6.004 Quiz 2 Review Problems

RISC-V Assembly instructions and their encoding:

<table>
<thead>
<tr>
<th>RV32I Instruction Set (MIT 6.8084 subset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>01000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>01000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>01000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
</tbody>
</table>
Problem 1. RISC-V Assembly

A) What is the binary encoding of the following RISC-V instruction?

```
sub x1, x2, x3
```

```
func7: 0100000
rd: x1: 00001
rs1: x2: 00010
rs2: x3: 00011
func3: 000
opcode: 0110011
```

binary instruction: func7 rs2 rs1 func3 rd opcode
binary instruction: 0100000 00011 00010 000 00001 0110011
rewritten binary instruction: 0100 0000 0011 0001 0000 0000 1011 0011
hex instruction: 0x403100B3

B) For each of the following sequences of instructions, assume all registers are initially 0. Specify the value of each used register after executing the sequence of instructions. Specify the address corresponding to all labels. Also, specify the address and contents of any modified memory locations.

A)
```
li x0, 3
li x1, 5
lui x2, 6
```

x0 = 0
x1 = 5
x2 = 0x6000

B)
```
li x4, 7
addi x5, x4, -2
xor x4, x5, x4
```

x4 = 0b010
x5 = 5

C)
```
. = 0
addi x5, x5, 2
blt x4, x5, L1
addi x4, x4, 5
L1: addi x4, x4, -3
```

x4 = -3
x5 = 2
L1 = 0xC
D)

\[
. = 0
\]

\[
\text{li } x2, 0x100
\]

\[
\text{addi } x6, x2, 4
\]

\[
\text{lw } x3, 8(x2)
\]

\[
\text{li } x4, 0x114
\]

\[
\text{mv } x5, x4
\]

\[
\text{sw } x4, 0xC(x6)
\]

\[. = 0x100\]

1
2
3
4
5
6
7

\[
x2 = 0x100
\]

\[
x3 = 3
\]

\[
x4 = 0x114
\]

\[
x5 = 0x114
\]

\[
x6 = 0x104
\]

addr 0x110 previously held 5, now it holds 0x114.
**RISC-V Calling Conventions: Some useful facts to know**

Arguments are passed in a0, a1, etc. with the first argument being in a0, the second in a1 and so on.

Return values are returned in a0 (and a1 if you have two return values).

The return address is stored in the ra register.

A and T registers can be used freely provided you don’t need the original value in them. If you do, must save them to the stack before making another procedure call.

Ra must be saved onto the stack if you are making nested procedure calls otherwise your original return address will be overwritten.

S registers and the SP must be returned in exactly the same state as they were when a function was called. This means all S registers must be saved onto the stack before modifying them. Prior to returning from a procedure, anything that was pushed onto the stack must be taken off the stack so that SP is restored to its original value.

To push N values onto the stack:

```assembly
addi sp, sp, -X       // X = N*4
sw x1, 0(sp)          // value 1
sw x2, 4(sp)          // value 2
sw x3, 8(sp)          // ...
sw x4, 0xC(sp)        // value N
```

To pop N values from the stack without restoring the values:

```assembly
addi sp, sp, X
```

To pop N values from the stack and restore the values:

```assembly
lw x1, 0(sp)          
lw x2, 4(sp)          
lw x3, 8(sp)          
lw x4, 0xC(sp)        
addi sp, sp, X
```

To make a procedure call you must save the return address in ra. This can be achieved using either:

```assembly
jal ra, label
```

or

```assembly
jal label           (this is a pseudoinstruction that gets translated into jal ra,label).
```

To return from a procedure call you must set the pc to the contents of ra. This can be achieved using either:

```assembly
ret     or    jr ra  or    jalr ra
```
Problem 2. RISC-V Calling Conventions and Stacks

For each implementation of the function funcA defined below, identify and correct any errors that result in incorrect behavior of the function as well as any errors in the use of the RISC-V calling conventions. Assume that funcA is called from code that follows the RISC-V calling conventions. Also, assume that funcA is originally called with x > 0.

FuncA is defined as:

```c
int funcA(int a, int b, int x) {
    if (x == 0) return b;
    else return x + funcA(a+b, b+2, x-1)
}
```

Implementation 1:

```asm
funcA:
    beqz a2, done
    addi sp, sp, -4
    sw ra, 0(sp)
    add a0, a0, a1
    addi a1, a1, 2
    addi a2, a2, -1
    jal ra, funcA
    add a0, a0, a2
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

done: mv a0, a1 // return b
    ret

Can’t assume that a2 was saved. Need to save it onto the stack before nested procedures call and then restore it before add a0, a0, a2.

Implementation 2:

