EECS 151/251A Homework 4

Due Friday, Oct 2nd, 2020

Midterm Practice [1 pt]

Before you start the rest of this homework assignment, please practice the mechanics of the midterm exam. Everything must be uploaded by Friday, October 2 (same time as this HW).

1. Make sure you read through the Exam Policy. Set up your workspace and Zoom to conform to the rules in the policy.
2. Submit your Zoom meeting ID that you will use for the exam to this form.
3. For this mock exam, you will just solve the practice problems on the next page only. Set up your recording and workspace according to the policy, solve the problem as you would on the actual exam (give yourself 5 minutes), then end the recording.
4. Upload your completed problems to Gradescope, under the assignment "Practice Exam". This will not be graded for correctness, just for completion.
5. Submit your recording link to this form. GSIs will contact you before the midterm if there are any issues with your recording.
6. Done! Now, proceed with the HW.

Version: 1 - 2020-10-06 08:04:35Z

Your Name (first last) SID

Practice Problems

1. Fill in the 5 stages of the RISC-V datapath (abbreviations are fine):

   Stage 1: _____

   Stage 2: _____

   Stage 3: _____

   Stage 4: _____

   Stage 5: _____

2. A key part of the RISC-V base (32-bit) datapath is the register file, which provides the source operands and destination for most computations.

   (a) Draw the conceptual diagram of the register file, labeling the input and output signals with their bitwidths.

   (b) What is the value in register `x0`? _____

   (c) During which operation is the input clock actually used: read or write? _____

Reading

In addition to reviewing the RISC-V ISA and datapath lectures, skim through the RISC-V ISA spec. In particular, focus on the Introduction, Chapter 2 (RV32I Base Integer Instruction Set), and the tables on pages 129 and 130.

Problem 1: RISC-V Manual Assembly [4 pts (1 each)]

Manually construct the binary instruction for the following assembly instructions. Submit all of the following for each instruction:

- The 32-bit binary number for the instruction
- The core instruction format it belongs to
- Delineate the 32 bits into the subfields of the instruction format and label each field with the opcode/registers/immediate/offset etc. specified by the instruction

Note: we highly encourage you to do this by hand from the ISA spec, but it is possible to assemble them using RISC-V GCC or venus.

(a) `sra x1, x2, x3`

(b) `andi x1, x2, 100`

(c) `sh x1, 4(x2)`

(d) `bne x6, x8, 1024`

Solution:

You should have manually assembled these by hand for this homework, but in the future you can use the RISC-V GCC to do it for you. The compiler is located on the instructional servers at `/home/ff/eecs151/tools-151/riscv-toolchain-fa19/bin`, which may already be on your `PATH`.

Create a file `test.S` with contents:

```
.global _start
.section .text
_start:
  sra x1, x2, x3
  andi x1, x2, 100
  sh x4, 4(x2)
  bne x6, x8, 1024
```

Run the compiler:

```
riscv64-unknown-elf-gcc -c -mabi=ilp32 -march=rv32i -static -mcmodel=medany \
  -fvisibility=hidden -nostdlib -nostartfiles test.S -o test.o
```

Dump the generated assembly:

```
riscv64-unknown-elf-objdump -Mnumeric -D test.o

test.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <_start>:
   0: 403150b3          sra  x1,x2,x3
   4: 06417093          andi x1,x2,100
   8: 00411223          sh   x4,4(x2)
   c: 00830463          beq  x6,x8,14 <_start+0x14>
  10: 0000006f          j    10 <_start+0x10>
```

