**Experiment #6**

**The Complete ISA**

**RISC-V MCU**

**CPE 233**

**Bryan Mealy**

**06/06/2022**

**Danny Dang**

**Dylan Sandall**

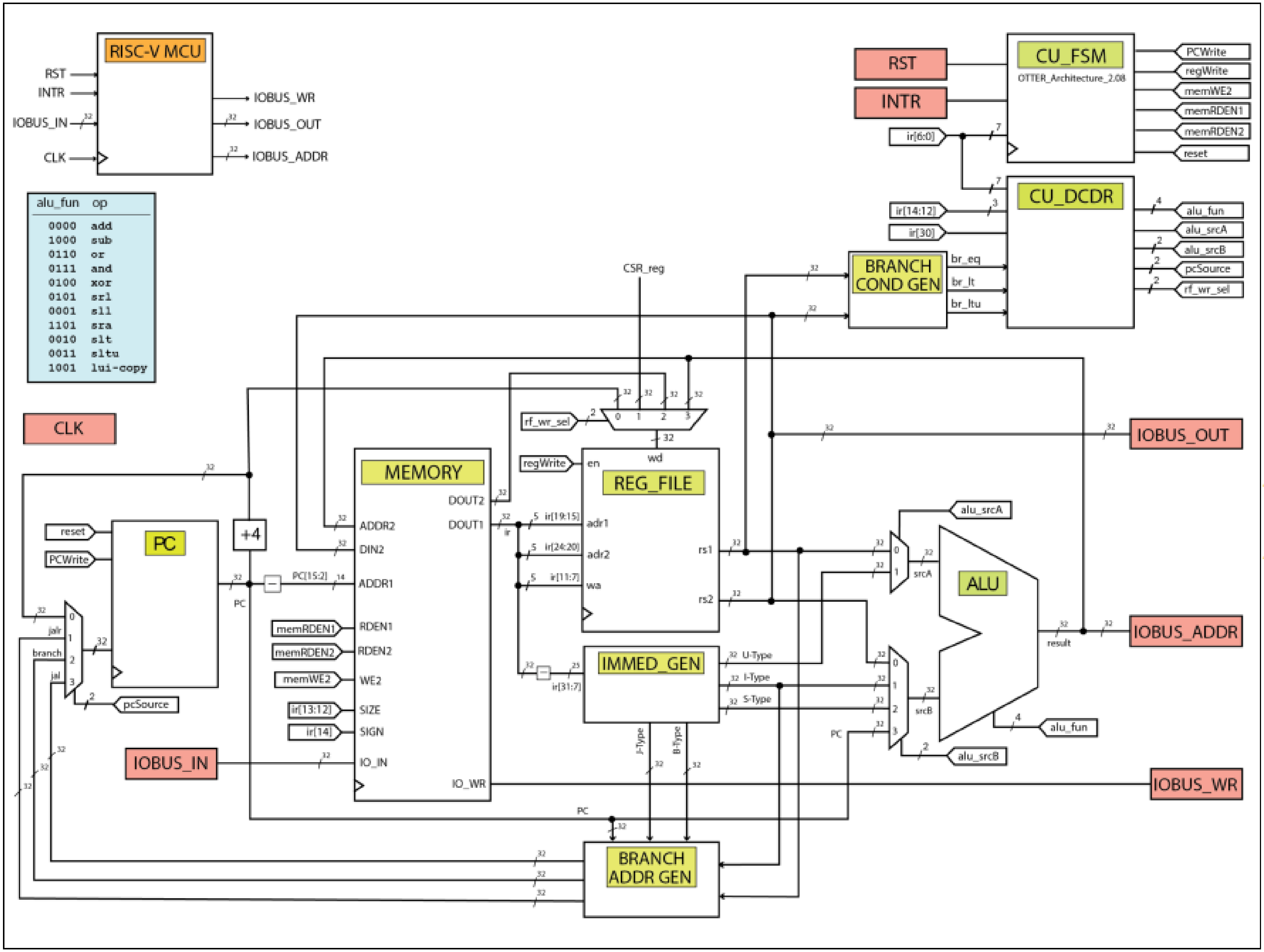
**Felix Demharter**

**Summary:**

In this experiment, we designed the BRANCH_COND_GEN and added this to our RISC-V while completing the rest of the instructions for the FSM and DCDR. We then tested if the instructions were correctly implemented by running a testall assembly code on the board that ran through 37 tests.

