Quiz #2

<table>
<thead>
<tr>
<th>Name</th>
<th>Solutions</th>
<th>Athena login name</th>
<th>Score</th>
</tr>
</thead>
</table>

Recitation section
- WF 11, 35-308 (Silvina)
- WF 1, 38-166 (Andy)
- None (pick up quiz in 32-G846)

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.

Problem 1. Assembly Language (15 points)

For the RISC-V instruction sequences below, provide the hex values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the unimp instruction. Also assume that all registers are initialized to 0 before execution of each sequence begins.

(A) (7 points)

```
. = @0x0
    li x11, 0x400
    lw x11, 0x0(x11)
    bge x11, x0, L1
    xor x12, x11, 0xffff
    j end

L1: srli x12, x11, 8
    end: unimp

. = @0x400
X: .word 0xC0C0A0A0
```

Value left in x11: 0x__C0C0A0A0____

Value left in x12: 0x__3F3F5F5F____
(B) (8 points)

```
   . = 0x0
   lw x11, 0x200(x0)
   mv x12, x0
   loop:
      andi x13, x11, 1
      add x12, x12, x13
      srl x11, x11, 1
      bnez x11, loop
      unimp
   . = 0x200
   X: .word 0x00140083

   Value left in x11: 0x__0__________
   Value left in x12: 0x__5__________
   Value left in x13: 0x__1__________
```
Problem 2. RISC-V Assembly and Calling Conventions (20 points)

For each of the code segments below, specify whether or not the RISC-V assembly code properly implements the desired functionality described in C and satisfies the RISC-V calling convention. If either the functionality or the calling convention are not satisfied, then provide code that is functionally correct and satisfies the calling convention.

Note that ‘g’ refers to another function whose specification is not provided to you. Also, note that there may be multiple errors in the code. You should correct them all.

(A) (7 points)

```c
int f(int a, int b) {
    return g(a + b, b) + a;
}
```

```assembly
f:
    mv s1, a0
    add a0, a1, a0
    jal ra, g
    add a0, a0, s1
    ret

g:
```

Assembly code produces expected results and follows calling convention:

YES  NO

If not, provide correct code here:

```assembly
f:
    addi sp, sp, -8
    sw s1, 0(sp)
    sw ra, 4(sp)
    mv s1, a0
    add a0, a1, a0
    jal ra, g
    add a0, a0, s1
    lw s1, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret

g:
```
(B) (6 points)

```c
int f(int a, int b, bool c) {
    if (c) return a * 2;
    else return a - b;
}
f: bnez a2, L1
    slli a0, a0, 2
    ret
L1: sub a0, a1, a0
    ret
```

<table>
<thead>
<tr>
<th>Assembly code produces expected results and follows calling convention:</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>If not, provide correct code here:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| f: beqz a2, L1
    slli a0, a0, 1
    ret |
| L1: sub a0, a0, a1 |

(C) (7 points)

```c
unsigned int f(unsigned int a, int b) {
    if (b < 0) {
        return g(a) + b;
    }
    else return a >> b;
}
f: slt t0, a1, x0
    beqz t0, L1
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, g
    add a0, a0, a1
    ret
L1: sra a0, a0, a1
    ret
g:
```

<table>
<thead>
<tr>
<th>Assembly code produces expected results and follows calling convention:</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>If not, provide correct code here:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| f: slt t0, a1, x0
    beqz t0, L1
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a1, 4(sp)
    jal ra, g
    lw ra, 0(sp)
    lw a1, 4(sp)
    addi sp, sp, 8
    add a0, a0, a1
    ret |
| L1: srl a0, a0, a1 |
| g: |
Problem 3. Procedures and Stacks (20 points)

You are given an incomplete listing of a C procedure and its translation to RISC-V assembly code (shown on the right):

```c
int f(int a, int b) {
    if (a < b)
        return ???;
    else
        return a | b;
}
```

(A) (3 points) Give the HEX encoding of the ‘sw ra, 8(sp)’ instruction.

Hex encoding of sw ra, 8(sp): 0x____112423_____

(B) (3 points) What is the missing C expression corresponding to the ‘???’ in the above program?

```
f(a+1, b-1) | (a ^ b)
```

(give C code expression)

(C) (2 point) How many words are stored on the stack every time we want to execute function f recursively?

```
Number of words pushed onto the stack for each recursive call to f? ______3____
```
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 \( f \), \textit{just prior} to the execution of the \texttt{addi sp, sp, 12} instruction at label L2. The diagram on the right shows the contents of a region of memory. All addresses and data values are shown in hex. \textbf{The current value in the SP register is 0xF40 and points to the location shown in the diagram.}

(D) (3 points) What are the arguments to the \textit{original} call to \( f \)? Write CAN’T TELL if you can’t tell.

\[
\text{Original arguments to } f, \quad a = \underline{3} \quad ; \quad b = \underline{7} \\
\]

(E) (3 points) What are the arguments to the \textit{current} call to \( f \)? Write CAN’T TELL if you can’t tell.

\[
\text{Current arguments to } f, \quad a = \underline{4} \quad ; \quad b = \underline{6} \\
\]

(F) (2 points) What value was in SP just prior to the initial call to \( f \)?

\[
\text{Initial contents of SP: 0xF58} \\
\]

(G) (2 points) What is the address of the ‘jal ra, f’ instruction that made the original call to \( f \)?

\[
\text{Address of original call to } f: \quad 0xA04 \\
\]

(H) (2 points) What is the hex address of the instruction at L1?

\[
\text{Address of instruction at L1: 0x118} \\
\]
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.

```
add a4, a2, a0
add a5, a3, a1
sltu t0, a4, a2
add a5, a5, t0
```

(B) (3 points) Does your code also work on signed integers in two’s complement encoding? Briefly explain why or why not.

Yes, because addition works in the same way whether numbers are unsigned or in two’s complement representation (note the comparison to find the carry bit must be unsigned in both cases).
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.

   ```
   addc a4, a2, a0
   addc a5, a3, a1
   ```

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.

```plaintext
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
        OPI
        IMM: 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.

```plaintext
ADDC: begin
    Bit#(33) sum = {1'b0, rVal1} + {1'b0, rVal2} + carry;
    data = sum[31:0];
    newCarry = sum[32];
    nextPc = pc + 4;
end
```
Problem 5. 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.

(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 0x3C all be resident in **Cache2** at the same time? **Explain why you chose your answer.**

0x00: index 0; 0x10: index 2; 0x18: index 3; 0x3C: index 3  
0x18 and 0x3C 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.

```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
```

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!