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. Assembly (18 points)

(A) (6 points) For the RISC-V instruction sequence below, provide the hex values of the specified registers after the sequence has been executed. Assume the program is located beginning at address 0x0 in memory and that all registers are initialized to 0.

```
. = 0x0
lui x4, 0x2
jal x5, L1
ori x4, x4, 0x3
L1: addi x0, x4, 0x16
    addi x6, x0, 0x200
```

Value of x4 (in hex) $0x2000$

Value of x5 (in hex) $0x8$

Value of x6 (in hex) $0x200$
(B) (4 points) You are given the following C code along with a partial translation into RISC-V assembly that is missing a single instruction (the first one). Please complete the assembly implementation by specifying the single missing instruction. Assume that \( x \) and \( y \) are signed 32-bit integers (defined as `int x; int y;`).

```c
if (x > y) {
    x = 5;
} else {
    y = 10;
}
```

// Partial assembly implementation
// Registers – x: x1, y: x2

```riscv
bge x2, x1, else________ (fill in the missing instruction)
li x1, 5
j endif
else:
    li x2, 10
endif:
```

(C) (8 points) Consider the following code sequence. This code is located beginning at address 0x0 in memory. Note that this code may modify itself.

```riscv
.L = 0x0
    bge x1, x0, L1
    lw x2, 0xC(x0)
    sw x2, 0x10(x0)
.L1: addi x1, x1, 2
    addi x1, x1, -1
```

(i) What is the address of label L1?

0xC

(ii) What are the final values of the register x1, when this code is run with the following initial values for x1?

If initial value of x1 = 5 \( \rightarrow \) Final value of x1 = \_]\_6\_\_

bge taken, final x1 = 5 (initial value) + 2 (addi) – 1 (2\(^{nd}\) addi)

If initial value of x1 = -5 \( \rightarrow \) Final value of x1 = \_]\_1\_\_

bge not taken, lw and sw overwrite 2\(^{nd}\) addi at 0x10 with addi at 0xC
final x1 = -5 (initial value) + 2 (addi) + 2 (addi written by sw)
Problem 2. RISC-V Assembly and Calling Convention (16 points)

(A) (6 points) The factorial C procedure given below implements the factorial function (e.g., factorial(5) = 5! = 5 x 4 x 3 x 2 x 1 = 120). This implementation uses a loop to compute the factorial. Since our RISC-V processor does not have a multiply instruction, the code uses a multiply procedure, mult, which multiplies two signed integers.

Ben Bitdiddle decides to implement this procedure in RISC-V assembly. But his assembly code, shown below, is not behaving as expected. Please fix the bugs so that his code works correctly and obeys the RISC-V calling convention.

The implementation of the mult procedure is not provided to you, but it does obey the calling convention.

```
int factorial(int n) {
    int prod = 1;
    for (int i = 1; i <= n; i++) {
        prod = mult(prod, i);
    }
    return prod;
}
```

```
factorial:
    addi sp, sp, -16
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw s2, 8(sp)
    sw ra, 12(sp)
    mv s0, a0
    li s1, 1 // set up prod
    li s2, 1 // set up i
loop:
    blt s0, s2, done
    mv a0, s1
    mv a1, s2
    jal ra, mult
    mv s1, a0
    addi s2, s2, 1
    j loop
done:
    lw s0, 0(sp)
    lw s1, 4(sp)
    lw s2, 8(sp)
    lw ra, 12(sp)
    addi sp, sp, 16
    jr ra // ret
```

Provide the correct assembly code here. Your implementation should obey the RISC-V calling convention. Note that there may be multiple errors in the code. Assume \(n \geq 1\).
(B) (10 points) The `sumSquares` C procedure given below computes the sum-of-squares function. Specifically, the `sumSquares` procedure takes a single argument `n` and computes $\sum_{i=1}^{n} i^2$ (e.g., `sumSquares(5) = 5^2 + 4^2 + 3^2 + 2^2 + 1^2 = 55`). This implementation is recursive. Again, since our RISC-V processor does not have a multiply instruction, the code uses a multiply procedure, `mult`, which multiplies two signed integers.

Ben Bitdiddle decides to implement this procedure in RISC-V assembly. But his assembly code, shown below, is not behaving as expected. Please answer the question below and fix the bugs so that his code works correctly and obeys the RISC-V calling convention.

The implementation of the `mult` procedure is not provided to you, but it does obey the calling convention.

```c
int sumSquares(int n) {
    if (n == 0)
        return 0;
    return sumSquares(n-1) + mult(n,n);
}
```

Ben’s buggy assembly code is given below. Provide the full instruction that the PC points to after executing `jr ra` (i.e., `ret`) for the first time in the given buggy code. Assume $n \geq 1$.

```assembly
sumSquares:
    bltz a0, done
    addi sp, sp, -12
    sw a0, 0(sp)
    addi a0, a0, -1
    jal ra, sumSquares
    mv s0, a0
    mv a1, a0
    j mult
    add a0, a0, s0
    lw a0, 0(sp)
    addi sp, sp, 4
done:
    jr ra  // ret
```

Provide the correct assembly code here. Your implementation should obey the RISC-V calling convention. Note that there may be multiple errors in the code.

```assembly
sumSquares:
    beqz a0, done
    addi sp, sp, -12
    sw a0, 0(sp)
    sw s0, 4(sp)
    sw ra, 8(sp)
    addi a0, a0, -1
    jal ra, sumSquares
    mv s0, a0
    lw a0, 0(sp)
    mv a1, a0
    jal ra, mult
    add a0, a0, s0
    lw s0, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, 12
done:
    jr ra  // ret
```

`mult: ...`
Problem 3. Procedures and Stacks (18 points)

