Problem 5. Bluespec Concurrency (9 points)

You are designing some hardware in Bluespec, and are trying to get ruleA and ruleB to execute concurrently. Both rules access FIFOs f and g.

```
rule ruleA;
  if (g.first) begin
    f.enq(g.first + 1);
    g.deq;
  end else begin
    f.enq(0);
  end
endrule

rule ruleB;
  g.enq(f.first + 1);
  f.deq;
endrule
```

If f is a Pipeline FIFO, what is the scheduling constraint between ruleA and ruleB if FIFO g is each of the following FIFO types? Answer CF, <, >, or C for each of the following.

- g is a Pipeline FIFO: ruleA __C__ ruleB
- g is a Bypass FIFO: ruleA __>__ ruleB
- g is a Conflict-Free FIFO: ruleA __>__ ruleB

Conflict Matrices for FIFOs:

<table>
<thead>
<tr>
<th>Pipeline FIFO</th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>deq</td>
<td>&lt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>&lt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bypass FIFO</th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>deq</td>
<td>&gt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>&gt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conflict-Free FIFO</th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>CF</td>
<td>CF</td>
</tr>
<tr>
<td>deq</td>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>
Problem 6. Caches (20 points)

Consider a direct-mapped cache, Cache1, shown on the right. It has 8 lines with a block size of 1 (1 word per line). Assume that data and addresses are 32 bits. Also assume that the cache has a valid bit and a dirty bit per line.

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) (2 points) To ensure the best cache performance for Cache1, which address bits should be used for the block/byte offset? Which bits should be used for the cache index? Which address bits should be used for the tag field?

- address bits used for block/byte offset: A[1:0]
- address bits used for cache index: A[4:2]
- address bits used for tag field: A[31:5]

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

The lower-order bits of the address change more often, so using them as the index bits reduces conflict misses.

(C) (2 points) How many bits of SRAM are required to store the contents of Cache1? Include all the state of each line, not just the data.

Bits per line = valid + dirty + tag + data = 2 + 27 + 32 = 61.
Total bits = 61 * 8 lines = 488.

Bits of SRAM for Cache1: 488

(D) (3 points) Could the four 32 bit addresses 0x00, 0x10, 0x18, and 0x3C all be resident in the cache at the same time? Explain why you chose your answer.

0x00: index 0; 0x10: index 4; 0x18: index 6; 0x3C: index 7
Since each address maps to a different index, these addresses don’t conflict in the cache and can all be resident at the same time.

All four addresses can be resident at the same time: YES  NO
Now consider **Cache2**, which also stores a total of 8 words but its configuration is a 4 line direct-mapped cache with block size of 2 (2 words per line).

(E) (2 points) For **Cache2**, which address bits should be used for the block/byte offset? Which bits should be used for the cache index? Which address bits should be used for the tag field?

- **address bits used for block/byte offset**: \( A[2:0] \)
- **address bits used for cache index**: \( A[4:3] \)
- **address bits used for tag field**: \( A[31:5] \)

(F) (3 points) Could the four 32 bit addresses 0x00, 0x10, 0x18, and 0xC all be resident in **Cache2** at the same time? **Explain why you chose your answer.**

- 0x00: index 0; 0x10: index 2; 0x18: index 3; 0xC: index 3
- 0x18 and 0xC conflict with each other because they both map to line 3.

All four addresses can be resident at the same time: **YES NO**

(G) (6 points) Find the cache hit ratio for the following loop on each of the two caches. The loop is executed for many iterations, so report the average behavior of a loop iteration and ignore cold-start effects. Assume the cache is empty before execution begins (all valid bits are 0). Remember the cache is used for both instruction and data accesses.

```
. = 0x100  // Code starts at address 0x100
// Assume x1 is initialized to 0, x2 is initialized to 2000 (number of array elements), and x3 is initialized to the base address of array A.

loop: addi x2, x2, -1
      slli x4, x2, 2    // x4 holds byte offset of array element x2
      add x4, x3, x4   // x4 holds address of element x2
      lw x5, 0(x4)     // x5 holds element x2 of array
      add x1, x1, x5   // x1 holds sum of array elements
      add x0, x0, x0   // dummy instruction
      add x0, x0, x0   // dummy instruction
      bnez x2, loop
```

Per loop iteration: 8 instr fetches and 1 data fetch. In **Cache1**, data word keeps replacing one of the instructions so each iteration you have 7 instr hits, 1 instr miss, and 1 data miss. **Cache2** has the same behavior (a data line replaces a line with instructions on each iteration).

**Cache1 hit ratio:** __7/9________

**Cache2 hit ratio:** __7/9________

END OF QUIZ 2!
Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for partial credit. You can use the extra white space and the backs of the pages for scratch work.

