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.

```bsv
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
```

1. What is the function `computeB` used for in this code?
2. How many steps does the circuit take to compute the function?
3. What is the initial value of the `a` register?
4. What is the final value of the `x` register after the circuit has been executed?
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 getX method of mkFoo at every clock cycle. Remember that an invoked method can only execute when it is ready. If the start method is called the first time with aIn = 28, what will the output sequence from getX()? What is the output of getI() after the start method is called?

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

2. Return value of getI(): 6

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

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

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

Does this change the output sequence of getX() of the module?

(circle one) 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(\(\cdot\)), OR(\(+\)), or NOT(\(\sim\)) gates across all expressions.

\[
\begin{align*}
a.en & : \quad \text{start.en} + t5 \\
validx.en & : \quad \text{getX.en} + t5 \\
x.en & : \quad t5 \\
i.en & : \quad \text{start.en} + t5
\end{align*}
\]
**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:

![Figure 1: State-Transition Diagram of Processor A](image)

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:

![Figure 2: State-Transition Diagram of Processor B](image)

(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><em>Processor A</em></th>
<th><em>Processor B</em></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 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;
        dstLoad <= mkReg(0);
        state <= Execute2;

    endrule

// code continues on the next page
rule doExecute2 if (state == Execute2);
    let rVal1 = rf.rd(dInst_e1.src1);
    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
        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
Problem 3. Caches (22 points)

Consider the two-way set associative cache shown below. It has 8 sets and a block size of 4 (4 words per line). The column labeled \( \text{Data}_x \) corresponds to the \( x \)th word of the block. The V bit specifies whether the line is valid, and the D bit specifies whether the line is dirty. Assume that the data and addresses are 32 bits.

(A) (2 points) Which address bits correspond to the block offset? Which bits provide the cache index? Which bits make up the tag field?

address bits used for block offset: \( A[3 : 0] \)


tag field: \( A[31 : 7] \)

(B) (3 points) Would a load request to address 0x08C8 result in a hit or a miss? If it results in a hit, specify what value is returned; if it is a miss, write NA.

Tag = 0x11, index = 4, block offset = 8 bytes = 2 words

Hit / Miss: ___HIT____

Returned value if hit or NA if miss: ___0x87____

(C) (3 points) Would a load request to address 0x0910 result in a hit or a miss? If it results in a hit, specify what value is returned; if it is a miss, write NA.

Tag = 0x12, index = 1, block offset = 0

Hit / Miss: ___HIT____

Returned value if hit or NA if miss: ___0x4B____
(D) (2 points) Would the following instruction result in a hit or a miss?

\[
\text{sw } x0, 0x104(x0) \quad 0x104 \rightarrow \text{tag} = 0x02, \text{index} = 0, \text{block offset} = 4 \text{ bytes} = 1 \text{ word}
\]

Hit / Miss: ___MISS____

(E) (4 points) Which entries of the cache would be modified by executing the above instruction, \text{sw x0, 0x104(x0)}? Assume that Way 0 (the left one) is the least recently used way. Fill in the empty cache line below to indicate the changes that would be made to the cache from executing this instruction. Specify which set of way 0 is modified as well as the contents of valid, dirty, tag, and data words. For any unknown values, enter '"?''.

Way 0

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data 0</th>
<th>Data 1</th>
<th>Data 2</th>
<th>Data 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0x02</td>
<td>?</td>
<td>0x0</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

(F) (8 points) Find the steady state cache hit ratio for the following loop on this two-way set associative cache with an LRU (least recently used) replacement policy. 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 that the cache is used for both instruction and data accesses.

\[
. = 0x400 \quad \text{// Code starts at address 0x400}
\]

// Assume x1 is initialized to 0, x2 is initialized to 1000 (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
bnez x2, loop

Instruction miss rate = 0
Data miss rate = ¼ (one miss every 4 iterations)
Every 4 iterations: 24/24 instruction hits + ¾ data hits \(\rightarrow\) Hit rate is 27/28

Steady-state hit ratio: __27/28________