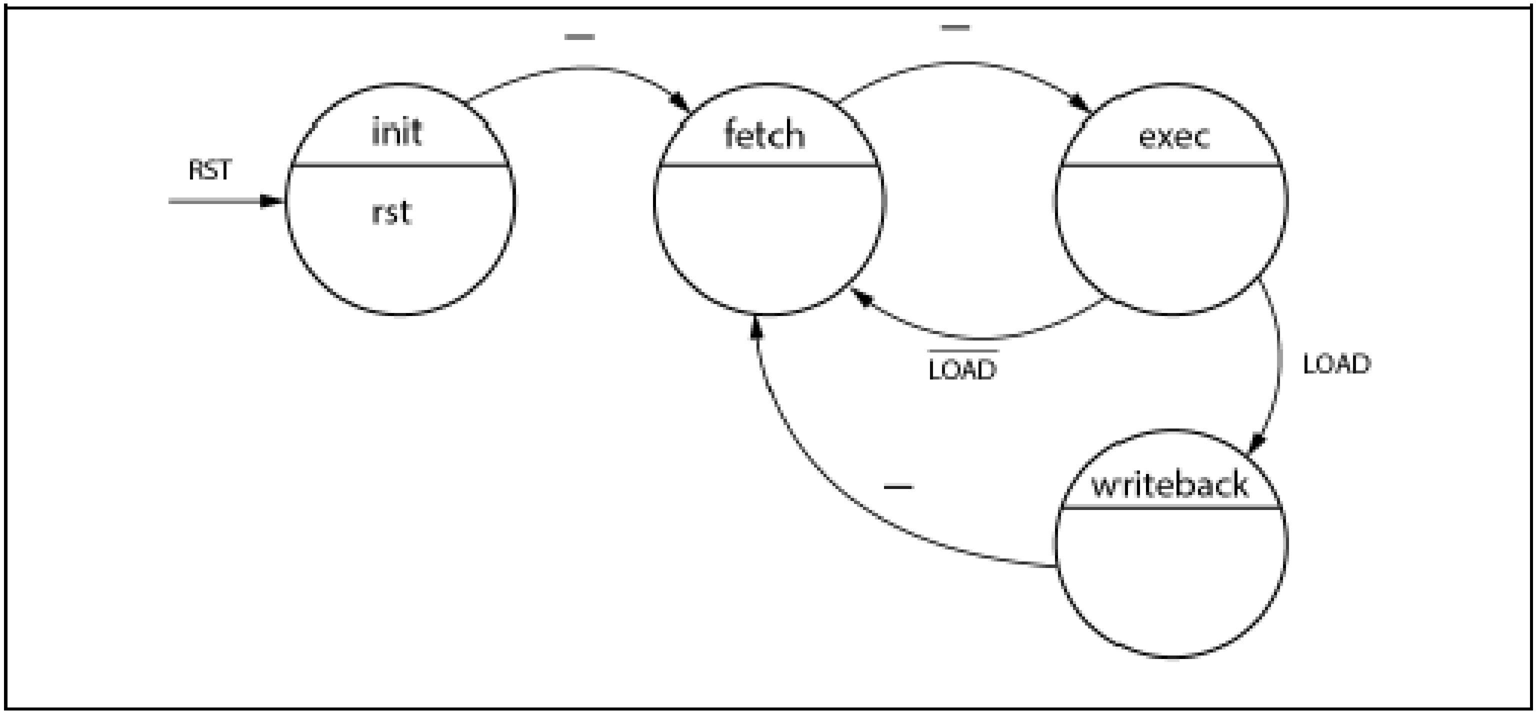
Demo: https://www.youtube.com/watch?v=4o95DDnOJqc

**Schematics:**



| alu_fun | op |
|---|---|
| 0000 | add |
| 1000 | sub |
| 0110 | or |
| 0111 | and |
| 0100 | xor |
| 0101 | srl |
| 0001 | sll |
| 1101 | sra |
| 0010 | slt |
| 0011 | sltu |
| 1001 | lui-copy |

**State Machine:**



## Source Code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Live Wire Engineering
// Engineer:  Dylan Sandall, Danny Dang, Felix Demharter
//
// Create Date: 05/14/2022
// Design Name:
// Module Name: RISC-V OTTER MCU
// Project Name: RISC-V OTTER
// Target Devices: Basys 3 Development Board
// Tool Versions:
// Description: MCU module, merges various submodules
//
// Dependencies: Memory, PC, REG_FILE, ALU, BAG, IMM_GEN, CU_DCDR, CU_FSM, Branch Condition
Generator, and intermediary hardware
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module  OTTER_MCU (
    input       rst,
    input       intr,
    input       clk,
    input       [31:0] iobus_in,
    output      [31:0] iobus_addr,
    output      [31:0] iobus_out,
    output      iobus_wr
);
    //PC wires
    wire [31:0] mux_out, PC;


