CPE 233 Lab #6 – Complete ISA RISC-V MCU

Astrid Augusta Yu

May 11, 2020

Contents

1 Questions

1. **Briefly describe the differences between load/store and Input/Output instructions.**

   Load/store and input/output are done using lw and sw instructions in the exact same way. However, load/store will store in the memory, while Input/Output does not necessarily store data anywhere. It may write to a device, but it does not necessarily write to a device's register.

2. **Does the assembler know the differences between load/store and Input/Output instructions? Briefly but completely explain.**

   No, the assembler does not, since we are using lw and sw as the input and output instructions, which are just generic instructions that the assembler will assemble.

3. **What is the maximum number of different unique bits that you could configure the RISC-V OTTER to input? Briefly but fully explain.**

   Since IOBUS_IN is 32 bits wide, it can input $2^{32} = 4294967296$ possble unique combinations of bits.

4. **What is the maximum number of different unique bits that you could configure the RISC-V OTTER to output? Briefly but fully explain.**

   Since IOBUS_OUT is 32 bits wide, it can also output $2^{32} = 4294967296$ possble unique combinations of bits.

5. **Can you use the same I/O port address for both inputs and outputs in the same complete RISC-V OTTER implementation.**

   Yes, you can. Essentially, reading and writing are separated by the IO_WR signal. When it is brought high, a device on that address can be set into write mode, and when it is set low, it is set into read mode. The device does not have to output the same value that it received earlier.

6. **If the "memory" portion of RISC-V OTTER memory changed from 2^{16} x 8 to 2^{14} x 8, what would be the new maximum number of unique input or output bits that could be addresses?**

   Addresses are 32 bits wide, and the memory is now 14 addresses wide. Therefore, the size of the IO address space is now $2^{32} - 2^{14} = 4294950912$

7. **This experiment suggested that you include all the control signals that a particular instruction used regardless of whether they were previously scheduled to be assigned at the beginning of the always block. Briefly state why this approach represents excellent HDL coding style.**

   If a control signal is unassigned by the end of the always_comb block, then it will have a default value instead of ending up undefined and possibly breaking things.

8. **How much memory do the control units in this experiment contain? Also, state which signal in the control units represents that memory.**

   The control unit FSM stores 2 bits of memory to keep track of its state. The control unit decoder stores no memory, as it is a fully combinatorial circuit.

9. **In assembly language-land, we refer to instructions that do nothing as "nops" (pronounced "know ops"). In academia, we refer to "nops" as administrators. The nop instruction in RISC-V OTTER is a pseudoinstruction. Show at least four different ways you can implement a nop instruction in RISC-V assembly language.**

```
sub x0, x5, x3
bne x0, x0
addi x0, x0, 32
jal x0, 4
```

2 Programming Assignment

```
# Test code
.data
    input: .word 0x10, 0x20, 0x30
    output: .byte 0, 0, 0  # These should be 0x10, 0x20, 0x30 at the end
.text
test:
    la x15, input
    la x20, output
    li x25, 3

    call subroutine
stop:
    j stop

#-----------------------------------------------------------------
#- Packs a certain number of contiguous words at one location
#- into bytes at another location.
#-
#- Parameters
#-      x15 - input words start location
#-      x20 - output bytes end location
#-      x25 - Number of values to pack
#- Tweaked registers - None
#-----------------------------------------------------------------
subroutine:
    addi sp, sp, -16
    sw x15, 0(sp)
    sw x20, 4(sp)
    sw x25, 8(sp)
    sw t0, 12(sp)
loop:
    beq x25, zero, end

    lw t0, 0(x15)
    sb t0, 0(x20)

    addi x15, x15, 4
    addi x20, x20, 1
    addi x25, x25, -1
    j loop
end:
    addi sp, sp, -16
    lw x15, 0(sp)
    lw x20, 4(sp)
    lw x25, 8(sp)
    lw t0, 12(sp)
    ret
```

