EECS 151/251A Homework 4

Due Friday, Oct 4th, 2019

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 table on page 130.

Problem 1: RISC-V Manual Assembly

Manually construct the binary instruction for the following assembly instructions. Provide the solution as a 32-bit binary number.

(a) `add x1, x2, x3`

(b) `addi x1, x2, 100`

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

(d) `beq x6, x8, 1024`

Solution:

You should have manually assembled these by hand for this homework, but in the future you can use GCC to do it for you:

Create a file `test.S` with contents:

```
.global _start
.section .text
_start:
  add x1, x2, x3
  addi x1, x2, 100
  lb x1, 4(x2)
  beq x6, x8, 1024
```

Run the compiler:

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

Dump the generated assembly:

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

test.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <_start>:
   0: 003100b3          add  x1,x2,x3
   4: 06410093          addi x1,x2,100
   8: 00410083          lb   x1,4(x2)
   c: 00831463          bne  x6,x8,14 <_start+0x14>
  10: 0000006f          j    10 <_start+0x10>
```

The answers are:

a) `0x003100b3 = 0b0000_0000_0011_0001_0000_0000_1011_0011`

b) `0x06410093 = 0b0000_0110_0100_0001_0000_0000_1001_0011`

c) `0x00410083 = 0b0000_0000_0100_0001_0000_0000_1000_0011`

d) `0x00831463 = 0b0000_0000_1000_0011_0001_0100_0110_0011`

Problem 2: RISC-V Assembly Programs

Write down the values of the specified registers after the following programs have run:

(a)

```
li x1, 100
li x2, 200
add x3, x1, x2
sub x3, x3, x1
```

`x1 = _____, x2 = _____, x3 = _____`

Solution:

`x1 = 100, x2 = 200, x3 = 200`

(b)

```
li x1, 100
li x0, 200
add x0, x1, x0
```

`x0 = _____, x1 = _____`

Solution:

x0 = 0, x1 = 100

(c)

```
li x1, 0xdead
li x2, 0xbeef
li x3, 0x1024
sh x1, 0(x3)
sh x2, 2(x3)
lw x4, 0(x3)
```

x1 = _____, x2 = _____, x4 = _____

Solution:

x1 = 0xdead, x2 = 0xbeef, x4 = 0xbeefdead

(d)

```
li x1, -1
li x2, 1
li x3, 0
bge x1, x2, f1
li x3, 100
f1: addi x3, x3, 100
```

x1 = _____, x2 = _____, x3 = _____

Solution:

x1 = 0xFFFFFFFF, x2 = 1, x3 = 200

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

```
li x1, 0
jalr x2, x0, 20
addi x1, x1, 100
addi x1, x1, 200
jal x0, end
jalr x3, x2, 0
end: nop
```

x1 = _____, x2 = _____, x3 = _____

Solution:

x1 = 300, x2 = 8, x3 = 24

Problem 3: RISC-V Psuedo-instructions

Several psuedo-instructions are defined by the RISC-V assembler which translate to sequences of one or more RISC-V base instructions. For each psuedo-instruction, write the RISC-V assembly instructions that implement it. Refer to page 130 of the RISC-V spec for a list of the base RISC-V instructions.

(a) `nop`. Do nothing, don't change the architectural state.

(b) `mv rd, rs`. Move the value in register `rs` to register `rd`.

(c) `li rd, imm`. Load a 32-bit immediate into register `rd`.

(d) `beqz rs, imm`. Branch if `rs` is greater than or equal to zero.

(e) `j imm`. Jump to PC += imm and don't link any register.

(f) `bgt rs1, rs2, imm`. Branch if `rs1` is greater than `rs2`.

Solution:

a) `nop`

`addi x0, x0, 0` or `add x0, x0, x0`.

Anything that writes to x0 only and doesn't modify architectural state is OK as a nop, but the RISC-V spec defines NOPs as encoded with the addi variant.

b) `mv rd, rs`

`addi rd, rs, 0` or `add rd, rs, x0`

c) `li rd, imm`

Load immediate is not straightforward to implement, but a generic solution involves a sequence of `lui` then `addi`.

```
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]`.

We will accept a naive (but incorrect) solution that just sets `imm_add = imm[11:0]` and `imm_lui = imm[31:12]`.

d) `beqz rs, imm`

`bge rs, x0, imm`

e) `j imm`

`jal x0, imm`

f) `bgt rs1, rs2, imm`

`blt rs2, rs1, imm`

Problem 4: RISC-V Instruction Decoder

Consider the complete RV32I datapath drawn in the lecture slides. Write down logical expressions for the following control signals in terms of an instruction's `opcode`, `funct3`, and `funct7` bits.