Consider the C procedure below and its translation to RISC-V assembly code, shown on the right.

```c
int f(int a) {
    if (a == 1)
        return 0;
    if (a & 1 == 0) {
        // a is even
        return 1 + f(a / 2);
    } else {
        // a is odd
        return 1 + f(a * 3 + 1);
    }
}
```

(A) (3 points) Give the HEX encoding of the `slli a1, a0, 1` instruction.

```
00000000|shamt| rs1 |001| rd |0010011
00000000|00001|01010|001|01011|0010011
0x 0 0 1 5 1 5 9 3
```

Hex encoding of `slli a1, a0, 1`: 0x 00151593

(B) (2 points) How much total stack memory does the program use to calculate f(3)?

```
f(3) \rightarrow f(10) \rightarrow f(5) \rightarrow f(16) \rightarrow f(8) \rightarrow f(4) \rightarrow f(2) \rightarrow f(1)
```

Number of words pushed onto the stack for f(3)? 7
The program’s initial call to function $f$ occurs outside of the function definition via the instruction ‘jal ra, $f’’. The program is interrupted at an execution (not necessarily the first) of function $f$, just prior to the execution of li a1, 1 at label $f$. The diagram on the right shows the contents of a region of memory. All addresses and data values are shown in hex. The current value in the SP register is 0xFBC and points to the location shown in the diagram.

(C) (4 points) Is the argument to the initial call to $f$ even or odd? Circle your choice:

- Even
- Odd
- Can’t tell

First recursive call has ra = 0x60, following ones have ra = 0x80, both jal instructions are 16 = 0x20 bytes away → first jal (for odd argument) is at address 0x5C, second jal (for even argument) is at 0x7C, and first recursive call takes the jal for odd (ra = 0x60).

(D) (3 points) What value was in SP just prior to the initial call to $f$?

Initial contents of SP: 0x FCC

(E) (3 points) What is the address of the ‘jal ra, $f’ instruction that made the initial call to $f$?

Saved ra of initial call is 0x100 → jal ra, $f$ is at 0x100 − 4 = 0xFC

Address of instruction that made initial call to $f$: 0x FC

(F) (3 points) What is the hex address of the instruction at label EVEN?

Address of instruction at label EVEN: 0x 70
Problem 4. 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?

Processor A                                 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 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 }

6.004 Fall 2018 - 7 of 12 - Quiz #2
(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>
<tr>
<td>OP</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPIMM</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRANCH</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUI</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>STORE</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 mkRF1e1R1W 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
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
Problem 5. 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 $Data_x$ 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.

<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>0x12</td>
<td>0x0A</td>
<td>0x1A</td>
<td>0x2A</td>
<td>0x3A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x12</td>
<td>0x4B</td>
<td>0x5B</td>
<td>0x6B</td>
<td>0x7B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x12</td>
<td>0x3C</td>
<td>0x2C</td>
<td>0x1C</td>
<td>0x0C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x12</td>
<td>0x7D</td>
<td>0x6D</td>
<td>0x5D</td>
<td>0x4D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x67</td>
<td>0x33</td>
<td>0x23</td>
<td>0x13</td>
<td>0x03</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x67</td>
<td>0x44</td>
<td>0x34</td>
<td>0x24</td>
<td>0x14</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0x34</td>
<td>0x55</td>
<td>0x65</td>
<td>0x75</td>
<td>0x85</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x58</td>
<td>0x66</td>
<td>0x76</td>
<td>0x86</td>
<td>0x96</td>
</tr>
</tbody>
</table>

<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>1</td>
<td>0</td>
<td>0x27</td>
<td>0x80</td>
<td>0x81</td>
<td>0x82</td>
<td>0x83</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x27</td>
<td>0xB4</td>
<td>0xB5</td>
<td>0xB6</td>
<td>0xB7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x90</td>
<td>0xC3</td>
<td>0xC2</td>
<td>0xC1</td>
<td>0xC0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x90</td>
<td>0xD3</td>
<td>0xD4</td>
<td>0xD5</td>
<td>0xD6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x11</td>
<td>0x89</td>
<td>0x88</td>
<td>0x87</td>
<td>0x86</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0xA0</td>
<td>0x92</td>
<td>0x93</td>
<td>0x94</td>
<td>0x95</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x37</td>
<td>0xF5</td>
<td>0xF6</td>
<td>0xF7</td>
<td>0xF8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x21</td>
<td>0xA7</td>
<td>0xA8</td>
<td>0xA9</td>
<td>0xAA</td>
</tr>
</tbody>
</table>

(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: $A[\underline{3}\_ : \underline{0}]$

address bits used for cache index: $A[\underline{6}\_ : \underline{4}]$

address bits used for tag field: $A[\underline{31}\_ : \underline{7}]$

(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>set __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.

\[
. = 0x400
\]

\[
// \text{Code starts at address } 0x400
// \text{Assume } x1 \text{ is initialized to 0, } x2 \text{ is initialized to 1000 (number of array elements), and } x3 \text{ is initialized to the base address of array } A.
\]

\[
\text{loop: addi } x2, x2, -1
\]

\[
\text{slli } x4, x2, 2
\]

\[
\text{add } x4, x3, x4
\]

\[
\text{lw } x5, 0(x4)
\]

\[
\text{add } x1, x1, x5
\]

\[
\text{bnez } x2, \text{loop}
\]

Instruction miss rate = 0

Data miss rate = \( \frac{1}{4} \) (one miss every 4 iterations)

Every 4 iterations: 24/24 instruction hits + \( \frac{3}{4} \) data hits \( \rightarrow \) Hit rate is 27/28

Steady-state hit ratio: \_\_27/28\_\_

END OF QUIZ 2!