RISC-V Otter MCU Development Report

Combined Technical Report – CPE 233

Author: Michael Albert
Course: CPE 233 – Computer Design and Assembly Language Programming
Date: 12/14/2025

---

1. Introduction

This cumulative report documents the design, implementation, and verification of the **Otter RISC-V Microcontroller Unit (MCU)** across seven laboratory experiments. The project begins by constructing foundational building blocks such as the **program counter** and **arithmetic logic unit (ALU)**, progressively integrating them into a functioning single-cycle RISC-V processor.

As the labs advance, the system is extended to support:

- Full RISC-V base integer instruction set (RV32I)
- Memory-mapped I/O for buttons, switches, LEDs, and 7-segment displays
- A complete control unit with fetch, execute, and write-back cycles
- Full interrupt support using CSRs (MIE, MPIE, MEPC, MTVEC)
- Timer-counter peripherals
- Firmware techniques including debouncing and interrupt-driven display control

The goal of this combined document is to present a **coherent progression** of the Otter MCU's development rather than a disconnected series of individual labs. Executive summaries from each stage provide a high-level narrative, while Section 3 synthesizes architectural concepts learned from the original question sections in the labs.

All hardware source code is consolidated in **Section 4**, and all example firmware is grouped in **Section 5**.

Image placeholders appear throughout Section 3 in positions corresponding to their conceptual relevance (Option A2).

---

2. Development Overview (Executive Summaries)

Below is a complete set of executive summaries extracted from all seven lab reports. Each summary appears under the original lab title for clarity.

2.1 Progress Report #1: The RISC-V MCU Program Counter

For this lab, I implemented the program counter for our RISC-V architecture. It counts up every four addresses to access instruction sets in RISC-V memory. I simulated it to ensure that outputs were correct. The design includes jump, branch, and jalr support and established the foundation for instruction sequencing in later labs.

2.2 The RISC-V MCU Arithmetic Logic Unit

I implemented the Arithmetic Logic Unit (ALU) for the RISC-V Otter MCU using SystemVerilog and the Xilinx Vivado toolchain. The module accepts a SEL input that determines which arithmetic or logical operation is applied to the two 32-bit operands. Simulation verified correctness for all RV32I ALU operations.

2.3 Constructing a Limited RISC-V MCU

I assembled the RISC-V MCU without interrupts using SystemVerilog and implemented a program that read a binary number from the switches, added one, and displayed the result on

the LEDs. This lab integrated the PC, ALU, register file, immediate generator, memory module, and basic control unit into a functional single-cycle CPU.

2.4 The Complete ISA RISC-V MCU

For this experiment, I implemented the full RISC-V base integer instruction set by upgrading the control unit (CU-FSM and CU-DCDR) and adding a branch-condition generator. I verified it using a comprehensive testbench. This expanded the limited MCU into a complete RV32I processor capable of executing a wide range of instructions.

2.5 The RISC-V MCU with Interrupts

In this experiment, I implemented the interrupt architecture on the RISC-V MCU by adding the CSR module and updating the control unit to support interrupt entry and exit. The MCU correctly saved return addresses, masked interrupts upon entry, vectored execution to the ISR, and restored machine state on mret.

2.6 Using Interrupts on the RISC-V MCU

I programmed an interrupt-driven application that scrolls the message **“CPE_233_is_Cool”** across the seven-segment displays on the Basys-3 board when a button is pressed. This lab introduced debounce handling, display multiplexing, and interrupt-driven firmware control.

2.7 More Interrupts on the RISC-V MCU (Timer-Counter & Debouncing)

In this experiment, I added a **timer-counter peripheral** to handle display multiplexing via interrupts and implemented **firmware debouncing** for clean button inputs. The timer-counter allowed hardware-driven timing rather than delay loops, improving system efficiency and responsiveness.

3. RISC-V Otter Development Progression

This section synthesizes all architectural understanding gained throughout the seven labs.

3.1 RISC-V and the Otter MCU

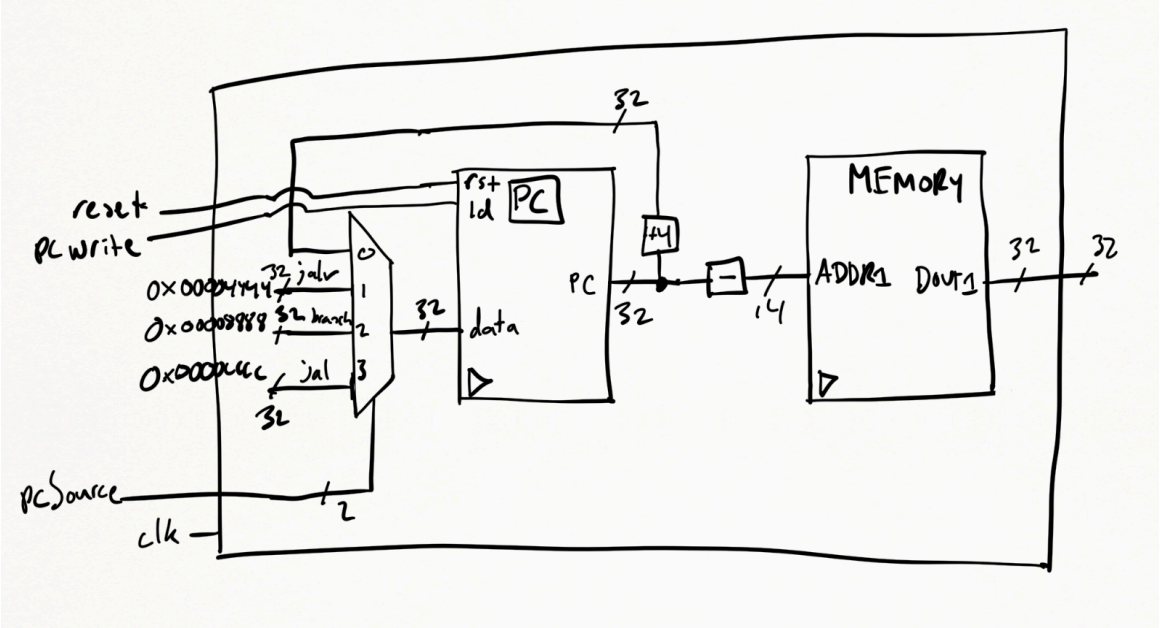
The Otter MCU implements the **32-bit RISC-V base integer instruction set (RV32I)**. It follows a classic RISC architecture emphasizing:

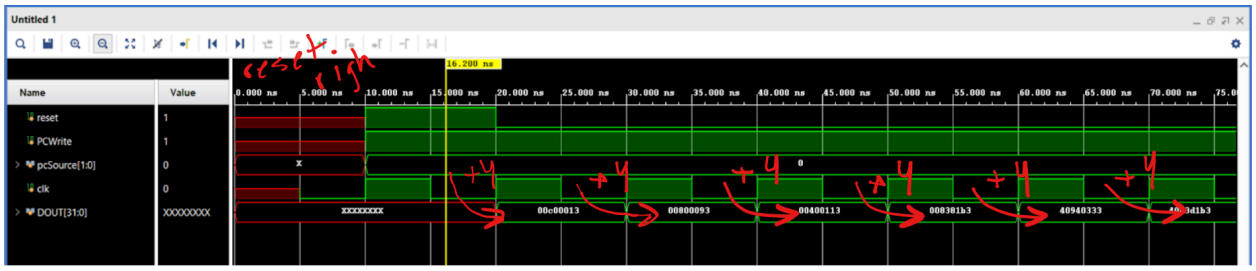
- Simple, uniform instruction execution
- A 32×32-bit register file
- Load/store memory model
- Fixed-width 32-bit instructions
- Straightforward pipeline mapping

This foundational stage of the design introduced the **program counter (PC)**, which increments by 4 each cycle unless modified by branch, jump, or jalr instructions.

Program Counter and Early Architecture Diagrams

The following figures illustrate the early Otter MCU structure and PC behavior:





A pc source signal of 0 makes the PC add 4 to the adress and outputs a new instruction from memory.

As the architecture expanded, additional components were integrated, including the ALU, immediate generator, and multiplexer network for PC selection (pcSource).