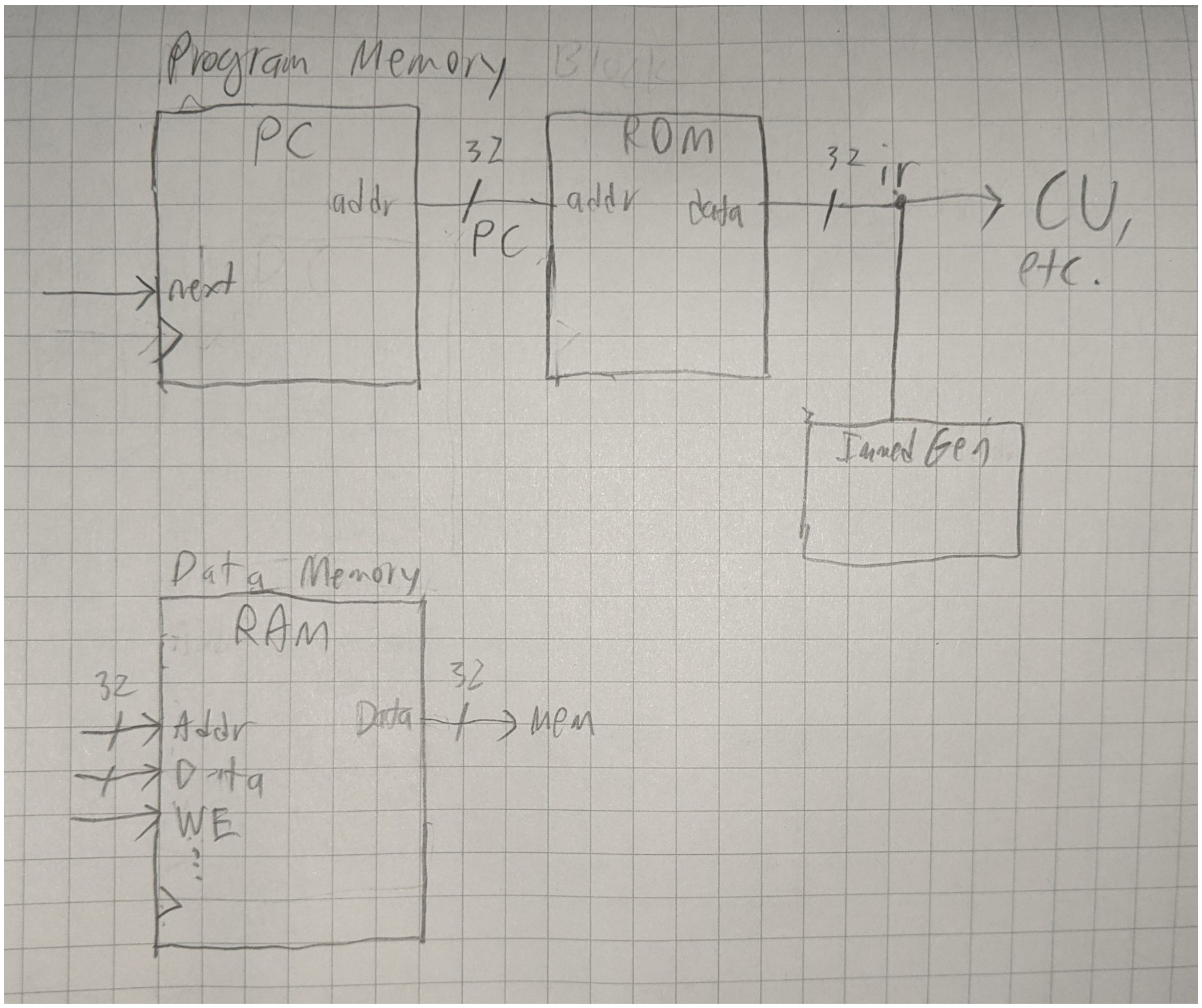

Figure 1: A diagram of the ROM/RAM configuration in the MCU. The other blocks are all identical, and thus omitted for brevity.

