Problem 1. Sequential Design Problem (12 points)

You are asked to design module F which uses modules G and H to calculate results. As requests come in to F, they are sent to either G or H (as described below) and the results are collected and returned when they are ready. Modules G and H can handle multiple inputs simultaneously, but they are in-order processors so the results will always come out in the order that the requests were made. Module F should also obey this property. Module F should encapsulate G, H and a FIFO queue q as shown below.

Modules F, G, and H have the same interface, shown below, though internally they are completely different.

```
interface Generic;
    method Action start(Bit#(32) x);
    method ActionValue#(Bit#(32)) getResult;
endinterface
```

The FIFO interface is:

```
interface Fifo#(numeric type size, type t);
    method Action enq(t x);
    method Action deq;
    method t first;
endinterface
```
The `start` method of F calls the `start` method of Module G if \(x\) is even and the `start` method of Module H if \(x\) is odd. It also enqueues \(x\) into an internal queue \(q\). Suppose the first element of \(q\) is \(y\). If \(y\) is even, the `getResult` method of F calls the `getResult` method of G, otherwise it calls the `getResult` method of H.

(A) (8 points) Please complete the code for `mkF`, a module that implements module F.

```verilog
module mkF (Generic);
    Generic g <- mkG;
    Generic h <- mkH;
    FIFO q <- mkFIFO(...);

    method Action start (Bit#(32) x);
        (x[0] == 0)? g.start(x): h.start(x);
        q.enq(x);
    endmethod

    method ActionValue#(Bit#(32)) getResult;
        let val <= (q.first() == 0)? g.getResult(): h.getResult();
        q.deq();
        return val;
    endmethod
endmodule
```

(B) Assume that modules G and H take different amounts of time to process their input. Consider how much the performance is affected by the size of FIFO \(q\) to answer the following.

1. (2 points) True or false: \(q\) of size 2 is sufficient to get maximum performance

   Circle one: True ... False

2. (2 points) Provide a reason for your answer

   If one of the modules is very slow while the other is fast, to maximize performance you want to have the ability to have the fast module process multiple inputs while the slow module is computing its result. In order to support this, \(q\) must be able to hold multiple requests for the faster module while also holding one request for the slower module.
Problem 2. Guarded Interfaces (16 points)

(A) (8 points) Consider the 1-input finite state machine (FSM) with the state transition diagram shown below. The module definition for this FSM is incomplete. Complete the rule definitions for S0-S3 and add any needed guards. If no guards are needed then leave the if (____) blank.

```
typedef enum { S0, S1, S2, S3 } StateName deriving (Bits, Eq);

interface exampleFSM;
    method Action next(Bit#(1) input);
    method StateName getState();
endinterface

module mkFSM(exampleFSM);
    Reg#(StateName) state <- mkReg(S0);
    Reg#(Bool) update <- mkReg(False);
    Reg#(Bit#(1)) in <- mkReg(0);

    method Action next(Bit#(1) input) if (!update);
        update <= True;
        in <= input;
    endmethod

    method Bit#(StateName) getState();
        return state;
    endmethod

rule State0 if (____state == S0 & update_______);
    update <= False;
    state <= (__in__ == _1__)?____S0_____: ____S2____;
endrule

rule State1 if (____state == S1 & update_______);
    update <= False;
    state <= ((__in__ == _1__)?____S1_____: ____S0____;
endrule
```
rule State2 if (________state == S2 && update__________);
  update <= False;
  state <= ((__in__== __1__)?___S3_____: _____S1_____);
endrule

rule State3 if (________state == S3 && update__________);
  update <= False;
  state <= ((__in__== __1__)?___S0_____: _____S1_____);
endrule
endmodule

(B) (8 Points) In Lab 2, we often interacted with memory as if it was a stack. We’ve done so from a software perspective, using a dedicated register as our stack pointer. We interacted with the stack through memory requests, i.e., with loads and stores.