ALU Design and Functionality

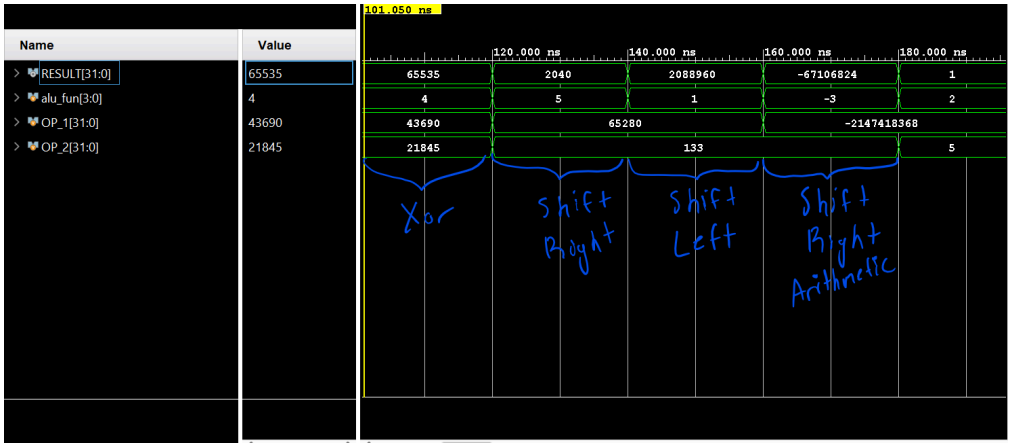
The ALU supports all arithmetic and logical operations required by RV32I, including:

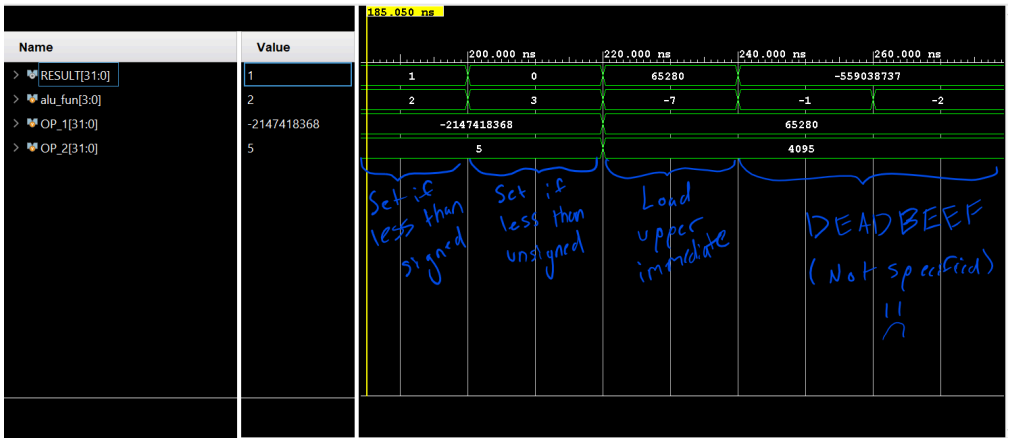
- ADD, SUB
- SLL, SRL, SRA
- XOR, OR, AND

- SLT, SLTU

Simulation waveforms validated the implementation for both signed and unsigned comparisons.

ALU Diagrams





These diagrams show the ALU datapath and example simulation traces for each operation.

3.2 Memory and Memory-Mapped I/O

The system memory is split into two conceptual regions:

1. Instruction/Data Memory Region (lower address space)

Used for instruction fetch and load/store operations.

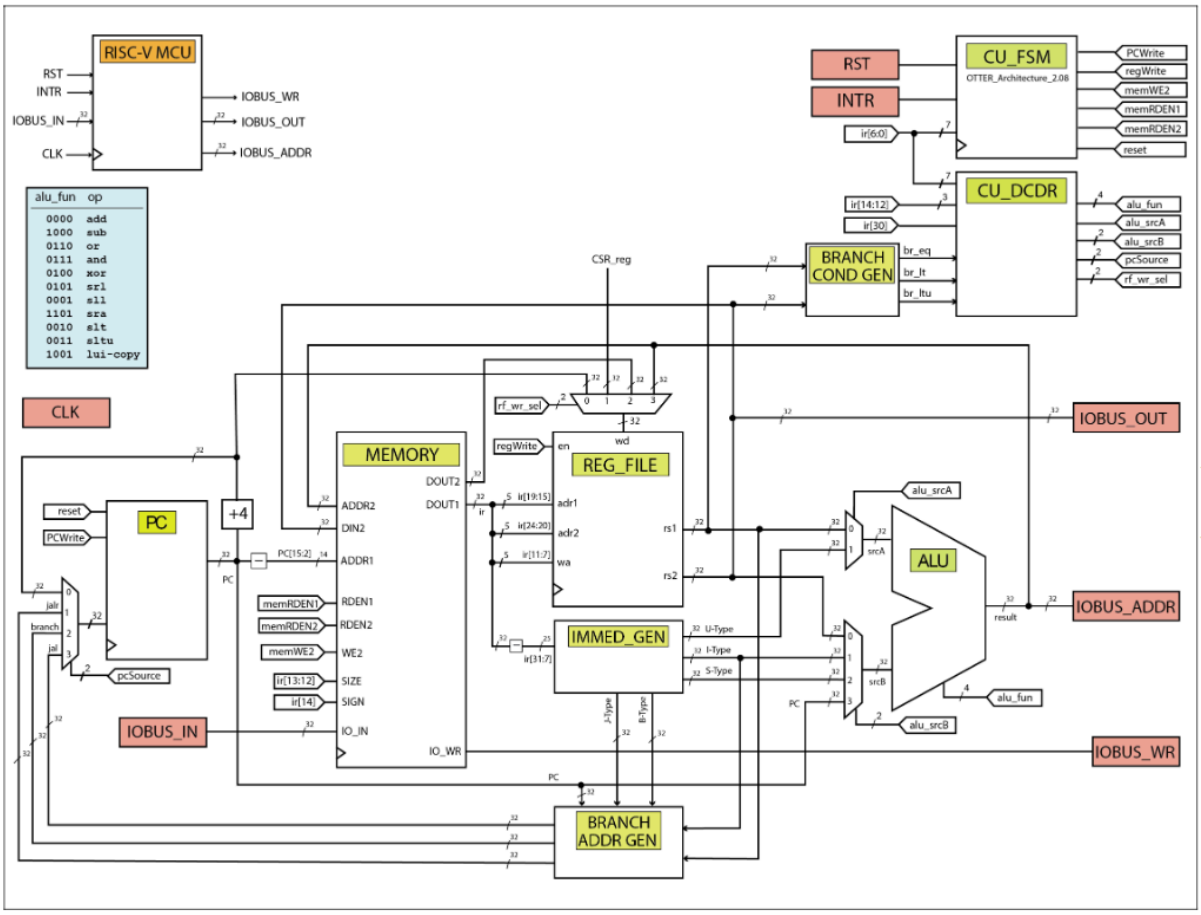
2. Memory-Mapped I/O Region (higher address space)

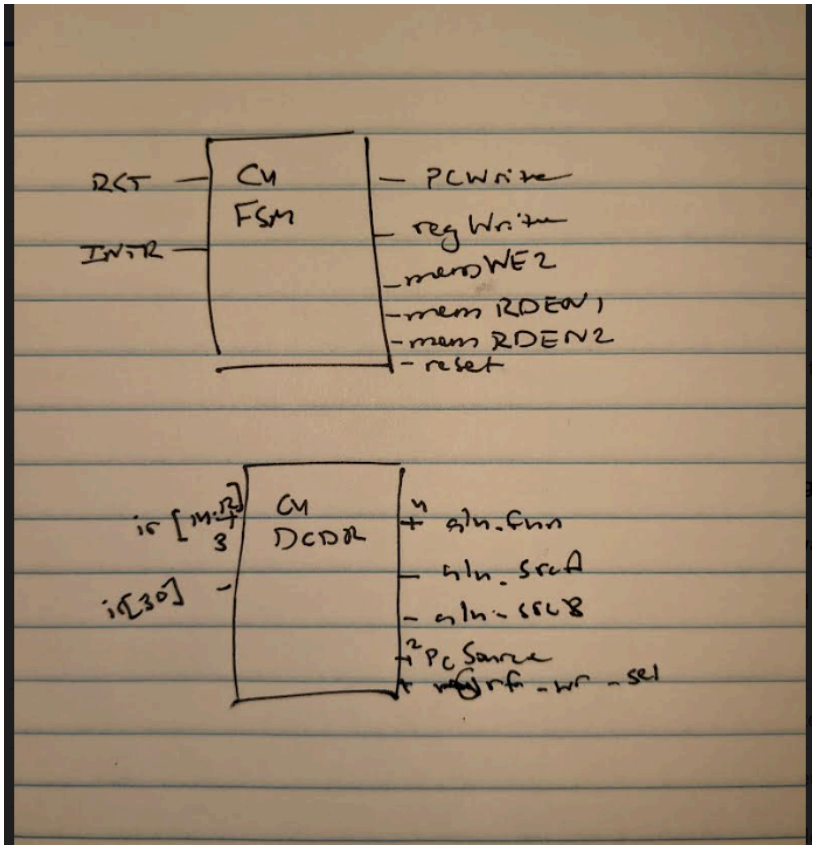
Used for:

- Switches
- Buttons
- LEDs
- Seven-segment displays
- Timer-counter registers

The Otter MCU interfaces with I/O through **IOBUS_IN**, **IOBUS_OUT**, and **IOBUS_ADDR**, enabling the CPU to read/write peripheral registers using the same load/store instructions used for memory.

Limited MCU Architecture Diagrams





This is the HLBBD's for the decoder and FSM of the RISC-V MCU

These represent the first full integration of:

- PC
- Register File
- ALU
- Memory
- Initial Control Unit (FSM + Decoder)

This stage marks the point where the Otter processor became fully capable of executing basic programs, including arithmetic and I/O operations.