3 Hardware Design Assignment

Essentially, the program data can be placed on a separate ROM, and have the program counter be connected directly to it, as shown in Figure 1. The consequences of this are that the program will not be able to modify itself, accidentally or purposefully. Additionally, there would be more space to store the data, since the program has been moved off the main memory. However, this layout would necessarily mean that the MCU will require more parts and be more expensive.

4 HDL Models

4.1 Otter MCU

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly
// Engineer: Astrid Yu
//
// Create Date: 04/29/2020 02:27:42 PM
// Design Name: Main Otter MCU Module
// Module Name: OTTER_MCU
// Project Name: Otter MCU
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module OTTER_MCU #(
    parameter MEM_FILE="otter_memory.mem"
    ) (
    input RST,
    input clk,
    input [31:0] iobus_in,
    input intr,
    output iobus_wr,
    output [31:0] iobus_out,
    output [31:0] iobus_addr
    );

    // Data buses
    logic [31:0]
        pc,
        pc_inc,
        ir,
        reg_wd,
        rs1,
        rs2,

        alu_src_a_data,
        alu_src_b_data,
        alu_result,

        jalr,
        branch,
        jal,
        mem_dout,

        b_type_imm,
        i_type_imm,
        j_type_imm,
        s_type_imm,
        u_type_imm;

    // Selectors
    logic [1:0] pc_source, rf_wr_sel, alu_src_b;
    logic [3:0] alu_fun;
    logic alu_src_a;

    // Flags
    logic
        reset,
        pc_write,

        reg_write,

        mem_we2,
        mem_rden1,
        mem_rden2,

        br_eq,
        br_lt,
        br_ltu;

    assign iobus_addr = alu_result;
    assign iobus_out = rs2;

    // Multiplexers
    always_comb case(rf_wr_sel)
        4'd0: reg_wd = pc + 4;
        4'd1: reg_wd = 32'hdeadbeef;  // TODO change to CSR_reg
        4'd2: reg_wd = mem_dout;
        4'd3: reg_wd = alu_result;
    endcase

    assign alu_src_a_data = alu_src_a
        ? u_type_imm
        : rs1;

    always_comb case(alu_src_b)
        4'd0: alu_src_b_data = rs2;
        4'd1: alu_src_b_data = i_type_imm;
        4'd2: alu_src_b_data = s_type_imm;
        4'd3: alu_src_b_data = pc;
    endcase

    // Submodules
    CU_FSM fsm(
        .clk(clk),
        .RST(RST),
        .intr(intr),
        .opcode(ir[6:0]),

        .pcWrite(pc_write),
        .regWrite(reg_write),
        .memWE2(mem_we2),
        .memRDEN1(mem_rden1),
        .memRDEN2(mem_rden2),
        .reset(reset)
    );

    BranchCondGen bcg(
        .rs1(rs1),
        .rs2(rs2),
        .br_eq(br_eq),
        .br_lt(br_lt),
        .br_ltu(br_ltu)
    );

    CU_DCDR cu_dcdr(
        .opcode(ir[6:0]),
        .func7(ir[31:25]),
        .func3(ir[14:12]),

        .br_eq(br_eq),
        .br_lt(br_lt),
        .br_ltu(br_ltu),

        .alu_fun(alu_fun),
        .pcSource(pc_source),
        .alu_srcA(alu_src_a),
        .alu_srcB(alu_src_b),
        .rf_wr_sel(rf_wr_sel)
    );

    ProgramCounter prog_counter(
        .clk(clk),

        .rst(reset),
        .pc_source(pc_source),
        .pc_write(pc_write),
        .jal(jal),
        .jalr(jalr),
        .branch(branch),

        .addr(pc),
        .addr_inc(pc_inc)
    );