The answers for the first 3 parts are:

a) `sra` is an R-type instruction.
`0x403150b3 = 0b0100_0000_0011_0001_0101_0000_1011_0011`
Delineated: `0100000_00011_00010_101_00001_0110011`
`0110011` in the `opcode` field denotes `OP`
`SRA = 101` in the `funct3` field ([14:12])
Register `x1` is the `rd` field ([11:7])
Register `x2` is the `rs1` field ([19:15])
Register `x3` is the `rs2` field ([24:20])

b) `andi` is an I-type instruction.
`0x06417093 = 0b0000_0110_0100_0001_0111_0000_1001_0011`
Delineated: `000001100100_00010_111_00001_0010011`
`0010011` in the `opcode` field denotes `OP-IMM`
`andi = 111` in the `funct3` field ([14:12])
Register `x1` is the `rd` field ([11:7])
Register `x2` is the `rs1` field ([19:15])
Immediate `100` is the `imm` field ([31:20])

c) `sh` is an S-type instruction.
`0x00411223 = 0b0000_0000_0100_0001_0001_0010_0010_0011`
Delineated: `0000000_00100_00010_001_00100_0100011`
`0100011` in the `opcode` field denotes `STORE`
`sh = 001` in the `funct3` field ([14:12]), denoting 16b stores
Register `x4` is the `rs2` field ([24:20])

Register `x2` is the `rs1` field ([19:15])
Offset `4` is the `imm` field ([11:7])

For the last part, the compiler created a `beq` instruction with an offset pointing to the line right after the end of the program instead of 1024. The correct solution can be found by assembling the instruction with venus:

`bne` is a B-type instruction.
`0x40831063 = 0b0100_0000_1000_0011_0001_0000_0110_0011`
Delineated: `0_100000_01000_00110_001_0000_0_1100011`
`1100011` in the `opcode` field denotes `BRANCH`
`bne = 001` in the `funct3` field ([14:12])
Register `x6` is the `rs1` field ([19:15])
Register `x8` is the `rs2` field ([24:20])
Offset `1024` is the `imm` field ([31:25], [11:6])

Problem 2: RISC-V Assembly Programs [4 pts (1 each)]

Write down the values of the specified registers after the following programs have run. Show your work by annotating the what happens/changes after each instruction. Note that some instructions are pseudo-instructions, such as `li` for load immediate. Refer to Table 25.2 in the RISC-V spec for a list of pseudo-instructions and their base implementations.

(a)
```
    li x0, 100
    li x1, 200
    sub x2, x1, x0
    addi x2, x2, 100
```

x0 = _____, x1 = _____, x2 = _____

Solution:

x0 = 0, x1 = 200, x3 = 300
Instruction-by-instruction:
x0 = 0 | x1 = 200 | x2 = 200 | x2 = 300

(b)
```
    li x1, 0xdead
    li x2, 0xbeef
    li x3, 0x128
    sb x2, 0(x3)
    sra x2, x2, x3
    sb x2, 1(x3)
    sb x1, 2(x3)
    sra x1, x1, x3
    sb x1, 3(x3)
    lw x4, 0(x3)
```

x1 = _____, x2 = _____, x4 = _____

Solution:

x1 = 0xde, x2 = 0xbe, x4 = 0xdeadbeef
Instruction-by-instruction:
x1 = 0xdead | x2 = 0xbeef | x3 = 0x128 | dmem @ 0x128 = 0xef | x2 = 0xbe | dmem @ 0x129 = 0xbe | dmem @ 0x12A = 0xad | x1 = 0xde | dmem @ 0x12B = 0xde | x4 = 0xdeadbeef

(c)
```
    li x1 -1
    li x2 1
f1: sll x3 x1 x2
    sub x1 x1 x3
    addi x2 x2 1
    blt x1 x2 f1
```

x1 = _____, x2 = _____, x3 = _____

Solution:

x1 = 21, x2 = 4, x3 = 0xFFFFFFE8 (-24)
Instruction-by-instruction:
x1 = -1 | x2 = 1 | x3 = 0xFFFFFFFE (-2) | x1 = 1 | x2 = 2 | branch to f1 | x3 = 4 | x1 = -3 | x2 = 3 | branch to f1 | x3 = 0xFFFFFFE8 (-24) | x1 = 21 | x2 = 4 | branch not taken

(d) Assume the following instructions start at address 0x0.

```
      li x1, 0
      jalr x2, x0, 12
      addi x1, x1, 100
      jalr x2, x2, 0
      addi x1, x1, 100
      jalr x2, x2, 0
      j more
more: jalr x2, x2, 0
      j end
end:  nop
```

x1 = _____, x2 = _____

Solution:

x1 = 300, x2 = 32
Instruction-by-instruction:
x1 = 0 | x2 = 8 | x2 = 16 | x1 = 100 | x2 = 16 | x1 = 200 | x2 = 24 | x1 = 300 | x2 = 28 | x2 = 32 | jump to end | nop

Problem 3: RV64M Multiplication ALU [5 pts (1 per inst.)]