This quiz has a separate appendix with the 6.004 RISC-V ISA reference tables.

Problem 1. Sequential Logic in BSV (24 points)

The following code implements a simple sequential circuit as a module that computes a function over a series of steps. Read the code and answer the questions about it below.

```hs
interface Foo;
    method Action start(Bit#(32) aIn);
    method ActionValue#(Bit#(32)) getX();
    method Bit#(32) getI();
endinterface

module mkFoo(Foo);
    Reg#(Bit#(32)) a <- mkReg(0);
    Reg#(Bit#(1)) validx <- mkReg(0);
    Reg#(Bit#(32)) x <- mkRegU();
    Reg#(Bit#(32)) i <- mkRegU();

    function Bit#(32) computeB(Bit#(32) in);
        Bit#(32) out = 0;
        if ( in >= 1 ) out = 1;
        if ( in >= 5 ) out = 5;
        if ( in >= 10 ) out = 10;
        return out;
    endfunction

    rule doComputeStep if (a > 0 && validx == 0);
        let b = computeB(a);
endrule
endmodule
```
a <= a - b;
x <= a;
validx <= 1;
i <= i + 1;
endrule

method Action start(Bit#(32) aIn) if (a==0);
a <= aIn;
i <= 0;
endmethod

method ActionValue#(Bit#(32)) getX() if (validx == 1);
validx <= 0;
return x;
endmethod

method Bit#(32) getI() if (a==0);
return i;
endmethod
endmodule
(A) (8 points) The module using mkFoo is invoking all the methods of mkFoo at every clock cycle. Remember that an invoked method can only execute when it is ready. The register values at the beginning of t0 is given. Fill the register values at the beginning of the following four clock cycles in the table below.

<table>
<thead>
<tr>
<th>time value</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>17</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>validx</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>27</td>
<td>17</td>
<td>17</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

(B) (4 points) The module using mkFoo is invoking the \texttt{getX} method of mkFoo at every clock cycle. Remember that an invoked method can only execute when it is ready. If the \texttt{start} method is called the first time with \texttt{aIn = 28}, what will the output sequence from \texttt{getX()}? What is the output of \texttt{getI()} after the \texttt{start} method is called?

1. Return value sequence of \texttt{getX()}: \texttt{28, 18, 8, 3, 2, 1}

2. Return value of \texttt{getI()}: \texttt{6}

(C) (2 points) Suppose we get rid of register \texttt{x} and modify the rule \texttt{doComputeStep} and method \texttt{getX} as follows.

\begin{verbatim}
rule doComputeStep if (a > 0 && validx == 0);
    let b = computeB(a);
    a <= a - b;
    x <= a;
    validx <= 1;
    i <= i + 1;
endrule

method ActionValue#(Bit#(32)) getX() if (validx == 1);
    validx <= 0;
    return a;
return x;
endmethod
\end{verbatim}

Does this change the output sequence of \texttt{getX()} of the module?

(circle one) \textbf{Yes} ... No ... Can't tell
(D) (2 points) Ignoring the changes in (C), suppose we modify the guard of `start` in the original code to `(a==0 && validx == 0)`. Does this change the output sequence of `getX()`?

(circle one)   Yes   ...   No   ... Can’t tell

(E) (8 points) A partial circuit diagram for the implementation of mkFoo is given below.

Give the Boolean expressions for each write enable signal of registers `a`, `validx`, `x`, and `i`. You may use only wire names from the picture (e.g., `t1`, `t2`, ..., `t9`, `start.en`, `getX.en`, `a`, `validx`, `x`, `i`) and at most a total of three AND(·), OR(+), or NOT(−) gates across all expressions.

- `a.en : start.en + t5`
- `validx.en : getX.en + t5`
- `x.en : t5`
- `i.en : start.en + t5`
Problem 2. Processor Implementation (26 points)

The multicycle RISC-V processor built in Lab 5, 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:

Instead of using a register file with two read ports, we want to build another multicycle RISC-V processor, Processor B, that uses a register file with only one read port (rd method). Since it is not possible to read two registers in the same cycle as in Execute of Processor A, we split Execute into two states, namely Execute1 and Execute2, where only one register is read in each of the new rules. The state-transition diagram of Processor B is implemented as follows:

(A) (6 points) Assume that the memory responds two cycles after a request is made (e.g., if a load request is made on cycle 1, the memory replies in cycle 3). How many cycles does 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>addi a4, a3, 0x134</td>
<td>3 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>lw a3, 4(a2)</td>
<td>5 cycles</td>
<td>6 (or 5*) cycles</td>
</tr>
<tr>
<td>bne a2, a3, 0x120</td>
<td>3 cycles</td>
<td>4 cycles</td>
</tr>
</tbody>
</table>

