CPE 233 Lab #5 – Limited RISC-V MCU

Astrid Augusta Yu

May 4, 2020

See it in action at
`https://web.microsoftstream.com/video/7c95873b-7c3d-4d35-b37d-9f209a627c94`

Contents

1 Questions

1. **Briefly describe the difference between programmed I/O and memory mapped I/O.**

   Programmed I/O means that there are separate instructions used to access memory and I/O devices. Memory-mapped I/O means that you access the I/O bus the same way you would access the memory, just using a different address space.

2. **Describe a situation where a NOP instruction or a NOP-type instruction would be useful.**

   If the MCU is waiting for a certain amount of time or on an external input, the NOP instruction could be useful as a "wait for a specific amount of time" instruction.

3. **Remember those setup and hold times? Whatever happened to those? Is this something we should be worried about in this Experiment? Briefly explain.**

   The setup and hold times don't really matter here because the clock speed is slow enough for all the signals to propagate through the circuit before the next clock cycle..

4. **For this experiment we were able to not include the BRANCH_COND_GEN module. Briefly explain why we were able to do this and still have the program work properly.**

   There were no branches instructions in the executed code. Therefore, the code could work without the BRANCH_COND_GEN.

5. **Could you use the addi instruction to add an immediate value of 0x3499 to another register? Briefly explain why or why not.**

   `0x3499` = `11 0100 1001 1001` making it a 14-bit number. However, addi and other i-type instructions only support 12-bit numbers. Therefore, you cannot use addi on `0x3499`.

6. **Either the jal or jalr instructions can be used to call subroutines, but only the jalrinstruction can be used to return from subroutines. Briefly explain why this is the case.**

   The subroutine could have been called from anywhere, so we can't do a relative jump because we don't have a static location that tells us where we jumped from. However, we have the call location stored in ra, and jalr can jump to an absolute address if it's stored in a register.

7. **Will you ever be required to encode pseudoinstructions. Briefly explain why or why not.**

   No. Pseudoinstructions are instructions that the assembler essentially expands into several base instructions. By implementing only the base instructions, we can implement the entire RISC-V instruction set.

8. **Briefly describe why the IOBUS_ADDR is an output from the ALU and not from a register or directly from memory.**

   All memory addresses are generated by adding the register and an immediate together in the ALU. Since I/O addresses are treated the same way as memory addresses, that's why IOBUS_ADDR is connected to the ALU.

2 Programming Assignment

```
#-----------------------------------------------------------------
#- Determines the largest and smallest unsigned byte values in
#- a given chunk of memory. Stores results in a given address.
#-
#- Parameters
#-      x30 - memory start location
#-      x10 - length of memory
#-      x8 - where to store results. Min is 1 word, followed by
#-           max, another word.
#- Tweaked registers - None
#-----------------------------------------------------------------
minmaxbytes:
    addi sp, sp, -20
    sw t0, 0(sp)        # Min
    sw t1, 4(sp)        # Max
    sw t2, 8(sp)        # RAM Value
    sw x10, 12(sp)
    sw x30, 16(sp)

    li t0, 255          # Min
    li t1, 0            # Max

loop:
    bgt x8, zero, end
    lbu t2, 0(x30)      # Load value into t2

    addi x30, x30, 1    # Increment address
    addi x10, x10, -1   # Decrement count

    bgt t2, t0, notmin  # Skip next if t2 is not min
    mv t0, t2           # If it is min, record it as such
notmin:
    blt t2, t1, loop    # Skip next if t2 is not max
    mv t1, t2           # If it is max, record it as such
    j loop

end:
    sw t0, 0(x8)
    sw t1, 4(x8)

    lw t0, 0(sp)        # Min
    lw t1, 4(sp)        # Max
    lw t2, 8(sp)        # RAM Value
    lw x10, 12(sp)
    lw x30, 16(sp)
    addi sp, sp, 20

    ret
```

3 Hardware Design Assignment

3.1 Sorting in firmware

Sorting in firmware is a lot more versatile, as you can essentially sort a list of unbounded size if your algorithm supports it. Additionally, it is easier to write a sorting algorithm in code than it is to design a sorting circuit. If written in a higher-level language, like C, it can be portable, too. However, microcontrollers are rather slow compared to what can be done in hardware, because a lot of the clock cycles are used in reading commands, not even executing them. This could also be a waste of power if that is something important in the circuit. Additionally, unless you have multiple microcontrollers or cores, it cannot be parallelized.

3.2 Sorting in hardware

Sorting in hardware is a lot faster than sorting in firmware, because you can parallelize the sorting as much as you want. However, that's at the cost of physical gates, which could be more expensive than a program written in firmware. Additionally, it takes more developer time to design and test a digital circuit than it does for a C or assembly program.

4 HDL Models

4.1 Top-Level MCU

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


module OTTER_MCU(
    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_inc;
    4'd1: reg_wd = 32'hdeadbeef;  // TODO change to CSR_reg
    4'd2: reg_wd = mem_dout;
    4'd3: reg_wd = alu_result;
endcase

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

assign alu_src_a_data = alu_src_a ? u_type_imm : rs1;

// 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)
);

// Pretend this is a branch boi
assign br_eq = 0;
assign br_lt = 0;
assign br_ltu = 0;

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(
    .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

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

        default: begin
            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

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
            pcSource = 2'b00;    // next
        end

        AUIPC: begin
            alu_fun = 4'b0000;   // add
            alu_srcA = 1;        // u-imm
            alu_srcB = 4'd3;     // pc
            rf_wr_sel = 2'b11;   // alu_result
            pcSource = 2'b00;    // next
        end

        JAL: begin
```

```
            rf_wr_sel = 2'b00;   // next pc
            pcSource = 2'b11;    // jal
        end

        LOAD: begin
            if(FUNC3 == 3'b010) begin  // instr: LW
                alu_fun = 4'b0000;     // add
                alu_srcA = 0;          // rs1
                alu_srcB = 2'd1;       // i imm
                rf_wr_sel = 2'd2;     // mem dout
                pcSource = 2'b00;      // next
            end
        end

        STORE: begin
            if(FUNC3 == 3'b010) begin  // instr: SW
                alu_fun = 4'b0000;     // add
                alu_srcA = 1'b0;       // rs1
                alu_srcB = 2'd2;       // s imm
            end
        end

        OP_IMM: begin
            case(FUNC3)
                3'b000: begin  // instr: ADDI
                    pcSource = 2'b00;  // next
                    alu_fun = 4'b0000; // add
                    alu_srcA = 1'b0;   // rs1
                    alu_srcB = 2'b01;  // i imm
                    rf_wr_sel = 2'd3;  // alu result
                end

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