Problem 1. Sequential Logic in BSV (15 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.

```
interface Seq;
    method Action start(Bit#(32) a);
    method ActionValue#(Bit#(32)) getI;
    method ActionValue#(Bit#(32)) getY;
endinterface

module mkSeq(Seq);
    Reg#(Bit#(32)) i <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bit#(32)) x <- mkReg(0);

    rule computeStep if (x > 1);
        i <= i+1;
        if (x > y) y <= x;
            if (x[0] == 1) x <= x + x + x + 1;
            else x <= x >> 1;
    endrule

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

    method Bit#(32) getI if (x == 1);
```

Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. You can use the extra white space and the backs of the pages for scratch work.

This quiz has a separate appendix that includes the RISC-V calling convention and the 6.S084 RISC-V reference guide.
```verilog
return i;
endmethod

method Bit#(32) getY if (x == 1);
    return y;
endmethod

method Action clearResults if (x == 1);
    x <= 0;
endmethod
endmodule

(A) (3 points) If at the beginning of `computeStep` the values in the registers are \( i = 17 \), \( y = 25 \), and \( x = 11 \), what are the values after rule `computeStep` runs once?

1. \( y = \)

2. \( x = \)
(B) (5 points) If start is called with value \(a = 10\), what will the output of \(\text{getI}()\) and \(\text{getY}()\) be when it is ready?

1. Return value of \(\text{getI}()\) =

2. Return value of \(\text{getY}()\) =

(C) (5 points) We have verified that for all possible 32-bit values of \(a\) except 0, when the module is started with \(\text{start}(a)\), the computeStep rule eventually writes 1 to \(x\) and results become ready.

Assume that \(a\) is a nonzero value. Let’s denote the sequence of \(k\) values that \(x\) goes through as \([x_0, x_1, ..., x_{k-1}]\). Note that \(x_0 = a\) (the initial value written by \(\text{start}\)), and \(x_{k-1} = 1\) (the final value written by \(\text{computeStep}\) after some number of cycles).

Write expressions valid for all \(a > 0\) for the return values of \(\text{getI}()\) and \(\text{getY}()\) as a function of \(k\) and \([x_0, x_1, ..., x_{k-1}]\).

1. Expression for \(\text{getI}()\) =

2. Expression for \(\text{getY}()\) =

(D) (2 points) Suppose we modify the computeStep rule as shown below. Does this change the functionality of the module?

<table>
<thead>
<tr>
<th>Original code</th>
<th>Modified code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{rule} computeStep if (x &gt; 1);</td>
<td>\text{rule} computeStep if (x &gt; 1);</td>
</tr>
<tr>
<td>\hspace{1cm} i \hspace{1cm} &lt;= \hspace{1cm} i+1;</td>
<td>\hspace{1cm} i \hspace{1cm} &lt;= \hspace{1cm} i+1;</td>
</tr>
<tr>
<td>\hspace{1cm} if (x &gt; y) y \hspace{1cm} &lt;= \hspace{1cm} x;</td>
<td>\hspace{1cm} if (x[0]==1) x \hspace{1cm} &lt;= \hspace{1cm} x + x + x + 1;</td>
</tr>
<tr>
<td>\hspace{1cm} if (x[0]==1) x \hspace{1cm} &lt;= \hspace{1cm} x + x + x + 1;</td>
<td>\hspace{1cm} else \hspace{1cm} x \hspace{1cm} &lt;= \hspace{1cm} x &gt;&gt; 1;</td>
</tr>
<tr>
<td>\hspace{1cm} \text{endrule}</td>
<td>\hspace{1cm} \text{endrule}</td>
</tr>
</tbody>
</table>

Does this change the functionality of the module? (circle one) Yes ... No ... Can’t tell
Problem 4. Hardware Implementation (25 points)

While our RISC-V processor has 32-bit registers and operates on 32-bit values, it is very common for programs to manipulate larger integer values, e.g., 64 or 128 bits long. In this problem, you will study the cost of using our 32-bit RISC-V processor to add these large integers, and extend our processor to make these additions faster.

To add large (e.g., 64-bit) numbers using 32-bit addition, we can for the most part use a sequence of 32-bit add instructions, adding large numbers in 32-bit chunks. The key issue is propagating the carry bit between these additions. Remember the design of a ripple-carry adder, shown on the right. You can build a large ripple-carry adder from two smaller ones by connect the carry-out of one to the carry-in of the other. The example on the right shows how to connect two 1-bit full adders to form a 2-bit ripple-carry adder. We will leverage this same principle, but using the 32-bit adder in our ALU.

(A) (6 points) Complete the assembly code below to add two 64-bit unsigned integers and produce a 64-bit unsigned integer result. Consider the following:

- Each 64-bit input and output is stored in two 32-bit registers. We use the following notation: \{a1, a0\} denotes that 32-bit registers a1 and a0 represent a single 64-bit number, a1 stores the upper 32 bits, and a0 stores the lower 32 bits.
- The numbers to be added are in \{a1, a0\} and \{a3, a2\}, and your code should produce the result in \{a5, a4\}.
- The code below shows the first two instructions of the solution. This code does not work as-is because it does not propagate the carry bit from the sum of the lower 32 bits into the sum of the upper 32 bits. Your code should handle this carry bit.
- For full credit, your code should use a total of at most four instructions (implementations with more instructions will earn partial credit).
- **Hint:** Although the `add` instruction does not store the carry bit anywhere, you can find the carry with a single comparison of the destination register and any one of the source registers.

\[
\begin{align*}
\text{add } & a4, a2, a0 \\
\text{add } & a5, a3, a1
\end{align*}
\]

(B) (3 points) Does your code also work on signed integers in two’s complement encoding? Briefly explain why or why not.
Dissatisfied with the performance of this code, Ben Bitdiddle proposes to extend the RISC-V ISA to store and use the carry bit. Specifically, Ben proposes the following changes:

1. Add a new 1-bit register, carry.
2. Add a new `addc` instruction (add-with-carry) that uses the value in the `carry` register as its carry-in, and also writes its carry-out bit to the `carry` register:

   ```
   addc rd, rs1, rs2 :
   {carry, reg[rd]} <= {1'b0, reg[rs1]} + {1'b0, reg[rs2]} + carry
   ```

(C) (4 points) Write assembly code that uses `addc` to add two 64-bit unsigned integers and produce a 64-bit unsigned integer result as in question (A). **Assume the carry bit is initially 0.** For full credit, your code should use the smallest possible number of instructions.

Let’s add Ben’s ISA extensions to our single-cycle processor implementation. Assume that we have added a new `IType ADDC` as shown below, and have already modified the decode function to decode `addc` instructions appropriately.