```asm
funcA:
    beqz a2, done
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a2, 4(sp)
    add a0, a0, a1
    addi a1, a1, 2
    addi a2, a2, -1
    jal ra, funcA
    add a0, a0, a2
    lw ra, 0(sp)
    lw a2, 4(sp)
    addi sp, sp, 8
    ret
```
done: mv a0, a1 // return b
ret

a2 needs to be restored before add a0, a0, a2 instruction

Implementation 3:

funcA:
   beqz a2, done
   addi t1, a1, 2
   add t0, a0, a1 // t0 = a0 + a1
   addi s2, a2, -1
   mv t2, a2
   mv a0, t0
   mv a1, t1
   mv a2, s2
   jal ra, funcA
   add a0, a0, t2
   ret

done:
   mv a0, a1 // return b
   ret

Need to save s2 and ra onto stack before using it:
addi sp, sp, -8
sw s2, 0(sp)
sw ra, 4(sp)

After returning from recursion, restore s2, ra and stack
lw s2, 0(sp)
lw ra, 4(sp)
addi sp, sp, 8

Also, can’t assume that t2 was is correct after returning from nested call unless we first save it onto the stack, and then restore it. Alternatively, the value that was saved in t2 could be saved in register s3 but then we would need to save s3 and restore s3 before and after.
Problem 3:
Stack Detective:

Assume that the code for the previous problem was written as shown below:

```c
int funcA(int a, int b, int x) {
    if (x == 0) return b;
    else return x + funcA(a+b, b+2, x-1)
}
```

```assembly
.beqz a2, done
addi sp, sp, -8
sw ra, 0(sp)
sw a2, 4(sp)
add a0, a0, a1
addi a1, a1, 2
addi a2, a2, -1
jal ra, funcA
lw a2, 4(sp)
.L1:
add a0, a0, a2
lw ra, 0(sp)
addi sp, sp, 8
ret

done:  mv a0, a1  // return b
ret
```

Assume execution was halted just as you are about to execute the instruction at label L1.

You are given the stack on the right at the time execution was halted.

A) What is the value of the empty cell in the stack trace above?

1 because previous saved value of x (a2) was 2.

B) | Original x | Original ra |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 0x1B0</td>
<td>3</td>
</tr>
<tr>
<td>A = 0x228</td>
<td>2</td>
</tr>
</tbody>
</table>
Caches:

Problem 4. Cache Hit Rates

The following questions ask you to evaluate alternative cache designs using a specific pattern of memory references. Each of the caches under consideration has a total capacity of 8 (4-byte) words, with one word stored in each cache line. The cache designs under consideration are:

DM: a direct-mapped cache.

S2: a 2-way set-associative cache with a least-recently-used (LRU) replacement policy.

The questions below present a repeating sequence of addresses (expressed as decimal numbers) for memory reads. After the access to the last address on the list, the sequence starts over with the first address on the list. Keep in mind that byte addressing is used; addresses of consecutive words in memory differ by 4. Each question asks for the hit ratio for each cache. Answer by considering the steady-state hit ratio, i.e., the percentage of memory references that hit in the cache after the sequence has been repeated many times. Your answer can be expressed as a percentage or a ratio.

<table>
<thead>
<tr>
<th>Repeating address sequence</th>
<th>Hit ratio for cache DM</th>
<th>Hit ratio for cache S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) 0, 8, 16, 24, 32</td>
<td>3/5</td>
<td>2/5</td>
</tr>
<tr>
<td>(B) 0, 4, 8, 12, 16, 20, 24, 28, 32</td>
<td>7/9</td>
<td>6/9</td>
</tr>
<tr>
<td>(C) 0, 4, 8, 12, 32, 36, 40, 44</td>
<td>0/8</td>
<td>8/8</td>
</tr>
</tbody>
</table>

Determine access line numbers for DM (word addr modulo 8)

(A) 0, 2, 4, 6, 0       (B) 0, 1, 2, 3, 0, 1, 2, 3  (C) 0, 1, 2, 3, 0, 1, 2, 3
2 misses/repeat         2 misses/repeat         8 misses/repeat

Determine access line numbers for S2 (word addr modulo 4)

(A) 0, 2, 0, 0       (B) 0, 1, 2, 3, 0, 1, 2, 3  (C) 0, 1, 2, 3, 0, 1, 2, 3
3 misses/repeat         3 misses/repeat         0 misses/repeat
Problem 5. Memory Hierarchy and Cache Accesses (8 points)

(A) (1 point)

Assuming the memory hierarchy configuration shown above which consists of a level 1 (L1) cache with access time of 4 ns and hit rate of 0.6, and a level 2 (L2) cache with access time of 8 ns and hit rate of 0.9, and a main memory whose access time is 100 ns, determine the AMAT (average memory access time) for this system.

\[
\text{AMAT (ns): } 11.2
\]

(B) (1 point) True or False. A 4 line 2-way set associative cache always has a better hit rate than a direct mapped cache with 8 lines.

TRUE \hspace{1cm} FALSE

For parts C-G, use the 2-line 2-way set associative cache with a block size of 2, shown below.

(C) (1 point) Which address bits are used for the offset, cache line index, and tag comparison?

Address bits used as offset (including byte offset): A[ 2 : 0 ]

Address bits used as cache line index: A[ 3 : 3 ]

Address bits used for tag comparison: A[ 31 : 4 ]

(D) (1 point) Give the address corresponding to the data 0x23 (line 0, data word B1) or write CAN’T TELL if there’s insufficient information to determine its address.

CAN’T TELL of address of 0x23 data (0x): 0x524

(E) (1 point) Would a store to address 0x71C hit or miss?

HIT \hspace{1cm} MISS
(F) (1 point) Suppose that the CPU executes a `sw` instruction writes the value 7 into location 0x71C. Fill in all *changed* entries for the cache in the unfilled cache below.