3.3 Control Unit and FSM Behavior

The control unit handles:

- Instruction fetch (FET)
- Decode/execute (EX)
- Memory access
- Register write-back (WB)

In the full RV32I implementation, the CU-FSM must manage multiple paths depending on opcode and funct fields. Load instructions require three cycles; stores also require special control of write-enable signals.

Branch and jump handling requires:

- Branch condition evaluation (EQ, LT, LTU, etc.)
- Branch address generation (using immediate values)
- PC selection logic (pcSource)

This development features:

- Full RV32I CPU datapath
- Complete ALU integration
- Branch logic and jump paths
- Expanded decoder and FSM behavior

3.4 Interrupt Architecture and ISRs

The Otter MCU's interrupt handling follows RISC-V machine-mode conventions:

Hardware Responsibilities:

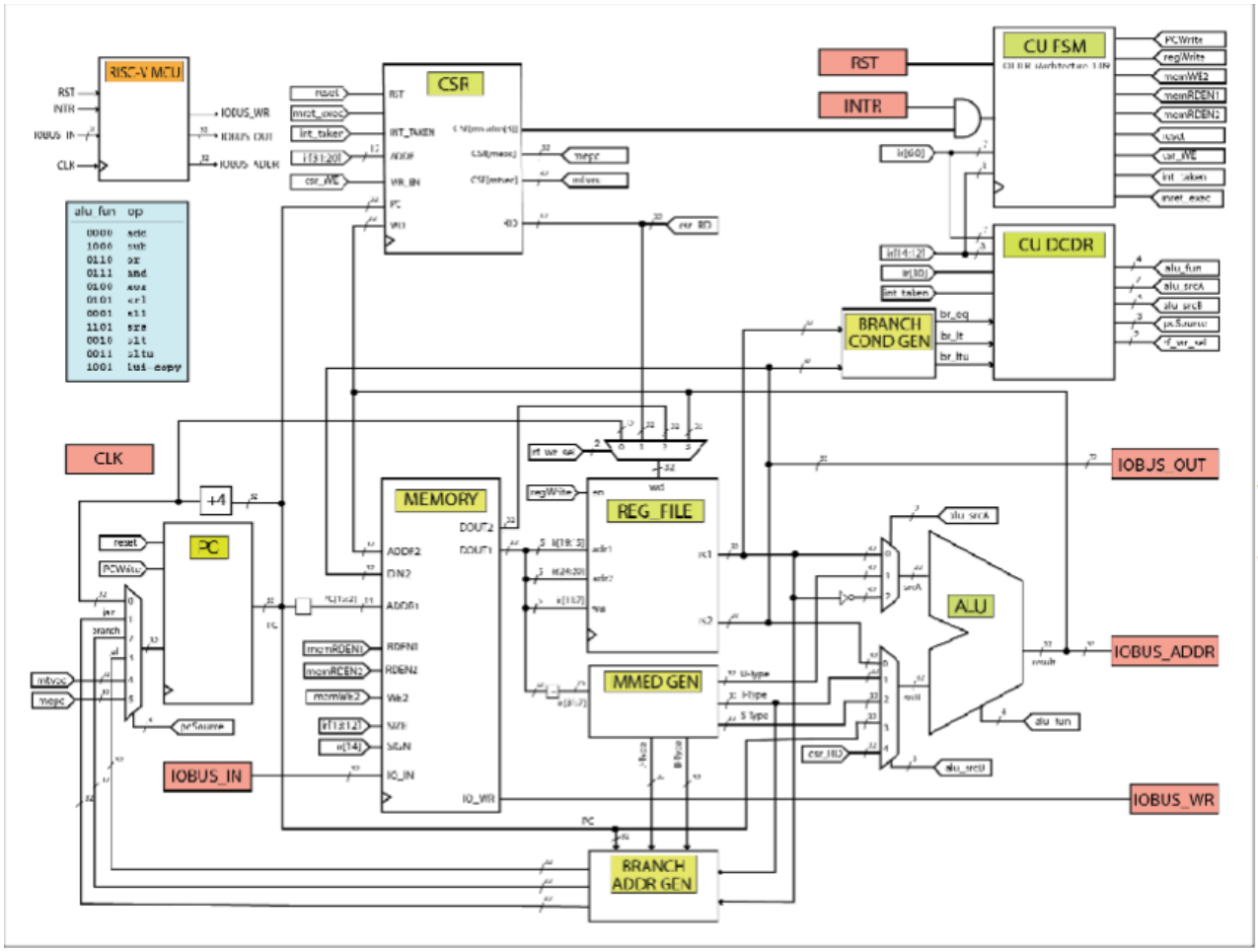
- Save PC into **MEPC**
- Copy **MIE** into **MPIE**
- Clear **MIE** to mask interrupts
- Jump to **MTVEC** (ISR address)

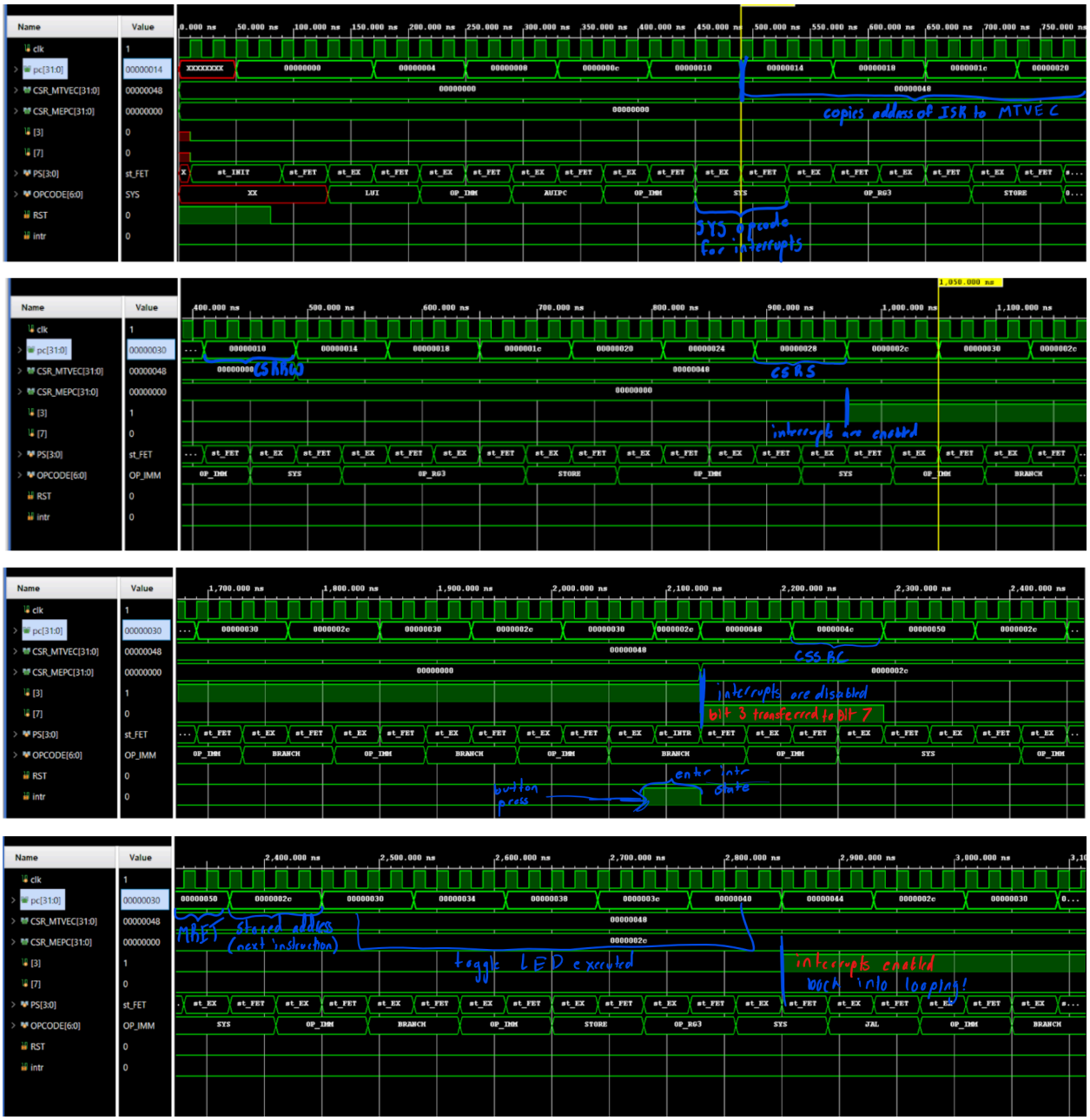
Software Responsibilities:

- Respond to the interrupt
- Optionally modify registers
- Execute **mret** to restore normal operation

This prevents nested interrupts unless specifically enabled by hardware changes (not implemented here).

OTTER Architecture Diagram with Interrupts





These images show the interrupt state in the finite state machine and CSR update mechanism.