(*) In processor B, a load can be issued at Execute1 instead of Execute2. This will take 5 cycles. After examining Processor B, we noticed that we can optimize it by having certain types of instructions (iType) spend fewer cycles. Processor C implements this optimization by completing the execution of these types of instructions in Execute1, and adding an extra state transition from Execute1 to Fetch. The state transition diagram is given in the next page and the suggested new transition is marked in red where iSET is a subset of iType values:

{ OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE }
(B) (4 points) For each iType, place a check mark if the iTYPE can be in iSET to correctly implement Processor C (i.e., if this instruction can be executed completely in two states, Fetch and Execute1).

<table>
<thead>
<tr>
<th>iType</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>In iSET?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All instructions with 0 or 1 source registers, except LOAD (which needs to go to LoadWait)

(C) (6 points) You can further optimize Processor C for LOAD instructions so that they take fewer cycles in the processor. Finish the state-transition diagram below only for LOAD instructions to implement such optimization by drawing state-transition arrows.

Let’s implement Processor B using Bluespec. The almost complete code is given below. This code is similar to the code in Lab 5. The main differences are highlighted in red.

Your job is to fill in the doExecute1 rule. Consider the following:
1) The new register file is implemented as a module named mkRFile1R1W whose x0 register is always zero and its interface RFile1R1W is given along with the code.
2) Types related to decode and execute functions are given along with the code.
3) You are only allowed to fill in the body of doExecute1. Other lines should remain the same.

For example, you cannot add extra registers.

(D) (10 points) Complete the doExecute1 rule for Processor B.

typedef Bit#(32) Word;
typedef Bit#(5) RIndx;

interface RFile1R1W;
    method Word rd(RIndx rindx); // Only 1 read port
    method Action wr(RIndx rindx, Word data);
endinterface

typedef enum {Fetch, Execute1, Execute2, LoadWait} State deriving (Bits, Eq);

// code continues on the next page
// types related to decode and execute functions
typedef struct {
    IType iType; AluFunc aluFunc; BrFunc brFunc;
    Word imm; Maybe<(RIndx) dst; RIndx src1; RIndx src2;
} DecodedInst;

typedef struct {
    IType iType; Maybe<(RIndx) dst;
    Word data; Word addr; Word nextPc;
} ExecInst;

function DecodedInst decode(Word inst);
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2, Word pc);

// Processor B implementation
module mkProcMulticycleB(Empty);
    Reg#(Word) pc <- mkReg(0);
    RFile1R1W rf <- mkRFile1R1W; // Register File with 1 read port
    Memory mem <- mkMemory;
    Reg#(State) state <- mkReg(Fetch);
    Reg#(Word) rVal1_e1 <- mkRegU;
    Reg#(DecodedInst) dInst_e1 <- mkRegU;
    Reg#(RIndx) dstLoad <- mkReg(0);

    rule doFetch if (state == Fetch);
        mem.req(MemReq{op: Ld, addr: pc, data: ?});
        state <= Execute1;
    endrule

    rule doExecute1 if (state == Execute1);
        // Write your code in this box

        let inst <- mem.resp();
        let dInst = decode(inst);
        rVal1_e1 <= rf.rd(dInst.src1);
        dInst_e1 <= dInst;
        state    <= Execute2;
    endrule

    // code continues on the next page
rule doExecute2 if (state == Execute2);
  let rVal2 = rf.rd(dInst_e1.src2);
  let eInst = execute(dInst_e1, rVal1_e1, rVal2, pc);
  if(eInst.iType == Unsupported) begin
    $display("[Halt] Reached unsupported instruction (0x%x)", inst);
    $finish;
  end else if (eInst.iType == LOAD) begin
    mem.req(MemReq{op: Ld, addr: eInst.addr, data: ?});
    dstLoad <= fromMaybe(?, eInst.dst); state <= LoadWait;
  end else if (eInst.iType == STORE) begin
    mem.req(MemReq{op: St, addr: eInst.addr, data: eInst.data});
    pc <= pc + 4; state <= Fetch;
  end else begin
    if(isValid(eInst.dst)) begin
      rf.wr(fromMaybe(? , eInst.dst), eInst.data); 
    end
    pc <= eInst.nextPc; state <= Fetch;
  end
endrule

rule doLoadWait if (state == LoadWait);
  Word data <- mem.resp();
  rf.wr(dstLoad, data);
  pc <= pc + 4;
  state <= Fetch;
endrule
endmodule