Suppose that we instead wish to have a small dedicated hardware stack, i.e., a Last-In-First-Out (LIFO) structure made with a vector of registers. Complete the definition of the stack module below by implementing the relevant interfaces. The stack is defined to hold 8 elements, each with a bit width of 32-bits. Include any needed guards. If a guard is not needed then leave the if (____) blank.

interface Stack;
  method Action push(Bit#(32) data);
  method ActionValue#(Bit#(32)) pop;
endinterface

module mkStack(Stack);
  Vector#(8, Reg#(Bit#(32))) buffer ← replicateM(mkRegU);  // Vector of n Uninitialized registers
  Reg#(Bit#(3)) ptr ← mkReg(0);
  Reg#(Bit#(4)) count ← mkReg(0);

  method Action push(Bit#(32) data) if (_____count < 8______________);
    ptr <= ptr + 1;
    count <= count + 1;
    buffer[ptr] <= data;
endmethod
method ActionValue#(Bit#(32)) pop if (____count > 0______________);
    ptr <= ptr - 1;
    count <= count - 1;
    return buffer[ptr - 1];
endmethod
endmodule

Problem 3. Sequential Logic in BSV (24 points)

The code on the following page implements a simple sequential circuit as a module that computes a function over a series of clock cycles. Read the code and answer the questions about it below. Note that the code contains blanks that you will fill in as part (A) of this problem. Each blank will contain one of method (and its type), rule, or function.

(A) (2 points) For each section of the code, specify if it is a function, method, or rule specification. For the methods, also specify the type of the method (Action, ActionValue, or Value).

<table>
<thead>
<tr>
<th>Method</th>
<th>Function, method, or rule</th>
<th>Method type (Action, ActionValue, or Value) or NA if not a method</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeShift</td>
<td>function</td>
<td>NA</td>
</tr>
<tr>
<td>doComputeStep</td>
<td>rule</td>
<td>NA</td>
</tr>
<tr>
<td>start</td>
<td>method</td>
<td>Action</td>
</tr>
<tr>
<td>getResult</td>
<td>method</td>
<td>ActionValue</td>
</tr>
</tbody>
</table>
interface Foo#(numeric type n);
    _______ start(Bit#(n) aIn, Bit#(TLog#(n)) bIn);
    _______ #(Bit#(1)) getResult();
endinterface

module mkFoo(Foo#(n));
    Reg#(Bit#(n)) a <- mkReg(0);
    Reg#(Bit#(TLog#(n))) b <- mkReg(0);
    Reg#(Bit#(n)) i <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);

    _______ Bit#(n) computeShift(Bit#(n) in);
        Bit#(n) out = 1;
        for (Bit#(n) i =0; i < in; i = i + 1) begin
            out = 2*out;
        end
        return out;
    end

    _______ doComputeStep if (busy && i != fromInteger(valueOf(TLog#(n))));
        let shift = computeShift(i);
        a <= (b[i] == 1)? a >> shift : a;
        i <= i + 1;
    end

    _______ start(Bit#(n) aIn, Bit#(TLog#(n)) bIn) if (!busy);
        a <= aIn;
        b <= bIn;
        i <= 0;
        busy <= True;
    end

    _______ #(Bit#(1)) getResult() if (busy && i ==
fromInteger(valueOf(TLog#(n))));
        busy <= False;
        return a[0];
    end
endmodule
(B) (6 points) The module using `mkFoo` is invoking all the methods of `mkFoo` at every clock cycle. Assume the following lines of code were run in the module before the zeroth clock cycle.

```plaintext
...
Foo#(8) dut <- mkFoo();
Bit#(8) a = 8'b01101001;
Bit#(3) b = 3'b011;

rule runFoo;
    dut.start(a, b);
...
```

In the table below, fill in the register values of the `mkFoo` module at the beginning of the following clock cycles. Assume t0 corresponds to the zeroth clock cycle, and so on.

<table>
<thead>
<tr>
<th>time value</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0b01101001</td>
<td>0b00110100</td>
<td>0b00001101</td>
<td>0b00001101</td>
<td>0b00001101</td>
<td>0b01101001</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>busy</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

(C) (6 points) Assume “`$display("Result is \%d", dut.getResult)"` is run in rule `getResult` of the module that is using `mkFoo`. This statement prints out the value of Result in decimal. Please answer the following questions. Remember that an invoked method can only execute when it is ready.

1. Please fill out the blank in the message shown in the console once the module is run.

   Result is ______1_______

2. Please write down an expression for what `computeShift` returns (in terms of i) and for the return value of the `getResult` method of `mkFoo` (in terms of a and b).

   ```plaintext
   computeShift returns: ______2^i__________ (in terms of i)
   
   getResult returns: ______a[b]___________ (in terms of a and b)
   ```
(D) (4 points) Check if the module is still functionally correct with the following changes (keeping everything else the same). You DO NOT need to fill in the blanks.