3.5 Timer-Counters and Debouncing

The **timer-counter peripheral** provides hardware-based timing for:

- Display multiplexing
- Periodic interrupts
- General-purpose timing tasks

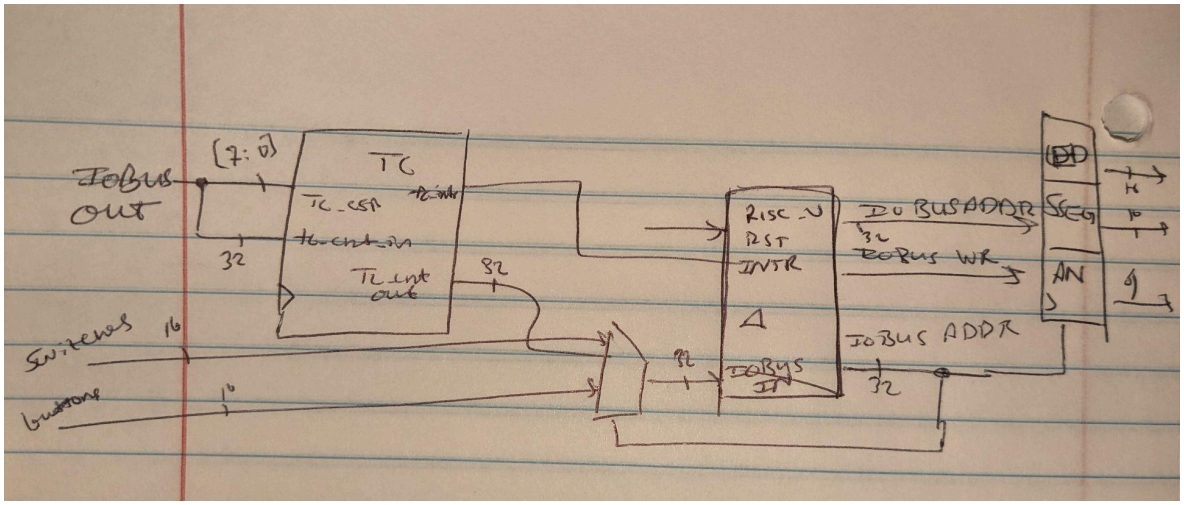
It compares a free-running counter to a programmed count and asserts its interrupt flag for three cycles.

Debouncing

Mechanical button bounce can generate false transitions. The lab explored:

1. **Hardware Debounce Module**
2. **Firmware Debounce Logic** using repeated reads and stabilization thresholds

Timer-Counter and Debounce Diagrams



This diagram illustrates:

- Timer-counter registers
- Counter operation
- Firmware debouncing flowchart
- ISR timing flow

3.6 Polling vs. Interrupts in Firmware

Polling requires the CPU to continuously check the status of inputs, which:

- Consumes CPU time
- Increases latency
- Reduces system responsiveness

Interrupt-driven design:

- Allows CPU to perform other tasks
- Responds immediately when events occur
- Reduces energy use (in embedded contexts)

Demo Program:

• LED port address = 0x1100C000
• Input port address = 0x11004444
• Doesn't use I/O instructions in the interrupt service routine
• Minimize the number of instructions

```
#-------------------------------------------------------------
----------------------
#-------------------------------------------------------------
----------------------
#- This program takes in 16 values from a lut and adds them
after each interrupt is triggered.
#- After all values are added up (after all interrupts
triggered) it averages the values and displays on the LEDs.
#-------------------------------------------------------------
----------------------
```

```
.data # 16 total values
my_lut:  .byte 0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x08, #LED
output patterns
            .byte 0x0C, 0x0E, 0x0F, 0x10, 0x18, 0x1C, 0x1E, 0x1F
.text
main:
init: li x16,0x1100C000      #load address of LEDs into x16
      li  x6,ISR             #load address of ISR into x6
    csrrw x0,mtvec, x6       #load address of ISR into CSR[mtvec]
       la x4,my_lut          #load address of LUT into my_lut
       li x5,0x00011C000     #load address of button into x5
        li x12,0              #load accumulator
        li x20,16             #loop admin
        sw x4,0(x13)          #store word of lookup table in x13

        mv x8,x0              #use x8 as flag
        mv x20,x0             #clear x20
        sw x20,0(x15)         #turn off all LEDs
unmask: li x10,0x8            #set bit[3] value in x10
        csrrs x0,mstatus,x10      # enable interrupts: set MIE in CSR
loop:  nop                    #do nothing
       beq x8,x0,loop         #wait for interrupt
addval:beq x20,0,avg          #when done looping, jump to vals
        lb  x11,0(x13)        #load byte of x13 into x11
       srli x13,x13,4         #shift for next value
       addi x20,x20,-1        #store halfword into LED
       add  x12,x12,x11       #increment to accumulator
       mv   x8,x0             #clear flag
       csrrs x0,mstatus,x10   #enable interrupt
       j     loop             #return to loop
Avg:  srli x12,4              #divide by 16 to get average
      sw  x12,(x16)           #store word x12 into x16

#-------------------------------------------------------------------
-----------------
#-------------------------------------------------------------------
-----------------
#- The ISR: sets bit x8 to act as flag
#-------------------------------------------------------------------
-----------------
ISR: li  x8,0x1            #set flag
     li  x9,0x80           #set bit7
  cssrc  x0,mstatus,x9     #clear bit7
done:    mret              #return back
```

4. Combined Source Code

This section consolidates **all hardware source code** used across Labs 1–7, grouped by the experiment they originally came from. All code is presented in clean, copy/paste-safe monospace format compatible with Google Docs.

4.1 The RISC-V MCU Program Counter — Source Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Module: Program_Counter
// Description: The PC increments by 4 normally, and supports
// branch, jump, and jalr operations.
//////////////////////////////////////////////////////////////////
module Program_Counter(
    input rst,              //reset
    input PCWrite,          //load for reg
    input [1:0] pcSource,   //sel for mux
    input clk,              //clock
    output [31:0] pc        //output of program counter
    );

logic [31:0] input_data;    //connection between reg and mux output


reg_nb_sclr REGISTER(
     .data_in(input_data), //input of reg is output of mux
     .data_out(pc),        //output of reg is output of PC
     .clk(clk),
     .clr(rst),
     .ld(PCWrite));

mux_4t1_nb MUX(
     .SEL(pcSource),
     .D0(pc + 32'h00000004), //Add 4 to go to next instruction
     .D1(32'h00004444),      //jal
     .D2(32'h00008888),      //branch
     .D3(32'h0000CCCC),      //jalr
     .D_OUT(input_data));    //outputs into reg
```

```
endmodule
```

4.2 The RISC-V MCU Arithmetic Logic Unit — Source Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Module: ALU
// Description: Supports RV32I arithmetic and logical ops.
//////////////////////////////////////////////////////////////////
module ALU(
    input [31:0] OP1,
    input [31:0] OP2,
    input [3:0]  SEL,
    output logic[31:0] RESULT
    );

always @ (SEL, OP1, OP2) begin
 case(SEL)
    4'b0000: RESULT <= OP1 + OP2;
    4'b1000: RESULT <= OP1 - OP2;
    4'b0110: RESULT <= OP1 | OP2;
    4'b0111: RESULT <= OP1 & OP2;
    4'b0100: RESULT <= OP1 ^ OP2;
    4'b0101: RESULT <= OP1 >> OP2[4:0];
    4'b0001: RESULT <= OP1 << OP2[4:0];
    4'b1101: RESULT <= ($signed(OP1) >>> OP2[4:0]);
    4'b0010: RESULT <= ($signed(OP1) < $signed(OP2)) ? 1:0;
    4'b0011: RESULT <= (OP1 < OP2) ? 1:0;
    4'b1001: RESULT <= OP1;
    default: RESULT <= $signed(32'hDEADBEEF);

    endcase
    end


endmodule
```

4.3 Constructing a Limited RISC-V MCU — Source Code

This includes the basic CPU integration: PC, ALU, Register File, Immediate Generator, Memory, CU_FSM, and CU_DCDR.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Module: RISV_V_MCU
// Description: Limited RISC-V MCU without interrupts.
//////////////////////////////////////////////////////////////////
module RISV_V_MCU(
    input clk,
    input INTR,
    input RST,
    input [31:0] IOBUS_IN,
    output IOBUS_WR,
    output [31:0] IOBUS_ADDR,
    output [31:0] IOBUS_OUT

    );

    logic  PCWrite;
    logic [1:0] pcSource   ;
    logic [31:0]U_Type_Imm ;
    logic [31:0]S_Type_Imm ;
    logic [31:0]J_Type_Imm ;
    logic [31:0]B_Type_Imm ;
    logic [31:0]I_Type_Imm ;
    logic [31:0]pc         ;
    logic [31:0]jal       ;
    logic [31:0]jalr       ;
    logic [31:0]branch     ;
    logic [31:0]ir         ;
    logic [31:0]D_OUT_TWO    ;
```

```
    logic reg_Write           ;
    logic [31:0] rs1          ;
    logic [31:0] rs2          ;
    logic [31:0] wd         ;
    logic alu_srcA           ;
    logic [31:0] srcA          ;
    logic [31:0] alu_srcB          ;
    logic [31:0] srcB              ;
    logic [3:0]  alu_fun            ;
    logic [31:0] result           ;
    logic [1:0] rf_wr_sel          ;
    logic reset             ;
    logic memWE2 ;
    logic memRDEN1 ;
    logic memRDEN2       ;
    logic regWrite      ;