You can simplify the expressions by comparing the `opcode` field against the constants in Table 25.1 in the RISC-V ISA manual. For example: `assign sig = (opcode == OP-32) || (opcode == LOAD);`

(a) `WBSel`

(b) `MemRW`, assume `0 = read` and `1 = write`

(c) `PCSel`

(d) `BSel`

Solution:

(a) `WBSel`

```
always @(*) begin
  case (opcode)
    LOAD: WBSel = 0; // dout of DMEM
    AUIPC, LUI, OP, OP-IMM: WBSel = 1; // output of ALU
    JAL, JALR: WBSel = 2; // PC + 4
    default: WBSel = x; // store, branches
  endcase
end
```

(b) `MemRW`, assume `0 = read` and `1 = write`

```
assign MemRW = (opcode == STORE);
```

(c) `PCSel`

```
always @(*) begin
  if (opcode == BRANCH) begin
    if (funct3 == 3'b000 && BrEq) PCSel = 1; // beq
    else if (funct3 == 3'b001 & !BrEq) PCSel = 1; // bne
    else if (((funct3 == 3'b100) || (funct3 == 3'b110)) & BrLt) PCSel = 1; // blt, bltu
    else if (((funct3 == 3'b101) || (funct3 == 3'b111)) & !BrLt) PCSel = 1; // bge, bgeu
  end
  else if ((opcode == JAL) || (opcode == JALR)) PCSel = 1;
  else PCSel = 0;
end
```

(d) `BSel`

```
assign BSel = (opcode == LUI) || (opcode == AUIPC) || (opcode == BRANCH)
            || (opcode == JAL) || (opcode == JALR) || (opcode == LOAD)
            || (opcode == STORE) || (opcode == OP-IMM);
```

All instructions feed the immediate into the ALU except `OP` ones (R-type arithmetic). The shift instructions (`slli, srli, srai`), can be lumped into the `OP-IMM` instructions since the ALU will only consider the bottom 5 bits of the immediate when shifting.

Problem 5: RISC-V Memory Decoder

The RV32I ISA defines 5 memory load instructions: `lb, lbu, lh, lhu, lw`. Complete the implementation of a Verilog module which converts the raw output of the data memory to the value to be written to the regfile according to the type of load instruction and the memory address.

```
module load_decoder(
  input [31:0] addr, // the byte-address for the load instruction
  input [31:0] raw_data, // the raw data from the DMEM
  input lb, lbu, lh, lhu, lw, // type of load instruction, only 1 is high at a time
  output [31:0] wb_data // writeback data (to the regfile)
);

    // Your implementation

endmodule
```

Solution:

```
module load_decoder(
  input [31:0] addr,
  input [31:0] raw_data,
  input lb, lbu, lh, lhu, lw,
  output [31:0] wb_data
);
always @(*) begin
  wb_data = raw_data;
  if (lw) wb_data = raw_data;
  else if (lhu) begin
    if (addr[1] == 1'b0) wb_data = {16'd0, raw_data[15:0]};
    else wb_data = {16'd0, raw_data[31:16]};
  end
  else if (lh) begin
    if (addr[1] == 1'b0) wb_data = {16{raw_data[15]}, raw_data[15:0]};
    else wb_data = {16{raw_data[31]}, raw_data[31:16]};
  end
  else if (lbu) begin
    case (addr[1:0])
```

```
        2'b00: wb_data = {24'd0, raw_data[7:0]};
        2'b01: wb_data = {24'd0, raw_data[15:8]};
        2'b10: wb_data = {24'd0, raw_data[23:16]};
        2'b11: wb_data = {24'd0, raw_data[31:24]};
      endcase
    end
    else if (lb) begin
      case (addr[1:0])
        2'b00: wb_data = {24{raw_data[7]}, raw_data[7:0]};
        2'b01: wb_data = {24{raw_data[15]}, raw_data[15:8]};
        2'b10: wb_data = {24{raw_data[23]}, raw_data[23:16]};
        2'b11: wb_data = {24{raw_data[31]}, raw_data[31:24]};
      endcase
    end
  end
  endmodule
```

Problem 6: RV64I ALU

Refer to Chapter 5 (RV64I Base Integer Instruction Set) in the RISC-V spec. Implement an ALU that supports the `add, sll, sub` 64-bit integer instructions as well as their 'W' suffix variants.

```
`define ALU_ADD 0
`define ALU_ADDW 1
`define ALU_SUB 2
`define ALU_SUBW 3
`define ALU_SLL 4
`define ALU_SLLW 5
`define ALU_SRA 6
`define ALU_SRAW 7
module rv64_alu(
  input [63:0] a,
  input [63:0] b,
  input [2:0] op, // op can be any of values `define'd above
  output [63:0] c,
);

  // Your implementation

endmodule
```

Solution:

```
module rv64_alu(
  input [63:0] a,
  input [63:0] b,
```

```
  input [2:0] op,
  output [63:0] c,
);
  wire [31:0] addw = a[31:0] + b[31:0];
  wire [31:0] subw = a[31:0] - b[31:0];
  wire [31:0] sllw = a[31:0] << b[4:0];
  wire [31:0] sraw = $signed(a[31:0]) >>> b[4:0];
  always @(*) begin
    case (op)
      `ALU_ADD: c = a + b;
      `ALU_ADDW: c = {32{addw[31]}, addw};
      `ALU_SUB: c = a - b;
      `ALU_SUBW: c = {32{subw[31]}, subw};
      `ALU_SLL: c = a << b[5:0];
      `ALU_SLLW: c = {32{sllw[31], sllw};
      `ALU_SRA: c = $signed(a) >>> b[5:0];
      `ALU_SRAW: c= {32{sraw[31]}, sraw};
    endcase
  end
endmodule
```