Refer to Chapter 7 ("M" Standard Extension for Integer Multiplication and Division, Version 2.0) in the RISC-V spec and the corresponding instruction set listing on page 131. Using the template given here, implement an ALU that supports the subset of the **64-bit "M" extension** that performs multiplication. Don't worry about any synthesizability/performance issues of your implementation in this exercise.

Pay attention to which bits are important in each variant of the multiply, as well as the signed-ness of the inputs to the multiply (*) operator. You may want to make sure you are comfortable with Verilog signed arithmetic and the concatenation operator first.

```
`define ALU_MUL 0
`define ALU_MULH 1
`define ALU_MULHSU 2
`define ALU_MULHU 3
`define ALU_MULW 4
module rv64mult_alu(
  input [63:0] a,
  input [63:0] b,
  input [2:0] op, // op is one of the values `define'd above
  output [63:0] c,
);

  // Your implementation

endmodule
```

Solution:

In this solution, the behavior is `MUL` as the default if op is an undefined value. Other implementations of the default case are allowed. Notes:

- `MUL` is straightfoward: the * operator will truncate MSBs of the result based on the width of the result wire/reg.
- `MULH` requires sign extension of both inputs to a total of 128 bits, followed by a regular multiply and then discarding of the lower 64 bits. Another correct answer would be a `c_all =` $signed(a)$`*signed(b)` where `c_all` is a 128 bit reg, followed by a `c = c_all >> 64`.
- `MULSU` requires sign extension of only input a but zero padding input b, and a similar multiplication/truncation.
- `MULU` requires zero padding of both inputs, and a similar multiplication/truncation.
- `MULW` is just a 32-bit multiplication of the lower 31 bits of a and b, followed by a sign extension to 64 bits.

- Alternate solutions may do manual sign extension by 1 bit, assigning the multiplication to a 128-bit intermediate reg, and then doing the 64 bit shift or bit slicing to get c.

```
module rv64mult_alu(
  input [63:0] a,
  input [63:0] b,
  input [2:0] op,
  output [63:0] c,
);

  always @(*) begin
    case (op)
      `ALU_MULH: c = (($signed({{64{a[63]}}, a}) * $signed({{64{b[63]}}, b})) >>> 64);
      `ALU_MULHSU: c = (($signed({{64{a[63]}}, a}) * {64'b0, b}) >> 64);
      `ALU_MULHU: c = (({64'b0, a} * {64'b0, b}) >> 64);
      `ALU_MULW: c = $signed(a[31:0] * b[31:0]);
      default: c = a * b;
    endcase
  end
endmodule
```

Problem 4: Extending RV32I Branching for ReLU [10 pts]

The Rectified Linear Units, or ReLU function is very useful in machine learning. It is the most popular activation function and is defined as $y = max(0, x)$. Graphically, it looks like this:

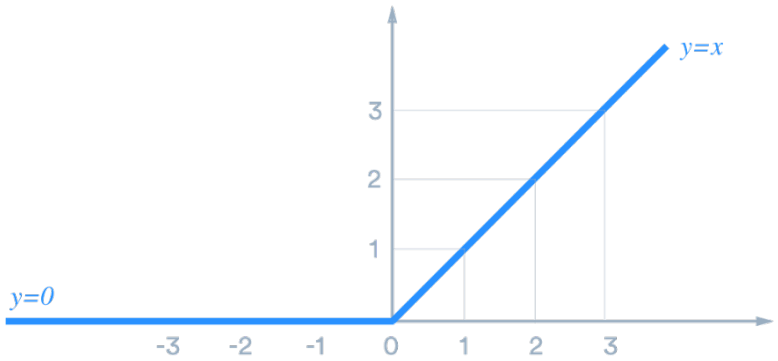

Figure 1: ReLU activation function

In RV32I, this function can be implemented with a couple instructions, using a branch. Let's try to extend RV32I to do a ReLU in a single instruction by using what exists in the datapath.