    //MEMORY wires
    wire [31:0] ir, DOUT2;
    wire IO_WR;

    //REG wires
    wire [31:0] rs1, rs2, wd;

    //IMMED_GEN wires
    wire [31:0] U_type, I_type, S_type, J_type, B_type;

    //BAG wires
    wire [31:0] jalr, branch, jal;

    //ALU wires
    wire [31:0] srcA, srcB, result;

    //insert BCD wires here

    //CU_FSM wires
    wire PCWrite, regWrite, memWE2, memRDEN1, memRDEN2, reset;

    //CU_DCDR wires
    wire [3:0] alu_fun;
    wire [1:0] alu_srcB, pcSource, rf_wr_sel;
```

```
 wire alu_srcA;

 //BRANCH_COND_GEN wires
 wire br_eq, br_lt, br_ltu;

 mux_4t1_nb  #(.n(32)) PC_mux  (
     .SEL   (pcSource),
     .D0    (PC + 4),
     .D1    (jalr),
     .D2    (branch),
     .D3    (jal),
     .D_OUT (mux_out)
 );

 cntr_up_clr_nb #(.n(32)) my_PC (
     .clk   (clk),
     .clr   (reset),
     .up    (0),
     .ld    (PCWrite),
     .D     (mux_out), //input
     .count (PC), //output
     .rco   ()
 );

 Memory          my_memory(
     .MEM_CLK    (clk),
     .MEM_RDEN1  (memRDEN1),
     .MEM_RDEN2  (memRDEN2),
     .MEM_WE2    (memWE2),
     .MEM_ADDR1  (PC[15:2]),
     .MEM_ADDR2  (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  (DOUT2)
 );

 mux_4t1_nb  #(.n(32)) my_regmux(
     .SEL        (rf_wr_sel),
     .D0         (PC + 4),
     .D1         (0),        //this will be replaced with CSR_reg
     .D2         (DOUT2),
     .D3         (result),
     .D_OUT      (wd)
 );

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

IMMED_GEN        my_immedgen(
     .ir         (ir),
    .U_type     (U_type),
    .I_type     (I_type),
    .S_type     (S_type),
    .J_type     (J_type),
    .B_type     (B_type)
);

BRANCH_ADDR_GEN my_bag(
    .J_type     (J_type),
    .B_type     (B_type),
    .I_type     (I_type),
    .rs1        (rs1),
    .PC         (PC),
    .jal        (jal),
    .branch     (branch),
    .jalr       (jalr)
);

mux_2t1_nb      #(.n(32)) my_alumux_A(
    .SEL        (alu_srcA),
    .D0         (rs1),
    .D1         (U_type),
    .D_OUT      (srcA)
);

mux_4t1_nb      #(.n(32)) my_alumux_B(
    .SEL        (alu_srcB),
    .D0         (rs2),
    .D1         (I_type),
    .D2         (S_type),
    .D3         (PC),
    .D_OUT      (srcB)
);

ALU             my_alu(
    .A          (srcA),
    .B          (srcB),
    .alu_fun    (alu_fun),
    .alu_out    (result)
);

BRANCH_COND_GEN my_bcg(
    .rs1        (rs1),
    .rs2        (rs2),
    .br_eq      (br_eq),
    .br_lt      (br_lt),
    .br_ltu     (br_ltu)
);