    Memory #(.MEM_FILE(MEM_FILE)) mem(
        .MEM_CLK (clk),
        .MEM_RDEN1 (mem_rden1),
        .MEM_RDEN2 (mem_rden2),
        .MEM_WE2 (mem_we2),
        .MEM_ADDR1 (pc[15:2]),
        .MEM_ADDR2 (alu_result),
        .MEM_DIN2 (rs2),
        .MEM_SIZE (ir[13:12]),
        .MEM_SIGN (ir[14]),
        .IO_IN (iobus_in),
        .IO_WR (iobus_wr),

        .MEM_DOUT1 (ir),
        .MEM_DOUT2 (mem_dout)
    );

    RegFile regfile(
        .clk(clk),

        .en(reg_write),
        .wd(reg_wd),
        .adr1(ir[19:15]),
        .adr2(ir[24:20]),
        .wa(ir[11:7]),

        .rs1(rs1),
        .rs2(rs2)
    );

    ImmedGen imd(
        .ir(ir[31:7]),

        .b_type_imm(b_type_imm),
        .i_type_imm(i_type_imm),
        .u_type_imm(u_type_imm),
        .j_type_imm(j_type_imm),
        .s_type_imm(s_type_imm)
    );

    BranchAddrGen bag(
        .rs(rs1),
        .pc(pc),
        .b_type_imm(b_type_imm),
        .j_type_imm(j_type_imm),
        .i_type_imm(i_type_imm),

        .jalr(jalr),
        .branch(branch),
        .jal(jal)
    );

    ALU alu(
        .alu_fun(alu_fun),
        .srcA(alu_src_a_data),
        .srcB(alu_src_b_data),
        .result(alu_result)
    );

endmodule
```

4.2 CU FSM

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer: James Ratner
//
// Create Date: 01/07/2020 09:12:54 PM
// Design Name:
// Module Name: top_level
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Control Unit Template/Starter File for RISC-V OTTER
//
//    //- instantiation template
//    module CU_FSM(
//        .intr     (),
//        .clk      (),
//        .RST      (),
//        .opcode   (),   // ir[6:0]
//        .pcWrite  (),
//        .regWrite (),
//        .memWE2   (),
//        .memRDEN1 (),
//        .memRDEN2 (),
//        .reset      ()   );
//
// Dependencies:
//
// Revision:
// Revision 1.00 - File Created - 02-01-2020 (from other people's files)
//          1.01 - (02-08-2020) switched states to enum type
//          1.02 - (02-25-2020) made PS assignment blocking
//                              made rst output asynchronous
//          1.03 - (04-24-2020) added "init" state to FSM
//                              changed rst to reset
//
//////////////////////////////////////////////////////////////////////////////////


module CU_FSM(
    input intr,
    input clk,
    input RST,
    input [6:0] opcode,     // ir[6:0]
    output logic pcWrite,
    output logic regWrite,
    output logic memWE2,
    output logic memRDEN1,
    output logic memRDEN2,
    output logic reset
  );
```

```
typedef enum logic [1:0] {
   st_INIT,
   st_FET,
   st_EX,
   st_WB
}  state_type;
state_type  NS,PS;

//- datatypes for RISC-V opcode types
typedef enum logic [6:0] {
    LUI    = 7'b0110111,
    AUIPC  = 7'b0010111,
    JAL    = 7'b1101111,
    JALR   = 7'b1100111,
    BRANCH = 7'b1100011,
    LOAD   = 7'b0000011,
    STORE  = 7'b0100011,
    OP_IMM = 7'b0010011,
    OP_RG3 = 7'b0110011
} opcode_t;
opcode_t OPCODE;    //- symbolic names for instruction opcodes

assign OPCODE = opcode_t'(opcode); //- Cast input as enum


//- state registers (PS)
always @(posedge clk) begin
    if (RST == 1)
        PS <= st_INIT;
    else
        PS <= NS;
end

always_comb begin
    //- schedule all outputs to avoid latch
    pcWrite = 1'b0;
    regWrite = 1'b0;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;

    case (PS)
        st_INIT: begin
            reset = 1'b1;
            NS = st_FET;
        end

        st_FET: begin
            memRDEN1 = 1'b1;
            NS = st_EX;
        end

        st_EX: begin
```

```
pcWrite = 1;
case (OPCODE)
    LOAD: begin
        regWrite = 0;
        memRDEN2 = 1;
        NS = st_WB;
    end

    STORE: begin
        regWrite = 0;
        memWE2 = 1;
        NS = st_FET;
    end

    BRANCH: begin
        NS = st_FET;
    end

    LUI: begin
        regWrite = 1;
        NS = st_FET;
    end

    AUIPC: begin
        regWrite = 1;
        NS = st_FET;
    end

    OP_IMM: begin
        regWrite = 1;
        memRDEN2 = 1;
        NS = st_FET;
    end

    OP_RG3: begin
        regWrite = 1;
        memRDEN2 = 1;
        NS = st_FET;
    end

    JAL: begin
        regWrite = 1;
        NS = st_FET;
     end

    JALR: begin
        regWrite = 1;
        NS = st_FET;
    end

    default: begin
        regWrite = 0;
        NS = st_FET;
    end
```

```
                endcase
            end

            st_WB: begin
                pcWrite = 0;
                regWrite = 1;
                memRDEN2 = 1;
                NS = st_FET;
            end

            default: NS = st_FET;

        endcase //- case statement for FSM states
    end

endmodule
```