(a) (1 pt) Fill in the assembly below needed to do a ReLU in RV32I, using a `bge` instruction. Assume register `x1` already stores x and we want register `x2` to get y.

```
      mv
      bge
      mv
end:  nop
```

(b) (4 pts) To guide you toward a solution, first complete the Verilog implementation of `ALUSel` in the Control Logic for RV32I. For this part, write down the logical expressions for the output control signals in terms of an instruction's `opcode`, `funct3`, and `funct7` bits, as well as the `BrEq` and `BrLT` signals from the branch comparator. You can also simplify the expressions by comparing the `opcode` field against the constants in Table 24.1 in the RISC-V ISA manual. For example: `assign sig = (opcode == OP-32) || (opcode == LOAD);`.

```
// The rest of the Control Logic module is not implemented here.
// ALUSel here is a 4-bit output, and should take the constants
// defined below for each type of operation the ALU is to do.
parameter ALU_ADD = 4'b0000;
parameter ALU_SUB = 4'b0001;
parameter ALU_SLT = 4'b0010;
parameter ALU_SLTU = 4'b0011;
parameter ALU_AND = 4'b0100;
parameter ALU_OR = 4'b0101;
parameter ALU_XOR = 4'b0110;
parameter ALU_SLL = 4'b0111;
parameter ALU_SRL = 4'b1000;
```

```
parameter ALU_SRA = 4'b1001;

always @(*) begin
  ALUSel = ALU_ADD; // default

  // Complete your implementation here

end
```

(c) (2 pts) We now want to implement ReLU as an R-type instruction. For this problem, the instruction format would be like `relu rd rs1 rs2` such that:

- We must specify register `x0` for `rs2` so as to check against the value 0 for ReLU. Any other value for `rs2` would give us an invalid result.
- `rs1` should be the value of x and `rd` would be the value of y.

In essence, we want the branch comparator to help us with the ReLU function. Submit the following:

(i) Refer to table 24.1 and the following description in the ISA spec. How can we extend our ISA to enable adding our ReLU instruction?

(ii) With the scheme, we do not need to update the datapath. Instead, what do we need to change?

(d) (2 pts) Now, update your Verilog code from b) to enable your new ReLU instruction.

(e) (1 pt) Could we also implement our new ReLU instruction as an I-type instruction instead? If so, how would its implementation be different from the R-type version?

Solution:

(a) Here is a solution assuming x is in the `x1` register and y is to be in the `x2` register.

```
      mv x2 x1
      bge x1 x0 8
      mv x2 x0
end: nop
```

An alternate correct solution would be to swap the 2 `mv` instructions and the registers for `bge`.

(b)
```
// omitting parameter declarations
always @(*) begin
  ALUSel = ALU_ADD; // default
  if (opcode == OP || opcode == OP-IMM) begin
    case (funct3)
      3'b000: ALUSel = funct7[5] ? ALU_SUB : ALU_ADD;
      3'b001: ALUSel = ALU_SLL;
      3'b010: ALUSel = ALU_SLT;
      3'b011: ALUSel = ALU_SLTU;
```

```
            3'b100: ALUSel = ALU_XOR;
            3'b101: ALUSel = funct7[5] ? ALU_SRA : ALU_SRL;
            3'b110: ALUSel = ALU_OR;
            3'b111: ALUSel = ALU_AND;
            default: ;
          endcase
        end
      end
```