CU_FSM          my_fsm(
    .intr       (intr),     //inputs
    .clk        (clk),
    .RST        (rst),
    .opcode     (ir[6:0]),
    .pcWrite    (PCWrite),
    .regWrite   (regWrite),
    .memWE2     (memWE2),
    .memRDEN1   (memRDEN1),
    .memRDEN2   (memRDEN2),
    .reset      (reset)
);
```

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

   assign iobus_addr = result;
   assign iobus_out = rs2;
   assign iobus_wr = IO_WR;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Danny Dang, Dylan Sandall, Felix Demharter
//
// Create Date: 05/19/2022 06:37:40 PM
// Design Name:
// Module Name: BRANCH_COND_GEN
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Module that establishes the relationship between the two register outputs with
// the DCDR
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


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

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



endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer: Danny Dang, Dylan Sandall, Felix Demharter
//
// 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
//     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
//
////////////////////////////////////////////////////////////////////////////////

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] {
       INIT,
          FETCH,
       EXECUTE,
       WRITEBACK
    }  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,
    LW   = 7'b0000011,
    SW  = 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'b1)
    begin
        PS <= INIT;
    end
    else
    begin
        PS <= NS;
    end
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)

        INIT: //waiting state
        begin
            pcWrite = 1'b0;
            regWrite = 1'b0;
            reset = 1'b1;
            memWE2 = 1'b0;
            memRDEN1 = 1'b0;
            memRDEN2 = 1'b0;
            NS = FETCH;
        end

        FETCH: //waiting state
        begin
            pcWrite = 1'b0;
            regWrite = 1'b0;
            reset = 1'b0;
            memWE2 = 1'b0;
            memRDEN1 = 1'b1;
            memRDEN2 = 1'b0;
            NS = EXECUTE;
        end

        EXECUTE: //decode + execute
        begin
```

```
pcWrite = 1'b1;
             case (OPCODE)
                 LUI:
       begin
            pcWrite = 1'b1;
            regWrite = 1'b1;
            reset = 1'b0;
            memWE2 = 1'b0;
            memRDEN1 = 1'b0;
            memRDEN2 = 1'b0;
            NS = FETCH;
       end

                    AUIPC:
                       begin
            pcWrite = 1'b1;
            regWrite = 1'b1;
            reset = 1'b0;
            memWE2 = 1'b0;
            memRDEN1 = 1'b0;
            memRDEN2 = 1'b0;
            NS = FETCH;
       end

                    BRANCH:
                       begin
            pcWrite = 1'b1;
            regWrite = 1'b0;
            reset = 1'b0;
            memWE2 = 1'b0;
            memRDEN1 = 1'b0;
            memRDEN2 = 1'b0;
            NS = FETCH;
       end

    SW:
       begin
          pcWrite = 1'b1;
          regWrite = 1'b0;
          reset = 1'b0;
          memWE2 = 1'b1;
          memRDEN1 = 1'b0;
          memRDEN2 = 1'b0;
          NS = FETCH;
       end

                    LW:
       begin
          pcWrite = 1'b0;
          regWrite = 1'b0;
          reset = 1'b0;
          memWE2 = 1'b0;
          memRDEN1 = 1'b0;
          memRDEN2 = 1'b1;
          NS = WRITEBACK;
       end

                    OP_IMM:  // addi
                       begin
          pcWrite = 1'b1;
                          regWrite = 1'b1;
          reset = 1'b0;
```

```
              memWE2 = 1'b0;
              memRDEN1 = 1'b0;
              memRDEN2 = 1'b0;
              NS = FETCH;
                           end

                        OP_RG3:
                           begin
                pcWrite = 1'b1;
                regWrite = 1'b1;
                reset = 1'b0;
                memWE2 = 1'b0;
                memRDEN1 = 1'b0;
                memRDEN2 = 1'b0;
                NS = FETCH;
           end

           JAL:
                           begin
              pcWrite = 1'b1;
                              regWrite = 1'b1;
              reset = 1'b0;
              memWE2 = 1'b0;
              memRDEN1 = 1'b0;
              memRDEN2 = 1'b0;
              NS = FETCH;
                           end

                        JALR:
                           begin
                pcWrite = 1'b1;
                regWrite = 1'b1;
                reset = 1'b0;
                memWE2 = 1'b0;
                memRDEN1 = 1'b0;
                memRDEN2 = 1'b0;
                NS = FETCH;
           end
        default:
                           begin
                              NS = FETCH;
                           end

    endcase
end

WRITEBACK:
begin
   pcWrite = 1'b1;
   regWrite = 1'b1;
   memWE2 = 1'b0;
   memRDEN1 = 1'b0;
   memRDEN2 = 1'b0;

   NS = FETCH;
end

default:
begin
    pcWrite = 1'b0;
    regWrite = 1'b0;
    reset = 1'b1;
```

```
                memWE2 = 1'b0;
                memRDEN1 = 1'b0;
                memRDEN2 = 1'b0;
                NS = FETCH;
             end

        endcase //- case statement for FSM states
    end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Ratner Surf Designs
// Engineer: Danny Dang, Dylan Sandall, Felix Demharter
//
// 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[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
// 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,
        LW   = 7'b0000011,
        SW  = 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;  alu_srcB = 2'b00;    rf_wr_sel = 2'b00;
        alu_srcA = 1'b0;   alu_fun  = 4'b0000;

        case(OPCODE)
            LUI:
            begin
                alu_fun = 4'b1001;  // LUI Copy
                alu_srcA = 1'b1;    // U type
                alu_srcB = 2'b00;   // Don't care
                pcSource = 2'b00;   // PC + 4
                rf_wr_sel = 2'b11;  // ALU output
            end

            AUIPC:
            begin
                alu_fun = 4'b0000;  // add
                alu_srcA = 1'b1;    // U type
                alu_srcB = 2'b11;   //  PC
                pcSource = 2'b00;   // PC + 4
                rf_wr_sel = 2'b11;  // ALU result
            end

            JAL:
            begin
                alu_fun = 4'b0000;  // Don't care
                alu_srcA = 1'b0;    // rs1
                alu_srcB = 2'b00;   // rs2
                pcSource = 2'b11;   // PC Mux for JAL
                rf_wr_sel = 2'b00;  // Memory out (DOUT1)
            end

            JALR:
            begin
                alu_fun = 4'b0000;  // Don't care
                alu_srcA = 1'b0;    // Don't care
                alu_srcB = 2'b00;   // Dont' care
                pcSource = 2'b01;   // JALR
                rf_wr_sel = 2'b00;  // PC + 4
```

```
 end

 BRANCH:
 begin
     alu_fun = 4'b0000;   // Don't care
     alu_srcA = 1'b0;     // Don't care
     alu_srcB = 2'b00;    // Don't care
     rf_wr_sel = 2'b00;   // Don't care
     if ((func3 == 3'b000) && (br_eq == 1))
         pcSource = 2'b10;
     else if ((func3 == 3'b001) && (br_eq == 0))
         pcSource = 2'b10;
     else if ((func3 == 3'b100) && (br_lt == 1))
         pcSource = 2'b10;
     else if((func3 == 3'b101) && ((br_lt == 0) || (br_eq == 1)))
         pcSource = 2'b10;
     else if((func3 == 3'b110) && (br_ltu == 1))
         pcSource = 2'b10;
     else if((func3 == 3'b111) && ((br_ltu == 0) || (br_eq == 1)))
         pcSource = 2'b10;
     else
         pcSource = 2'b00;
 end

 LW:
 begin
     alu_fun = 4'b0000;   // add
     alu_srcA = 1'b0;     // rs1
     alu_srcB = 2'b01;    // I-Type
     pcSource = 2'b00;    // PC + 4
     rf_wr_sel = 2'b10;   // MemoryOut2 (DOUT2)
 end

 SW:
 begin
     alu_fun = 4'b0000;   // Don't care
     alu_srcA = 1'b0;     // rs1
     alu_srcB = 2'b10;    // S-Type
     pcSource = 2'b00;    // PC + 4
     rf_wr_sel = 2'b00;   // PC_OUT + 4
 end

OP_IMM: //addi
begin
     if (func3 == 3'b101) // The shift right condition
         alu_fun = {func7, func3};
     else                 // All other conditions
         alu_fun = {1'b0, func3};
     alu_srcA = 1'b0;     // rs1
     alu_srcB = 2'b01;    // I-type
     pcSource = 2'b00;    // PC + 4
     rf_wr_sel = 2'b11;   // ALU output
end

 OP_RG3:
 begin
     alu_fun = {func7, func3};
     alu_srcA = 1'b0;    // rs1
     alu_srcB = 2'b00;   // rs2
     pcSource = 2'b00;   // PC + 4
     rf_wr_sel = 2'b11;  // ALU output
 end
```