IG Immed_Gen(.ir(ir[31:7]),
             .U_Type_Imm(U_Type_Imm),
             .J_Type_Imm(J_Type_Imm),
             .S_Type_Imm(S_Type_Imm),
             .B_Type_Imm(B_Type_Imm),
             .I_Type_Imm(I_Type_Imm)
            );

RISC_V_BAG Branch_Addr_Gen( .J_Type_Imm(J_Type_Imm),
                            .B_Type_Imm(B_Type_Imm),
                            .I_Type_Imm(I_Type_Imm),
                            .rs(rs1),
                            .pc(pc),
                            .jalr(jalr),
                            .jal(jal),
                            .branch(branch)
                           );

Program_Counter PC( .rst(reset),
                    .PCWrite(PCWrite),
                    .pcSource(pcSource),
                    .clk(clk),
                    .pc(pc),
```

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

Memory Otter_Mem(
                   .MEM_CLK      (clk),
                   .MEM_RDEN1    (memRDEN1),
                   .MEM_RDEN2    (memRDEN2 ),
                   .MEM_WE2      (memWE2),
                   .MEM_ADDR1    (pc[15:2]),
                   .MEM_ADDR2    (result),
                   .MEM_DIN2     (),
                   .MEM_SIZE     (ir[13:12]),
                   .MEM_SIGN     (ir[14]),
                   .IO_IN        (IOBUS_IN),
                   .IO_WR        (IOBUS_WR),
                   .MEM_DOUT1    (ir),
                   .MEM_DOUT2    (D_OUT_TWO)
     );

 RegFile RegFile(
        .clk(clk),
        .adr1(ir[19:15]),
        .adr2(ir[24:20]),
        .wa(ir[11:7]),
        .en(regWrite),
        .rs1(rs1),
        .rs2(rs2),
        .wd(wd));

mux_2t1_nb SRCA_MUX(
    .SEL(alu_srcA),
    .D0(rs1),
    .D1(U_Type_Imm),
    .D_OUT(srcA));

mux_4t1_nb srcBMux(
    .SEL(alu_srcB),
    .D0(rs2),
    .D1(I_Type_Imm),
```

```
    .D2(S_Type_Imm),
    .D3(pc),
    .D_OUT(srcB));

ALU ALU(
    .OP1(srcA),
    .OP2(srcB),
    .SEL(alu_fun),
    .RESULT(result)
    );

CU_DCDR Decoder(
    .br_eq(),
    .br_lt(),
    .br_ltu(),
    .opcode(ir[6:0]),
    .func7(ir[30]),
    .func3(ir[14:12]),
    .alu_fun(alu_fun),
    .alu_srcA(alu_srcA),
    .alu_srcB(alu_srcB),
    .pcSource(pcSource),
    .rf_wr_sel(rf_wr_sel));

mux_4t1_nb REG_MUX(
    .SEL(rf_wr_sel),
    .D0(pc + 32'h00000004),
    .D1(),
    .D2(D_OUT_TWO),
    .D3(result),
    .D_OUT(wd)
    );

CU_FSM FSM(
    .RST(RST),
    .intr(INTR),
    .opcode(ir[6:0]),
    .clk(clk),
    .reset(reset),
    .pcWrite(PCWrite),
    .memWE2(memWE2),
```

```
    .memRDEN1(memRDEN1),
    .memRDEN2(memRDEN2),
    .regWrite(regWrite));

assign IOBUS_ADDR = result;
assign IOBUS_OUT = rs2;
assign IOBUS_ADDR = result;

end module
```

4.4 The Complete ISA RISC-V MCU — Source Code

(Contains the full FSM and Decoder supporting all RV32I instructions.)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer: James Ratner, Modifications by Michael Albert
//
// Create Date: 01/07/2020 09:12:54 PM
// Design Name:
// Module Name: FSM
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Control Unit Template/Starter File for RISC-V OTTER
//
//     //- instantiation template
//     CU_FSM my_fsm(
//        .intr     (xxxx),
//        .clk      (xxxx),
```

```
//         .RST      (xxxx),
//         .opcode   (xxxx),   // ir[6:0]
//         .pcWrite  (xxxx),
//         .regWrite (xxxx),
//         .memWE2   (xxxx),
//         .memRDEN1 (xxxx),
//         .memRDEN2 (xxxx),
//         .reset    (xxxx)   );
//
// 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
//          1.04 - (04-29-2020) removed typos to allow synthesis
//          1.05 - (10-14-2020) fixed instantiation comment
(thanks AF)
//          1.06 - (12-10-2020) cleared most outputs, added
commentes
//          1.07 - (05-01-2023) fixed indentation and formatting
//
//////////////////////////////////////////////////////////////////
//////////////////

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  //waiting state
        reset = 1'b1;
        NS = st_FET;
    end

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

    st_EX: begin    //decode + execute
        pcWrite = 1'b1;
        case (OPCODE)
            LOAD: begin   //LW
                regWrite = 1'b0;
                memRDEN2 = 1'b1;
                pcWrite = 1'b0;
                NS = st_WB;
            end

            AUIPC: begin //AUIPC
                regWrite = 1'b1;
                NS = st_FET;
                end

            JALR: begin //JALR
                regWrite = 1'b1;
                NS = st_FET;
                end

            STORE: begin   //Store
                regWrite = 1'b0;
                memWE2 = 1'b1;
                NS = st_FET;
            end

            BRANCH: begin   //Branch
                regWrite = 1'b0;
                NS = st_FET;
            end
```

```
                LUI: begin      //LUI
                    regWrite = 1'b1;
                    NS = st_FET;
                end

                OP_IMM: begin // immediate opps
                    regWrite = 1'b1;
                    NS = st_FET;
                end

                JAL: begin   //JAL
                    regWrite = 1'b1;
                    pcWrite = 1'b1;
                    NS = st_FET;
                end

                OP_RG3: begin   // non-immediate opps
                    regWrite = 1'b1;
                    NS = st_FET;
                end

                default: begin
                    NS = st_FET;
                end

            endcase
        end

        st_WB: begin    //For Load Instructions
           regWrite = 1'b1;
           pcWrite = 1'b1;
           memRDEN2 = 1'b0;
           NS = st_FET;
        end

        default: NS = st_FET;

    endcase //- case statement for FSM states
end
```

```
endmodule



`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Ratner Surf Designs
// Engineer: James Ratner, Modifications by Michael Albert
//
// Create Date: 01/29/2019 04:56:13 PM
// Design Name:
// Module Name: CU_Decoder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:   This is the decoder for the
//                RISC-V otter with all 37 base instructions implemented
//
// Dependencies:
//
// CU_DCDR my_cu_dcdr(
//    .br_eq     (),
//    .br_lt     (),
//    .br_ltu    (),
//    .opcode    (),    //-  ir[6:0]
//    .func7     (),    //-  ir[30]
//    .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
//          1.03 - (05-12-2020) - reduced func7 to one bit
//          1.04 - (05-31-2020) - removed misleading code
//          1.05 - (05-01-2023) - reindent and fix formatting
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////////////

module CU_DCDR(
    input br_eq,
    input br_lt,
    input br_ltu,
    input [6:0] opcode,   //-  ir[6:0]
    input func7,          //-  ir[30]
    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

    typedef enum logic [0:0] {
        //func7 labels
        SRLI = 1'b0,
        SRAI = 1'b1
    } func7_t;
    func7_t FUNC7; // -define variable of new opcode type

    assign FUNC7 = func7_t'(func7);

    always_comb begin
        //- schedule all values to avoid latch
        pcSource = 2'b00;  alu_srcB = 2'b00;    rf_wr_sel =
2'b00;
        alu_srcA = 1'b0;   alu_fun  = 4'b0000;

        case(OPCODE)
            LUI: begin              //LUI
                alu_fun = 4'b1001;
                alu_srcA = 2'b01;
                rf_wr_sel = 2'b11;
            end

            JAL: begin              //Jal
                rf_wr_sel = 2'b00;
                pcSource = 3'b011;
            end

            JALR: begin
                alu_fun = 4'b0000;
                alu_srcA = 2'b00;
```

```
    alu_srcA = 3'b000;
    pcSource = 1'b001;
    rf_wr_sel= 2'b00;

end

LOAD: begin             //Load
    alu_fun = 4'b0000;
    alu_srcA = 2'b00;
    alu_srcB = 2'b01;
    rf_wr_sel = 2'b10;
end

STORE: begin            //SW
    alu_fun = 4'b0000;
    alu_srcA = 2'b00;
    alu_srcB = 2'b10;
end

AUIPC: begin //AUIPC
    alu_fun = 4'b0000;
    alu_srcA = 2'b01;
    alu_srcB = 2'b11;
    rf_wr_sel = 2'b11;
    end

OP_RG3: begin          //Ops not immed
    alu_srcA = 2'b00;
    alu_srcB = 2'b00;
    rf_wr_sel = 2'b11;
    case(FUNC3)
        3'b000: begin  //ADD or SUB
            case(FUNC7)
                1'b0: begin //ADD
                    alu_fun = 4'b0000;
                end