```
typedef enum { OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE, AUIPC, ADDC, Unsupported } IType deriving (Bits, Eq, FShow);
```

We want to modify the `execute` function to provide the carry output and update the carry register every instruction, as shown in the module below. New code is highlighted in red.

```
typedef struct {
  IType iType;
  Maybe#(RIndx) rd;
  Word data;
  Word addr;
  Word nextPc;
  Bit#(1) carry;
} ExecInst deriving (Bits, Eq, FShow);
```

```
module mkProc(Empty);
  Reg#(Bit#(1)) carry <- mkReg(0);
  Reg#(Word)  pc <- mkReg(0);
  ...
  rule doProc;
  ...
  let eInst = execute(dInst, rVal1, rVal2, pc, carry);
      // Always update the carry bit
      carry <= eInst.carry;
      // Other state updates
  ...
endrule
endmodule
```
(D)(12 points) Modify the execute function below to implement the `addc` instruction.

```markdown
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2, Word pc, Bit#(1) carry);

let imm = dInst.imm;
let brFunc = dInst.brFunc;
let aluFunc = dInst.aluFunc;
Word data = ?;
Word nextPc = ?;
Word addr = ?;

**Bit#(1) newCarry = carry;**

```case (dInst.iType) matches
    OP: begin data = alu(rVal1, rVal2, aluFunc); nextPc = pc+4; end
    OPIMM: begin data = alu(rVal1, imm, aluFunc); nextPc = pc+4; end
    ADDC: // Fill in
    BRANCH: begin nextPc = aluBr(rVal1, rVal2, brFunc) ? pc+imm : pc+4; end

endcase
```

ExecInst eInst = ?;
eInst.iType = dInst.iType;
eInst.rd = dInst.rd;
eInst.data = data;
eInst.addr = addr;
eInst.nextPc = nextPc;
eInst.carry = newCarry;
return eInst;
endfunction
```

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

```plaintext
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 _____ ruleB
- g is a Bypass FIFO: ruleA _____ ruleB
- g is a Conflict-Free FIFO: ruleA _____ ruleB

Conflict Matrices for FIFOs:

### Pipeline FIFO

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

### Bypass FIFO

<table>
<thead>
<tr>
<th></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>

### Conflict-Free FIFO

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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[_____:_____]
- address bits used for cache index: A[_____:_____]
- address bits used for tag field: A[_____:_____]

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

(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 of SRAM for Cache1: ____________

(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.

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:** \( A[____:____] \)
- **address bits used for cache index:** \( A[____:____] \)
- **address bits used for tag field:** \( A[____:____] \)

(F) (3 points) Could the four 32 bit addresses 0x00, 0x10, 0x18, and 0x3C all be resident in Cache2 at the same time? **Explain why you chose your answer.**

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.

```assembly
. = 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
```

**Cache1 hit ratio:**

**Cache2 hit ratio:**

END OF QUIZ 2!