4.3 CU Decoder

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Ratner Surf Designs
// Engineer: James Ratner
//
// Create Date: 01/29/2019 04:56:13 PM
// Design Name:
// Module Name: CU_Decoder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// CU_DCDR my_cu_dcdr(
//   .br_eq     (),
//   .br_lt     (),
//   .br_ltu    (),
//   .opcode    (),    //-  ir[6:0]
//   .func7     (),    //-  ir[31:25]
//   .func3     (),    //-  ir[14:12]
//   .alu_fun   (),
//   .pcSource  (),
//   .alu_srcA  (),
//   .alu_srcB  (),
//   .rf_wr_sel ()   );
//
//
// Revision:
// Revision 1.00 - File Created (02-01-2020) - from Paul, Joseph, & Celina
//          1.01 - (02-08-2020) - removed unneeded else’s; fixed assignments
//          1.02 - (02-25-2020) - made all assignments blocking
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module CU_DCDR(
    input br_eq,
    input br_lt,
    input br_ltu,
    input [6:0] opcode,   //-  ir[6:0]
    input [6:0] func7,    //-  ir[31:25]
    input [2:0] func3,    //-  ir[14:12]
    output logic [3:0] alu_fun,
    output logic [1:0] pcSource,
    output logic alu_srcA,
    output logic [1:0] alu_srcB,
    output logic [1:0] rf_wr_sel
    );

    //- datatypes for RISC-V opcode types
```

```
typedef enum logic [6:0] {
    LUI    = 7'b0110111,
    AUIPC  = 7'b0010111,
    JAL    = 7'b1101111,
    JALR   = 7'b1100111,
    BRANCH = 7'b1100011,
    LOAD   = 7'b0000011,
    STORE  = 7'b0100011,
    OP_IMM = 7'b0010011,
    OP_RG3 = 7'b0110011
} opcode_t;
opcode_t OPCODE; //- define variable of new opcode type

assign OPCODE = opcode_t'(opcode); //- Cast input enum

//- datatype for func3Symbols tied to values
typedef enum logic [2:0] {
    //BRANCH labels
    BEQ = 3'b000,
    BNE = 3'b001,
    BLT = 3'b100,
    BGE = 3'b101,
    BLTU = 3'b110,
    BGEU = 3'b111
} func3_t;
func3_t FUNC3; //- define variable of new opcode type