                1'b1: begin //SUB
                    alu_fun = 4'b1000;
                end
             endcase
```

```
        end

    3'b001: begin //SLL
        alu_fun = 4'b0001;
    end

    3'b010: begin   //SLT
        alu_fun = 4'b0010;
    end

    3'b011: begin   //SLTU
        alu_fun  = 4'b0011;
    end

    3'b100: begin //XOR
        alu_fun = 4'b0100;
    end

    3'b101: begin //SRL or SRA
        case(FUNC7)
            1'b0: begin //SRL
                alu_fun = 4'b0101;
            end

            1'b1: begin //SRA
                alu_fun = 4'b1101;
            end
        endcase
        end

    3'b110: begin //OR
        alu_fun = 4'b0110;
    end

    3'b111: begin //AND
        alu_fun = 4'b0111;

      end
endcase
```

```
                end

                OP_IMM: begin      //immediate opperations
                    alu_srcA = 2'b00;
                    alu_srcB = 2'b01;
                    rf_wr_sel = 2'b11;
                    case(FUNC3)
                        3'b000: begin    // instr: ADDI
                            alu_fun = 4'b0000;

                        end

                        3'b010: begin  // instr:  SLTI
                            alu_fun = 4'b0010;

                        end

                        3'b011: begin //INSTR: SLTIU
                            alu_fun = 4'b0011;

                         end

                         3'b110: begin //INSTR: ORI
                            alu_fun = 4'b0110;
;
                         end

                         3'b100: begin //INSTR: XORI
                            alu_fun = 4'b0100;

                         end

                         3'b111: begin //INSTR: ANDI
                            alu_fun = 4'b0111;

                         end

                        3'b001: begin //INSTR: SLLI
                            alu_fun = 4'b0001;
```

```
            end


            3'b101: begin //INSTR: SRLI and SRAI
               case(FUNC7)
                 SRLI: begin //SRLI
                   alu_fun = 4'b0101;
                 end

                 SRAI: begin //SRAI
                    alu_fun = 4'b1101;
                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_fun = 4'b0000;
                alu_srcA = 1'b0;
                alu_srcB = 2'b00;
                rf_wr_sel = 2'b00;
            end

        endcase
    end

    BRANCH: begin
         rf_wr_sel = 2'b11;
         case(FUNC3)

            BEQ: begin   //Branch if Equal
                if (br_eq) begin
                    pcSource = 2'b10;
```

```
                end

                else begin
                    pcSource = 2'b00;
                end
                end

            BNE: begin //Branch if Not Equal
                if (~br_eq) begin
                    pcSource = 2'b10;
                end

                else begin
                    pcSource = 2'b00;
                end
                end

            BLT: begin //Branch if Less Than
                if (br_lt) begin
                    pcSource = 2'b10;
                end

                else begin
                    pcSource = 2'b00;
                end
                end

            BGE: begin          //Branch GE or equal
                if (~br_lt | br_eq) begin
                    pcSource = 2'b10;
                end

                else begin
                    pcSource = 2'b00;
                end
                end

            BLTU: begin               //Branch if less
//                                      than unsigned
               if (br_ltu) begin
                    pcSource = 2'b10;
```

```
                        end

                        else begin
                            pcSource = 2'b00;
                        end
                        end
                    BGEU: begin                         //Branch GEU
                       if (~br_ltu | br_eq) begin
                            pcSource = 2'b10;
                        end

                        else begin
                            pcSource = 2'b00;
                        end
                     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
```

4.5 The RISC-V MCU with Interrupts — Source Code

Contains ISR entry/exit logic, CSR write-enable, and mret handling.

```
//////////////////////////////////////////////////////////////////
```

```
// CU_FSM with interrupt support
//////////////////////////////////////////////////////////////////
module CU_FSM(
    input intr,
    input clk,
    input RST,
    input [2:0] func3,      //-  ir[14:12]
    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,
    output logic csr_WE,
    output logic int_taken,
    output logic mret_exec

);

    typedef  enum logic [3:0] {
       st_INIT,
       st_FET,
       st_EX,
       st_WB,
       st_INTR
    }  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,
        SYS    = 7'b1110011
    } 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;
    mret_exec = 1'b0;  int_taken = 1'b0;    csr_WE = 1'b0;

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

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

        st_EX: begin    //decode + execute
            pcWrite = 1'b1;
            case (OPCODE)
                LOAD: begin   //LW
                    regWrite = 1'b0;
                    memRDEN2 = 1'b1;
                    pcWrite = 1'b0;
                    NS = st_WB;
                end

                AUIPC: begin //AUIPC
                    regWrite = 1'b1;
                    if (intr)
                        NS = st_INTR;
                    else
                        NS = st_FET;
                    end
```

```
JALR: begin //JALR
    regWrite = 1'b1;
    if (intr)
        NS = st_INTR;
    else
        NS = st_FET;

    end

STORE: begin   //Store
    regWrite = 1'b0;
    memWE2 = 1'b1;
    if (intr)
        NS = st_INTR;
    else
        NS = st_FET;
end

BRANCH: begin   //Branch
    regWrite = 1'b0;
    if (intr)
        NS = st_INTR;
    else
        NS = st_FET;
end

LUI: begin      //LUI
    regWrite = 1'b1;
    if (intr)
        NS = st_INTR;
    else
        NS = st_FET;
end

OP_IMM: begin // immediate opps
    regWrite = 1'b1;
    if (intr)
        NS = st_INTR;
    else
        NS = st_FET;
end

JAL: begin   //JAL
    regWrite = 1'b1;
    pcWrite = 1'b1;
```

```
            if (intr)
                NS = st_INTR;
            else
                NS = st_FET;
        end

        OP_RG3: begin   // non-immediate opps
            regWrite = 1'b1;
            if (intr)
                NS = st_INTR;
            else
                NS = st_FET;
        end

        SYS: begin
            if (func3 == 3'b000) begin
                mret_exec = 1'b1;
                if (intr)
                    NS = st_INTR;
                else
                    NS = st_FET;
                end
            else begin
                regWrite = 1'b1;
                csr_WE = 1'b1;
                if (intr)
                    NS = st_INTR;
                else
                    NS = st_FET;
                end
                end

        default: begin
            NS = st_FET;
        end

    endcase
end

st_WB: begin    //For Load Instructions
   regWrite = 1'b1;
   pcWrite = 1'b1;
   memRDEN2 = 1'b0;
   if (intr)
      NS = st_INTR;
```

```
                else
                    NS = st_FET;
            end

            st_INTR: begin  //interupt
                int_taken = 1'b1;
                pcWrite = 1'b1;
                NS = st_FET;
            end

            default: NS = st_FET;

        endcase //- case statement for FSM states
    end


endmodule

module CU_DCDR(
    input br_eq,
    input br_lt,
    input br_ltu,
    input [6:0] opcode,    //-  ir[6:0]
    input func7,           //-  ir[30]
    input [2:0] func3,     //-  ir[14:12]
    input int_taken,
    output logic [3:0] alu_fun,
    output logic [2:0] pcSource,
    output logic [1:0] alu_srcA,
    output logic [2:0] 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,
        SYS    = 7'b1110011
    } 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

typedef enum logic [0:0] {
    //func7 labels
    SRLI = 1'b0,
    SRAI = 1'b1
} func7_t;
func7_t FUNC7; // -define variable of new opcode type

assign FUNC7 = func7_t'(func7);

always_comb begin
    //- schedule all values to avoid latch
    pcSource = 3'b000;  alu_srcB = 3'b000;    rf_wr_sel = 2'b00;
    alu_srcA = 2'b00;   alu_fun  = 4'b0000;

    if (int_taken) begin
        pcSource <= 3'b100;
        alu_srcB <= 3'b000;
        rf_wr_sel <= 2'b00;
        alu_srcA <= 2'b00;
        alu_fun  <= 4'b0000;
    end

    else begin

        case(OPCODE)
            LUI: begin                //LUI
                alu_fun = 4'b1001;
```

```
    alu_srcA = 2'b01;
    rf_wr_sel = 2'b11;
end

JAL: begin                //Jal
    rf_wr_sel = 2'b00;
    pcSource = 3'b011;
end

JALR: begin
    alu_fun = 4'b0000;
    alu_srcA = 2'b00;
    pcSource = 3'b001;
    rf_wr_sel= 2'b00;

end

LOAD: begin             //Load
    alu_fun = 4'b0000;
    alu_srcA = 2'b00;
    alu_srcB = 3'b001;
    rf_wr_sel = 2'b10;
end

STORE: begin            //SW
    alu_fun = 4'b0000;
    alu_srcA = 2'b00;
    alu_srcB = 3'b010;
end

AUIPC: begin //AUIPC
    alu_fun = 4'b0000;
    alu_srcA = 2'b01;
    alu_srcB = 3'b011;
    rf_wr_sel = 2'b11;
end

OP_RG3: begin          //Ops not immed
    alu_srcA = 2'b00;
    alu_srcB = 3'b000;
    rf_wr_sel = 2'b11;
    case(FUNC3)
        3'b000: begin  //ADD or SUB
            case(FUNC7)
            1'b0: begin //ADD
```