(c) We should encode the new ReLU instruction within the opcode `custom-0` or `custom-1`, as these are reserved for user extensions within RV32I.
The control logic needs to be changed such that when `BrLT` from the branch comparator is 0, we take x directly (`x1 OR x0`), and when it is 1, we take 0 directly (`x1 AND x0`).
(Optional) It would be more robust to condition this on the branch comparator doing a signed comparison (`(BrLT  !BrUn)` instead of just `BrLT`) because an unsigned comparison makes no sense. This could also take the form of assuming that bit 6 of `funct7` is 0 like other signed branching instructions, since we are conditioning the new logic only on the new custom opcode.
Depending on implementation, some other control logic outputs (e.g. `WBSel`) may also need to be updated to account for the new opcode.

(d) This assumes ReLU is in a new opcode constant `CUSTOM-0`:

```
// skipped parameter declarations
always @(*) begin
  ALUSel = ALU_ADD; // default
  if (opcode == OP || opcode == OP-IMM) begin
    // this is all the same as above
  end
  // this is new
  else if (opcode == CUSTOM-0) begin
    ALUSel = BrLT ? ALU_AND : ALU_OR;
  end
end
```

(e) Yes. Instead of `rs2` being register `x0`, we would just need to put 0 as the immediate, set `BSel` just as we would with the other integer register-immediate instructions, and implement `ALUSel` the same way.

Problem 5: Extending RV32I for BNNs [10 pts]

We want to add another, more complex instruction to this ISA that may help us with certain neural network applications: a bitwise multiply-and-accumulate (MAC) operation. This operation can also be considered as a bitwise dot product in linear algebra speak. Here are the specifications:

- Each 32-bit datum in our architecture represents a 32-element long vector of binary values
- Bits translate to binary values as follows: a "0" bit represents the value "-1" and a "1" bit represents the value "+1"
- The MAC is therefore calculated as follows, shown for a 4-bit vector example for understanding.
  Given input binary vectors `x = {1,-1,1,-1} = 1010` and `y = {-1,1,1,-1} = 0110`:
  `mac(x, y) = (+1 * -1) + (-1 * +1) + (+1 * +1) + (-1 * -1) = 0`.

Background: Binary Neural Networks (BNNs) are neural networks with weight and activation matrices quantized to ± 1. This extreme degree of quantization is desired for applications where data movement must be kept to an absolute minimum because it compresses one dimension of the weight and activation matrices. Furthermore, BNNs promise to dramatically save computational resources, because a MAC can use bitwise computations rather than integer multiplication and addition.

Your task: Implement the algorithm that accomplishes this bitwise MAC, called XNOR-Popcount. In this algorithm, the bitwise XNOR is taken, followed by a popcount, and finally a scaling adjustment. In pseudocode:

```
a = xnor(x,y)
b = popcount(a)
c = len(a) // this is known (32)
mac(x,y) = 2 * b - c
```

The popcount operation here (otherwise known as Hamming weight) is also very useful for other applications such as communications and cryptography, but is missing from the RISC-V ISA spec. Due to popular demand, it is currently draft proposed in the "B" or bitmanip extension.

The popcount of a binary number is the sum of all the 1's in the bitstring. For example, the popcount of `10110100` is 4. The naïve method of implementing popcount using an integer instruction set would be check each bit one-by-one, incrementing a counter if we see a 1. In a C implementation for 32 bits, it would look like this:

```c
int pop(unsigned x) {
    int count = 0;
    for (i = 0; i < 32; i++) { // loop through all bits
        if ((x & 1) == 1)      // mask LSB and check for 1
```

```
            count++;
        x >>= 1;                // right shift by 1
    }
    return count;
}
```

Clearly, there are an extremely large number of RV32I instructions needed to implement this. There is a much faster algorithm using an integer instruction set, as described in Hacker's Delight:

```
int pop(unsigned x)
{
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x0000003F;
}
```

To accelerate the MAC operation for BNNs, we would like to implement XNOR-Popcount as a single instruction. To do so, you'll need to add an accelerator (new ALU). For these questions, assume that in your new accelerator, you can implement any number of new combinational logic operations (no matter how complex), and that the input binary vector `x` is already stored in the register file.