1. _______ doComputeStep if (busy && i != fromInteger(valueOf(TLog#(n)))));
   let shift = computeShift(i);
   i <= i + 1;
   a <= (b[i] == 1)? a >> shift : a;
   end__________  
   Circle one: True ... False

2. _______ doComputeStep if (i != fromInteger(valueOf(TLog#(n)))));
   let shift = computeShift(i);
   a <= (b[i] == 1)? a >> shift : a;
   i <= i + 1;
   end__________  
   Circle one: True ... False

3. _______ doComputeStep if (busy);
   let shift = computeShift(i);
   a <= (b[i] == 1)? a >> shift : a;
   i <= i + 1;
   end__________  
   Circle one: True ... False

4. _______ doComputeStep if (i != fromInteger(valueOf(TLog#(n)))));
   ...
   _______ start(Bit#(n) aIn, Bit#(TLog#(n)) bIn) if (a == 0);
   ...
   _______ #(Bit#(1)) getResult() if (i == fromInteger(valueOf(TLog#(n))));
   Circle one: True ... False
(E) (6 points) A partial circuit diagram for the implementation of `mkFoo` is given below.

1. (2 points) Complete the `start.rdy` and `getResult.rdy` signals by drawing wires from the tails of the green arrows to the proper register, function, or gate output. Use **TWO** basic logic gates.

2. (4 points) Write the Boolean expressions for each write enable signal of registers `a`, `b`, `i`, and `busy`. You may use **ONLY** wire names in the figure (e.g., `w1, w2, …, w6, start.en, getResult.en`) and at most a **total of eight** `AND(·), OR(+), or NOT(~)` gates across all expressions.

   - `a.en : _______ start.en + W_4 \cdot \sim W_5__________`
   - `b.en : _______ start.en ____________________________`
   - `i.en : _______ start.en + W_4 \cdot \sim W_5__________`
   - `busy.en : _____ start.en + getResult.en__________`
Problem 4. Multi-Cycle Processors (13 points)

The multicycle RISC-V processor built in Lab 6, which we call Processor A, uses:
1) a shared instruction/data memory with split request (req) and response (resp) methods,
2) a register file with two read ports (rd1, rd2 methods) and a write port (wr method).

Processor A uses three states (Fetch, Execute, LoadWait) to process instructions. It starts at Fetch and follows the state-transition diagram given below:

![State-Transition Diagram of Processor A](image)

Figure 1: State-Transition Diagram of Processor A

Ben Bitdiddle has developed a new design for the register file that’s smaller and can be clocked faster but has the odd property that writes take two cycles. On the first cycle, you give it the data you want to write and on the next cycle, you tell it which register to write to. Instead of a single wr method, it now has separate wrData and wrIndex methods which must be invoked in that order. Assume that the register file takes care of storing the data in some register until the index is transmitted. To handle the two cycle write, Ben adds an extra state to create Processor B with the following state diagram:

![State-Transition Diagram of Processor B](image)

Figure 2: State-Transition Diagram of Processor B

In Processor B, the wrData method is invoked in the same states that write to the register file in Processor A. Then, the RFwrite state completes the process by invoking the wrIndex method.

(A) (6 points) Assume that the memory responds three cycles after a request is made (e.g., if a load request is made on cycle 1, the memory replies in cycle 4). How many cycles would each instruction below take in Processor A and Processor B?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Processor A</th>
<th>Processor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor a4, a3, a1</td>
<td>4 cycles</td>
<td>5 cycles</td>
</tr>
<tr>
<td>blt a2, a3, loop</td>
<td>4 cycles</td>
<td>5 cycles</td>
</tr>
<tr>
<td>lw a3, 0x8(a2)</td>
<td>7 cycles</td>
<td>8 cycles</td>
</tr>
</tbody>
</table>
After looking more closely, Ben sees that some types of instructions could be executed in fewer cycles because they don’t need to use the RFwrite state. These instructions have already completed their work in the Execute state. Ben creates Processor C by adding a state transition from Execute directly back to Fetch, shown in red in the diagram below. In the diagram, there are now three paths leaving the Execute stage, labelled X, Y and Z. For each instruction, Ben can determine which path to take just by looking at the instruction’s iTy pe.

![Diagram](image.png)

Figure 3: State-Transition Diagram of Processor C

(B) (3 points) Fill in the table below; indicating which path (X, Y or Z) should be used for each iTy pe to properly execute it in the fewest number of cycles in Processor C.