```
            alu_fun = 4'b0000; //0000
        end

        1'b1: begin //SUB
            alu_fun = 4'b1000;
        end
    endcase
    end

3'b001: begin //SLL
    alu_fun = 4'b0001;
end

3'b010: begin   //SLT
    alu_fun = 4'b0010;
end

3'b011: begin   //SLTU
    alu_fun  = 4'b0011;
end

3'b100: begin //XOR
    alu_fun = 4'b0100;
end

3'b101: begin //SRL or SRA
    case(FUNC7)
        1'b0: begin //SRL
            alu_fun = 4'b0101;
        end

        1'b1: begin //SRA
            alu_fun = 4'b1101;
        end
    endcase
    end

3'b110: begin //OR
    alu_fun = 4'b0110;
end

3'b111: begin //AND
    alu_fun = 4'b0111;

end
```

```
            endcase
        end

        OP_IMM: begin     //immediate opperations
            alu_srcA = 2'b00;
            alu_srcB = 3'b001;
            rf_wr_sel = 2'b11;
            case(FUNC3)
                3'b000: begin   // instr: ADDI
                    alu_fun = 4'b0000;

                end

                3'b010: begin  // instr:  SLTI
                    alu_fun = 4'b0010;

                end

                3'b011: begin //INSTR: SLTIU
                    alu_fun = 4'b0011;

                end

                3'b110: begin //INSTR: ORI
                    alu_fun = 4'b0110;
;
                end

                3'b100: begin //INSTR: XORI
                    alu_fun = 4'b0100;

                end

                3'b111: begin //INSTR: ANDI
                    alu_fun = 4'b0111;

                end

                3'b001: begin //INSTR: SLLI
                    alu_fun = 4'b0001;

                end


                3'b101: begin //INSTR: SRLI and SRAI
```

```
                case(FUNC7)
                SRLI: begin //SRLI
                    alu_fun = 4'b0101;
                end

                SRAI: begin //SRAI
                    alu_fun = 4'b1101;
            end

            default: begin
                pcSource = 3'b000;
                alu_fun = 4'b0000;
                alu_srcA = 2'b00;
                alu_srcB = 3'b000;
                rf_wr_sel = 2'b00;
            end
            endcase
            end

            default: begin
                pcSource = 3'b000;
                alu_fun = 4'b0000;
                alu_srcA = 2'b00;
                alu_srcB = 3'b000;
                rf_wr_sel = 2'b00;
            end

        endcase
    end

    SYS: begin  //interupt instructions
        case(FUNC3)
            3'b001: begin //CSRRW
                alu_fun = 4'b1001; //copy ?
                alu_srcA = 2'b00;
                alu_srcB = 3'b000;
                rf_wr_sel = 2'b01;
            end

            3'b011: begin //CSRRC
                alu_fun = 4'b0111; //and
                alu_srcA = 2'b10;  //inverted rs1
                alu_srcB = 3'b100; //csr rd
                rf_wr_sel = 2'b01; //alu result
            end
```

```
            3'b010: begin //CSRRS
                alu_fun = 4'b0110; //or
                alu_srcA = 2'b00;  //rs1
                alu_srcB = 3'b100; //csr rd
                rf_wr_sel = 2'b01; //alu result
            end

            3'b000: begin //mret
                pcSource = 3'b101;
            end

            default: begin
                pcSource = 3'b000;
                alu_fun = 4'b0000;
                alu_srcA = 2'b00;
                alu_srcB = 3'b000;
                rf_wr_sel = 2'b00;
            end
            endcase
            end

    BRANCH: begin
        rf_wr_sel = 2'b11;
        case(FUNC3)

            BEQ: begin   //Branch if Equal
                if (br_eq) begin
                    pcSource = 3'b010;
                end

                else begin
                    pcSource = 3'b000;
                end
                end

            BNE: begin //Branch if Not Equal
                if (~br_eq) begin
                    pcSource = 3'b010;
                end

                else begin
                    pcSource = 3'b000;
                end
                end
```

```
                    BLT: begin //Branch if Less Than
                        if (br_lt) begin
                            pcSource = 3'b010;
                        end

                        else begin
                            pcSource = 3'b000;
                        end
                        end

                    BGE: begin   //Branch is greater than or
equal
                        if (~br_lt | br_eq) begin
                            pcSource = 3'b010;
                        end

                        else begin
                            pcSource = 3'b000;
                        end
                        end

                    BLTU: begin    //Branch if less than unsigned
                        if (br_ltu) begin
                            pcSource = 3'b010;
                        end

                        else begin
                            pcSource = 3'b000;
                        end
                        end
                    BGEU: begin //Branch if greater than or equal
                        if (~br_ltu | br_eq) begin
                            pcSource = 3'b010;
                        end

                        else begin
                            pcSource = 3'b000;
                        end
                    end
            endcase
            end


                default: begin
```

```
                    pcSource = 3'b000;
                    alu_srcB = 3'b000;
                    rf_wr_sel = 2'b00;
                    alu_srcA = 2'b00;
                    alu_fun = 4'b0000;
                end
            endcase
    end
    end

endmodule
```

4.6 More Interrupts (Timer-Counter Wrapper) — Source Code

This includes the expanded wrapper connecting:

- OTTER MCU
- Timer-counter peripheral
- Memory-mapped I/O
- Seven-segment display hardware

```
//////////////////////////////////////////////////////////////////
// OTTER_Wrapper with Timer-Counter Integration
//////////////////////////////////////////////////////////////////

module OTTER_Wrapper(
   input clk,
   input [4:0] buttons,
   input [15:0] switches,
   output logic [15:0] leds,
   output logic [7:0] segs,
   output logic [3:0] an    );

   //- INPUT PORT IDS
-----------------------------------------------------------
   localparam SWITCHES_PORT_ADDR = 32'h11008000;  // 0x1100_8000
   localparam BUTTONS_PORT_ADDR  = 32'h11008004;  // 0x1100_8004

   //- timer-counter input support
   localparam TMR_CNTR_CNT_OUT  = 32'h11008008;   // 0x1100_8004
```

```
   //- OUTPUT PORT IDS
-----------------------------------------------------------
   localparam LEDS_PORT_ADDR     = 32'h1100C000;  // 0x1100_C000
   localparam SEGS_PORT_ADDR     = 32'h1100C004;  // 0x1100_C004
   localparam ANODES_PORT_ADDR   = 32'h1100C008;  // 0x1100_C008

   //- timer-counter output support
   localparam TMR_CNTR_CSR_ADDR     = 32'h1100D000;   //
0x1100_D000
   localparam TMR_CNTR_CNT_IN_ADDR  = 32'h1100D004;   //
0x1100_D004

   //- Signals for connecting OTTER_MCU to OTTER_wrapper
   logic s_interrupt;
   logic s_reset;
   logic s_clk = 0;

   //- register for dev board output devices
---------------------------------
   logic [7:0]  r_segs;   //  register for segments (cathodes)
   logic [15:0] r_leds;   //  register for LEDs
   logic [3:0]  r_an;     //  register for display enables
(anodes)

   logic [7:0]  r_tc_csr;    // timer-counter count input
   logic [31:0] r_tc_cnt_in; // timer-counter count input


   logic [31:0] IOBUS_out;
   logic [31:0] IOBUS_in;
   logic [31:0] IOBUS_addr;
   logic IOBUS_wr;

   logic [31:0] s_tc_cnt_out;
   logic s_tc_intr;

   assign s_interrupt = buttons[4];
   assign s_reset = buttons[3];

   //- Instantiate RISC-V OTTER MCU
    RISV_V_MCU  my_otter(
```

```
      .RST         (s_reset),
      .INTR        (s_tc_intr),
      .clk         (s_clk),
      .IOBUS_IN    (IOBUS_in),
      .IOBUS_OUT   (IOBUS_out),
      .IOBUS_ADDR  (IOBUS_addr),
      .IOBUS_WR    (IOBUS_wr)   );

   // instantiate the timer-counter module
   timer_counter #(.n(3))  my_tc (
      .clk        (s_clk),
      .tc_cnt_in  (r_tc_cnt_in),
      .tc_csr     (r_tc_csr),
      .tc_intr    (s_tc_intr),
      .tc_cnt_out (s_tc_cnt_out)  );

    //- Divide clk by 2
    always_ff @ (posedge clk)
         s_clk <= ~s_clk;

    //- Drive dev board output devices with registers
    always_ff @ (posedge s_clk)
    begin
       if (IOBUS_wr == 1)
       begin
          case(IOBUS_addr)
             LEDS_PORT_ADDR:       r_leds <= IOBUS_out[15:0];
             SEGS_PORT_ADDR:       r_segs <= IOBUS_out[7:0];
             ANODES_PORT_ADDR:     r_an  <= IOBUS_out[3:0];
             TMR_CNTR_CSR_ADDR:    r_tc_csr  <= IOBUS_out[7:0];
             TMR_CNTR_CNT_IN_ADDR: r_tc_cnt_in <=
IOBUS_out[31:0];
             default:    r_leds <= 0;
          endcase
       end
    end

    //- MUX to route input devices to I/O Bus
     //-   IOBUS_addr is the select signal to the MUX
     always_comb
    begin
```

