly accessed: \textbf{0x0, 0x4, 0x10, 0x14, 0x18, 0x104, 0x210}. Assuming the sequence has been accessed many times, what will be the cache hit ratio for accessing this sequence of addresses the next time around? You may use the cache below to help you.

\begin{center}
\begin{tabular}{c|c|c|c|c}
  \textbf{V} & \textbf{D} & \textbf{Tag} & \textbf{Word 0} & \textbf{Word 1} \\
\hline
  row 0 & & & M[0x0] & M[0x4] \\
  & & & M[0x100] & M[0x104] \\
  row 1 & & & & \\
  row 2 & & & M[0x10] & M[0x14] \\
  & & & M[0x210] & \\
  row 3 & & & & \\
  row 4 & & & & \\
  row 5 & & & & \\
  row 6 & & & & \\
  row 7 & & & & \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
  \item 0x0: miss, bring in 0x0 and 0x4
  \item 0x4: hit
  \item 0x10: miss, bring in 0x10 and 0x14
  \item 0x14: hit
  \item 0x18: hit (loop repeats over and over)
  \item 0x104: miss, bring in 0x100 and 0x104
  \item 0x210: miss, bring in 0x210 and 0x214.
\end{itemize}

\textbf{Cache hit ratio:______} \textbf{3/7__________}

\textbf{END OF QUIZ 2!}