(a) (3 pts) Fill in the assembly instructions below that are needed to calculate the popcount using the faster popcount algorithm. The destination registers have been given to you, the original number is already in register `x1`, and the result should also be in register `x1` at the end:

```
srli x2
li x3
and x2
sub x1
li x3
and x2
srli x4
and x4
add x1
srli x2
add x1
li x2
and x1
srli x2
add x1
srli x2
add x1
andi x1
```

How many RV32I instructions is this?

(b) (2 pts) For this faster popcount, if you could extend RV32I with one fused (i.e. 2+ instructions combined) I-type instruction to minimize the number of instructions, what logic function should it implement? How many instructions would it save compared to part a)?

(c) (3 pts) It turns out at the hardware level, we can do a popcount without all this fancy shifting and masking, and can therefore do an XNOR-Popcount purely combinationally. Explain how and write the Verilog implementation of this.

(d) (1 pt) Is the algorithm you used in part c) practical, especially as the length of the input vectors grows (e.g. we put this in a 64-bit ISA instead)? Explain.

(e) (1 pt) Let us now expand this operation to perform a binary MAC on vectors larger than 32 bits on our 32-bit architecture. How would you do this over multiple instructions without changing the RV32I datapath, control logic, or your new ALU from part d)?

Solution:

(a)

```
srli x2 x1 1
li x3 0x55555555
and x2 x2 x3
sub x1 x1 x2
li x3 0x33333333
and x2 x1 x3
srli x4 x1 2
and x4 x4 x3
add x1 x2 x4
srli x2 x1 4
add x1 x1 x2
li x2 0x0F0F0F0F
and x1 x1 x2
srli x2 x1 8
add x1 x1 x2
srli x2 x1 16
add x1 x1 x2
andi x1 x1 0x0000003F
```

Trick question! This is actually **21 RV32I instructions** because `li rd, imm` is generally implemented as:

```
lui rd, imm_lui
addi rd, x0, imm_add
```

`lui` must come before `addi` because it forces the bottom 12 bits of `rd` to 0. The `imm_add` must be `imm[11:0]` since `lui` is unable to set those bits. Then we can calculate what `imm_lui` ought to be to get `imm` correctly loaded.

```
imm_lui = (imm - sign_ext(imm[11:0]))[31:12]
```

(b) We should add a fused I-type instruction that does the operation of `x = x + (x >> imm)`. This **saves 3 instructions**.

(c) Your Verilog module should at least have this snippet:

```
// skipped module/ports declaration
wire [31:0] a;
wire [5:0] b;
always @(*) begin
  a = x ~^ y;
  b = 0;
  for (i = 0; i < 32; i = i+1) b = b + a[i];
  dot = 2 * b - 32;
end
```

Summation of every single bit in a long chain is also acceptable, but using the `for` loop is much more concise.

(d) No, the `for` loop that is used to generate the popcount results in a large adder tree, which would be a long path in a synthesized design. The depth grows directly with the XLEN. In real hardware implementations, popcounts take multiple cycles (like other complex ops like floating-point multiply, etc.) and also use more advanced networks of half and full adder cells.

(e) We would break up the MAC over multiple XNOR-Popcount instructions, separated in between by an add instruction that would increment the running popcount. To address the cases where the input vectors are not multiples of 32 bits, there are multiple solutions. An easy way would be to make the unused bits 0's (hence making -1 * -1 = 1 multiplies), and then subtract a correction factor from the final result based on the number of filled 0's.

For your own additional information:
Theoretically, we could modify the datapath and ALU such that the ALU has 3 inputs, where the 3rd input is also the destination register. However, this is not practical because register files can't generally have more than 2 read ports, and it would make the instruction decoding far more complicated.
An alternative (though not necessarily more efficient and is not covered in this class) method would be to make our XNOR-Popcount an atomic arithmetic instruction, which reads the running count from data memory and writes back to it on the same cycle.