assign FUNC3 = func3_t'(func3); //- Cast input enum

// Branch condition selector
logic raw_branch_cond;
always_comb case(func3[2:1])
    2'b00: raw_branch_cond = br_eq;     // BEQ, BNE
    2'b10: raw_branch_cond = br_lt;     // BLT, BGE
    2'b11: raw_branch_cond = br_ltu;    // BLTU, BGEU
    default: raw_branch_cond = 0;       // ruh roh
endcase

logic branch_cond;
assign branch_cond = raw_branch_cond ^ func3[0];

logic alu_flag;
assign alu_flag = func7[5];

logic [3:0] op_alu_fun;
always_comb case(func3) inside
    3'b?01: op_alu_fun = {alu_flag, func3};
    3'b000: op_alu_fun = {alu_flag & (OPCODE == OP_RG3), func3};
    default: op_alu_fun = {1'b0, func3};
endcase

always_comb begin
    //- schedule all values to avoid latch
    pcSource = 2'b00;
```

```
rf_wr_sel = 2'b00;

alu_srcA = 1'b0;
alu_srcB = 2'b00;
alu_fun  = 4'b0000;

case(OPCODE)
    LUI: begin
        alu_fun = 4'b1001;   // lui
        alu_srcA = 1;        // u-imm
        rf_wr_sel = 2'b11;   // alu_result
    end

    AUIPC: begin
        alu_fun = 4'b0000;   // add
        alu_srcA = 1;        // u-imm
        alu_srcB = 2'd3;     // pc
        rf_wr_sel = 2'd3;   // alu_result
    end

    JAL: begin
        rf_wr_sel = 2'd0;   // next pc
        pcSource = 2'd3;     // jal
    end

    JALR: begin
        rf_wr_sel = 2'd0;   // next pc
        pcSource = 2'd1;       // jalr
    end

    LOAD: begin
        alu_fun = 4'b0000;     // add
        alu_srcA = 0;          // rs1
        alu_srcB = 2'd1;       // i imm
        rf_wr_sel = 2'd2;      // mem dout
    end

    STORE: begin
        alu_fun = 4'b0000;     // add
        alu_srcA = 1'b0;       // rs1
        alu_srcB = 2'd2;       // s imm
    end

    BRANCH:
        pcSource = branch_cond   // Invert if invert bit
            ? 2'd2      // Condition success, branch
            : 2'd0;     // Condition fail, next

    OP_IMM: begin
        pcSource = 2'b00;  // next
        alu_srcA = 1'b0;   // rs1
        alu_srcB = 2'd1;   // i imm
        rf_wr_sel = 2'd3;  // alu result
        alu_fun = op_alu_fun;  // translated func
```

```
            end

            OP_RG3: begin
                pcSource = 2'b00;  // next
                alu_srcA = 0;   // rs1
                alu_srcB = 2'd0;   // rs2
                rf_wr_sel = 2'd3;  // alu result
                alu_fun = op_alu_fun;  // translated func
            end

            default: begin
                 pcSource = 2'b00;
                 alu_srcB = 2'b00;
                 rf_wr_sel = 2'b00;
                 alu_srcA = 1'b0;
                 alu_fun = 4'b0000;
            end
        endcase
    end

endmodule
```

4.4 Branch Condition Generator

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly
// Engineer: Astrid Yu
//
// Create Date: 05/04/2020 02:43:37 PM
// Design Name: Branch Condition Generator
// Module Name: BranchCondGen
// Project Name: Otter MCU
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module BranchCondGen(
    input [31:0] rs1,
    input [31:0] rs2,
    output br_eq,
    output br_lt,
    output br_ltu
    );

    assign br_eq = rs1 == rs2;
    assign br_lt = $signed(rs1) < $signed(rs2);
    assign br_ltu = rs1 < rs2;
endmodule
```