```
        IOBUS_in=32'b0;
        case(IOBUS_addr)
           SWITCHES_PORT_ADDR: IOBUS_in[15:0] = switches;
           BUTTONS_PORT_ADDR:  IOBUS_in[4:0]  = buttons;
           TMR_CNTR_CNT_OUT:   IOBUS_in[31:0] = s_tc_cnt_out;
           default: IOBUS_in=32'b0;
        endcase
    end

     //- assign registered outputs to actual outputs
     assign leds = r_leds;
     assign segs = r_segs;
     assign an = r_an;


endmodule
```

5. Demo and Example Programs

This section collects all firmware-level RISC-V assembly programs written during the Otter MCU development. They demonstrate interrupts, display multiplexing, averaging logic, debouncing, and general I/O processing.

All code blocks are copy/paste-friendly and preserve indentation.

5.1 Demo Program — LED Blinker and Pattern Generator (from Complete ISA MCU)

```
li x10, 0x11008004      # Buttons
li x11, 0x1100C000      # LED output
li x12, 0x1             # Start with rightmost LED on
li x8, 0x1              # Display LED pattern
li x13, 0x1
```

```
main:
    li x7, 16           # Count for display pattern 2
    li x9, 16           # Count for display pattern 1

blink:
    sw x12, 0(x11)      # Turn LED on
    call delay_ff
    sw x0, 0(x11)       # Turn LED off
    call delay_ff
    lw x14, 0(x10)      # Read button
    andi x14, x14, 1
    beq x14, x0, blink  # No button = keep blinking

display_1:
    beq x9, x0, display_2
    sw x8, 0(x11)
    slli x8, x8, 1      # Shift pattern left
    addi x9, x9, -1
    call delay_ff
    j display_1

display_2:
    beq x7, x0, main
    sw x8, 0(x11)
    srli x8, x8, 1      # Shift pattern right
    addi x7, x7, -1
    call delay_ff
    j display_2

# Delay subroutine
delay_ff:
    li x31, 0x3FFFF
loop:
    beq x31, x0, done
    addi x31, x31, -1
    j loop
done:
```

```
    ret
```

5.2 Example Program — Interrupt-Based Value Averager (from Lab 6)

```
# Averages the 4 most recent switch values on each interrupt.
# Shows result on LEDs.

avg_maker:

Init:
    li x9, 0x11008000    # Switch port
    li x11, 0x1100C000   # LED port
    la x6, ISR
    csrrw x0, mtvec, x6  # Set ISR address

    mv x12, x0
    mv x13, x0
    mv x14, x0
    mv x15, x0

begin:
    mv x8, x0
    li x10, 0x8
    csrrs x0, mstatus, x10  # Enable interrupts

loop:
    nop
    beq x8, x0, loop        # Wait for interrupt flag

    lhu x10,0(x9)           # Read switch value
    mv x15,x14
    mv x14,x13
    mv x13,x12
    mv x12,x10
```

```
    add x10, x10, x12
    add x10, x10, x13
    add x10, x10, x14
    add x10, x10, x15

    srli x10, x10, 2        # Divide by 4
    sh x10, 0(x11)          # Output to LEDs

    j begin

ISR:
    li x8, 0x1              # Set flag
    li x10, 0x80
    csrrc x0, mstatus, x10  # Clear MPIE
    mret
```

5.3 Example Program — Interrupt-Driven LUT Averaging (16 Samples)

```
# Reads 16 values from LUT via interrupt triggers.
# Averages them and outputs LED pattern.

.data
my_lut: .byte 0x00, 0x01, 0x02, 0x03,
        0x04, 0x06, 0x07, 0x08,
        0x0C, 0x0E, 0x0F, 0x10,
        0x18, 0x1C, 0x1E, 0x1F

.text
main:

init:
    li x16,0x1100C000      # LED port
    li x6, ISR
```

```
    csrrw x0, mtvec, x6

    la x4,my_lut
    li x5,0x00011C000      # Input port
    li x12,0               # Accumulator
    li x20,16              # Sample count
    mv x8,x0

unmask:
    li x10,0x8
    csrrs x0,mstatus,x10

loop:
    nop
    beq x8,x0,loop

addval:
    beq x20,0,avg

    lb x11,0(x4)
    addi x4,x4,1

    add x12,x12,x11
    addi x20,x20,-1

    mv x8,x0
    csrrs x0,mstatus,x10
    j loop

avg:
    srli x12, x12, 4
    sw x12, 0(x16)

ISR:
    li x8,0x1
    li x9,0x80
    csrrc x0,mstatus,x9
    mret
```

5.4 Example Program — 7-Segment Display Scroll Using Interrupts (from Lab 6)

```
.data
mess: .byte 0x63,0x31,0x61,0xFF,0x25,0x0D,0x0D,
            0xFF,0x9F,0x49,0xFF,0x63,0xC5,0xC5,
            0xE3,0xFF,0x63,0x31,0x61

.text
init:
    la x12, mess
    la x23, mess
    la x15, mess
    li x13, 0x1100C004
    li x14, 0x1100C008
    la x8, ISR
    csrrw x0, mtvec, x8

    mv x16, x0
    mv x21, x0
    mv x22, x0
    li x18, 0xF
    li x19, 0xF
    li x20, 0xE

unmask:
    li x10, 0x8
    csrrs x0, mstatus, x10

loop:
    sw x18, 0(x14)
    lb x22, 0(x15)
    sw x22, 0(x13)
```

```
    addi x16,x16,1
    andi x16,x16,3

    add x15,x23,x16

    slli x19,x19,1
    andi x19,x19,0xF
    beq x19,x20,skip
    addi x19,x19,1

skip:
    sw x19,0(x14)
    j loop

ISR:
    addi x21,x21,1
    andi x21,x21,0xF
    add x23,x12,x21
    mret
```

5.5 Example Program — Zero Blink, Count Up/Down with Button Trigger (from Lab 7)

```
.data
sseg: .byte 0x03,0x9F,0x25,0x0D,0x99,
            0x49,0x41,0x1F,0x01,0x09

.text
init:
    la x9, sseg
    li x10, 0x11008004
    li x11, 0x1100C004
    li x12, 0x1100C008
    li x15, 0x7
```

```
    li x21, 0xF

reset_cnt:
    li x13,10
    li x14,10

zero_blink:
    lbu x16,0(x9)
    sw x15,0(x12)
    sw x16,0(x11)
    call delay_ff
    sw x21,0(x12)
    call delay_ff

    lw x17,0(x10)
    andi x17,x17,4
    beq x17,x0,zero_blink

turn_on_again:
    sw x15,0(x12)

count_up:
    beq x13,x0,adjust_LUT_one
    lbu x16,0(x9)
    sw x16,0(x11)
    addi x9,x9,1
    addi x13,x13,-1
    call delay_ff
    j count_up

adjust_LUT_one:
    addi x9,x9,-1

count_down:
    beq x14,x0,adjust_LUT_two
    lbu x16,0(x9)
    sw x16,0(x11)
    addi x9,x9,-1
```

```
    addi x14,x14,-1
    call delay_ff
    j count_down

adjust_LUT_two:
    addi x9,x9,1
    j reset_cnt

delay_ff:
    li x31,0x5FFFFF
loop:
    beq x31,x0,done
    addi x31,x31,-1
    j loop
done:
    ret
```

5.6 Example Program — Timer-Counter Driven Scroll with Firmware Debounce (from Lab 7)

```
.data
mess: .byte 0x63,0x31,0x61,0xFF,0x25,0x0D,0x0D,
            0xFF,0x9F,0x49,0xFF,0x63,0xC5,0xC5,
            0xE3,0xFF,0x63,0x31,0x61

.text
init:
    li x4, 0x11008004
    la x12, mess
    la x23, mess
    la x15, mess
    li x13, 0x1100C004
    li x14, 0x1100C008
```

```
    la x8, ISR
    csrrw x0, mtvec, x8

    mv x16, x0
    mv x21, x0
    mv x22, x0
    li x18, 0xF
    li x19, 0xF
    li x20, 0xE

    li x5, 0x5FF
    li x27,0x1100D000
    li x28,0x1100D004
    sw x5,0(x28)

    li x31,0x01
    sw x31,0(x27)
    mv x31,x0
    li x25,0xFF
    mv x7,x0

unmask:
    li x10,0x8
    csrrs x0,mstatus,x10

dbounce_loop:

reset:
    mv x31,x0

read_btn:
    lw x29,0(x4)
    andi x29,x29,1
    beq x29,x7,reset

db_cnt:
    addi x31,x31,1
    bne x31,x25,read_btn
```

```
    mv x7,x29
    beq x7,x0,reset
    call update_offset
    j reset

update_offset:
    addi x21,x21,1
    andi x21,x21,0xF
    add x23,x12,x21
    ret

ISR:
    sw x18,0(x14)
    lb x22,0(x15)
    sw x22,0(x13)

    addi x16,x16,1
    andi x16,x16,3

    add x15,x23,x16

    slli x19,x19,1
    andi x19,x19,0xF
    beq x19,x20,skip
    addi x19,x19,1
skip:
    sw x19,0(x14)
    mret
```