<table>
<thead>
<tr>
<th>iTy pe</th>
<th>OP</th>
<th>OPIMM</th>
<th>BRANCH</th>
<th>LUI</th>
<th>JAL</th>
<th>JALR</th>
<th>LOAD</th>
<th>STORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Path</td>
<td>X</td>
<td>X</td>
<td>Z</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

(C) (4 Points) Alyssa P. Hacker looks at Ben’s design and realizes that he doesn’t actually need a separate RFwrite state at all and he could have kept the original state transition diagram from Processor A. Using Alyssa’s plan, which state would invoke the wrIndex method? What two pieces of information would need to be saved in registers for this state to use (assume the registers can only hold simple Bit values not structs)?

State: ____ Fetch ________________________

Info #1: ____ Register index to write ________

Info #2: _____ Valid bit (whether to write to the RF or not) ________
Problem 5. Bluespec Concurrency and Conflicts (18 points)

Consider the following trio of snippets of Bluespec code, each containing different versions of a pair of rules. In each version there are three registers x, y, and z that are being read from and written to.

Version 1

\[
\begin{align*}
\text{doA} & : x <= x + 1; \\
& \quad z <= z + x; \\
\text{doB} & : y <= z + 4;
\end{align*}
\]

Version 2

\[
\begin{align*}
\text{doA} & : x <= y + 2; \\
& \quad z <= x - 1; \\
\text{doB} & : y <= x + 1;
\end{align*}
\]

Version 3

\[
\begin{align*}
\text{doA} & : y <= x + y; \\
& \quad z <= z + x; \\
\text{doB} & : x <= x + 8;
\end{align*}
\]

(A) (6 points) For each version, give the value tuple (x,y,z) that the registers will have following a single execution of both rules given starting values of (3,6,1) with each rule ordering.

<table>
<thead>
<tr>
<th>Rule Ordering</th>
<th>Version 1</th>
<th>Version 2</th>
<th>Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent</td>
<td>(4,5,4)</td>
<td>(8,4,2)</td>
<td>(11,9,4)</td>
</tr>
<tr>
<td>doA &lt; doB</td>
<td>(4,8,4)</td>
<td>(8,9,2)</td>
<td>(11,9,4)</td>
</tr>
<tr>
<td>doB &lt; doA</td>
<td>(4,5,4)</td>
<td>(6,4,2)</td>
<td>(11,17,12)</td>
</tr>
</tbody>
</table>

(B) (6 points) Fill in the Conflict Matrix for each version, with C representing conflicting, CF representing conflict free, and < and > representing that the corresponding rule of the row appears to execute before and after the corresponding rule of the column, respectively.

<table>
<thead>
<tr>
<th>Version 1</th>
<th>doA</th>
<th>doB</th>
</tr>
</thead>
<tbody>
<tr>
<td>doA</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>doB</td>
<td>&lt;</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version 2</th>
<th>doA</th>
<th>doB</th>
</tr>
</thead>
<tbody>
<tr>
<td>doA</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>doB</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
(C) (6 points) For each version, select the expression that when removed from rule doA causes doA and doB to become CONFLICT FREE. If removing neither of the options would do so, then select Impossible. If the rules are already conflict free, then select Unnecessary.

<table>
<thead>
<tr>
<th>Version 3</th>
<th>doA</th>
<th>doB</th>
</tr>
</thead>
<tbody>
<tr>
<td>doA</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>doB</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

**Version 1**

\[ x \leq x + 1 \text{ OR } z \leq z + x \text{ OR } \text{Impossible} \text{ OR } \text{Unnecessary} \]

**Version 2**

\[ x \leq y + 2 \text{ OR } z \leq x - 1 \text{ OR } \text{Impossible} \text{ OR } \text{Unnecessary} \]

**Version 3**

\[ y \leq x + y \text{ OR } z \leq z + x \text{ OR } \text{Impossible} \text{ OR } \text{Unnecessary} \]
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 1.
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 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.

\[ 2.4 = 2 + 12(1 - HR) \]

\[ HR = \frac{11.6}{12} \]

minimum hit ratio for 2.4 cycle average access time: \( \frac{11.6}{12} \)

(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: \( 0x0, 0x4, 0x10, 0x14, 0x18, 0x104, 0x210 \). 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.

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>M[0x0]</td>
<td>M[0x4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M[0x100]</td>
<td>M[0x104]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M[0x10]</td>
<td>M[0x14]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M[0x210]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M[0x18]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x0: miss, bring in 0x0 and 0x4
0x4: hit
0x10: miss, bring in 0x10 and 0x14
0x14: hit
0x18: hit (loop repeats over and over)
0x104: miss, bring in 0x100 and 0x104
0x210: miss, bring in 0x210 and 0x214.

Cache hit ratio: \( \frac{3}{7} \)

END OF QUIZ 2!