```
<table>
<thead>
<tr>
<th>line</th>
<th>D</th>
<th>V</th>
<th>tag</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
</tbody>
</table>
```

It’s a store so set D bit, then update contents of A1 in line 1.

For parts G and H compute the line number and tag for each data access. Indicate if the access will result in a hit or a miss. Provide the value returned by the `lw`, or enter CAN’T TELL if you can’t tell from the information provided. (1 point each)

<table>
<thead>
<tr>
<th>Access</th>
<th>Line number</th>
<th>Tag</th>
<th>Hit? (y or n)</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>(G) lw from address 0x528</td>
<td>1</td>
<td>0x52</td>
<td>N</td>
<td>Can’t tell (note V=0)</td>
</tr>
<tr>
<td>(H) lw from address 0x430</td>
<td>0</td>
<td>0x43</td>
<td>Y</td>
<td>0x67</td>
</tr>
</tbody>
</table>
Problem 6: Hardware Implementation

We want to add the following instruction to the RISC-V ISA:

\texttt{lwri rd, rs1, rs2}

The behavior of this new \texttt{lwri} instruction is that it adds the contents of source registers \texttt{rs1} and \texttt{rs2} and uses the result as the address for your load. In other words:

\[
\text{reg}[\text{rd}] \leftarrow \text{mem}[\text{reg}[\text{rs1}] + 4\times\text{reg}[\text{rs2}]] \\
\text{pc} \leftarrow \text{pc} + 4
\]

Suppose you were given a decoder that was fully functioning prior to the addition of this new instruction, and you were told that a new type \texttt{IType LWRI} has been added for this instruction. Fill in the required changes to the Decode and Execute functions in order to support adding this instruction. We have included relevant portions of these functions below to simplify this task. Assume that the rest of the hardware implementation is the same as what you have seen in lecture and lab.

(A) (12 points) Fill in the required additions to the code below.

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

function DecodedInst decode(Bit#(32) inst);
...
Bit#(7) opLWRI = 7'b0001011; // or other available opcode

DecodedInst dInst = ?;
  dInst.iType = Unsupported;
  case(opcode)
  ...
  opLoad: if (funct3 == fnLW) begin
  end
  opLWRI: dInst = DecodedInst { rd: tagged Valid rd, rs1: rs1, rs2: rs2, imm: ?, brFunc: ?, aluFunc: ?, iType: LWRI };
  endcase
  return dInst;
endfunction
```

*Write your code below. Please add arrows to indicate where your code fits in the decode function.*
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2, 
        Word pc);
        
        let imm = dInst.imm;
        let brFunc = dInst.brFunc;
        let aluFunc = dInst.aluFunc;
        Word data = ?;
        Word nextPc = ?;
        Word addr = ?;
        case (dInst.iType)
                OP: begin data = alu(rVal1, rVal2, aluFunc); nextPc = pc+4; end
                OPIMM: begin data = alu(rVal1, imm, aluFunc); nextPc = pc+4; end
                BRANCH: begin nextPc=aluBr(rVal1,rVal2,brFunc)? pc+imm : pc+4; end
                LUI: begin data = imm; nextPc = pc+4; end
                JAL: begin data = pc+4; nextPc = pc+imm; end
                JALR: begin data = pc+4; nextPc = (rVal1+imm) & ~1; end
                LOAD: begin addr = rVal1+imm; nextPc = pc+4; end
                STORE: begin data = rVal2; addr = rVal1+imm; nextPc = pc+4; end
                AUIPC: begin data = pc+imm; nextPc = pc+4; end
                LWRI: begin addr = rVal1+(rVal2<<2); nextPc = pc+4; end
        endcase
        ExecInst eInst = ?;
        eInst.iType = dInst.iType;
        eInst.rd = dInst.rd;
        eInst.data = data;
        eInst.addr = addr;
        eInst.nextPc = nextPc;
        return eInst;
        endfunction

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

(B) (3 points) Could a store operation that calculated the address in the same manner as LWRI be added without major changes to the RISC-V ISA?

Yes  No

(C) (3 points) If you answered Yes to B then provide a brief summary of the required changes. If you answered No to B then explain why not.

A store instruction already makes use of rs1, and rs2 because it needs rs2 to hold the data to be written to memory. So this kind of address calculation would not be possible for a store operation.

(D) Do we need to add an extra stage in our multi-cycle implementation to implement LWRI?

No