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

endmodule
```

**Questions:**

1. Briefly describe how the MCU differentiates between load/store and Input/Output instructions. For this problem, we’re not talking about the opcodes associated with those instructions.
One way the MCU differentiates a load/store from an input/output instruction is the address to be accessed. IO addresses use 0x0001_0000 and higher addresses, but load/store addresses are below this.

2. Briefly but completely describe why the load-type instructions (lb, lbu, lh, lhu, and lw) require three cycles to execute.
Load type instructions make use of three cycles because the ALU must be used to generate the address to be loaded from. Fetch retrieves the value within the source register, Execute adds the immediate to this value, and Writeback does the actual loading from this value (which represents an address).

3. What is the maximum number of different unique bits that you could configure the RISC-V MCU to input? Briefly but fully explain. Write an equation; don’t generate the final number.
32 * ( FFFF_FFFF - 0001_000) = 32* (2^32 - 2^16)
32 bits per address, total num of addresses minus addresses reserved for memory.

4. What is the maximum number of different unique bits that you could configure the RISC-V MCU to output? Briefly but fully explain. Write an equation; don’t generate the final number.
32 * ( FFFF_FFFF - 0001_000) = 32* (2^32 - 2^16)
32 bits per address, total num of addresses minus addresses reserved for memory.
This is the same number of bits as the input number, this is because all addresses used for input can also be used for output, and the bit width of each address is still 32.

5. Can you use the same I/O port address for both inputs and outputs in the same complete RISC-V MCU implementation? Briefly explain.
Yes! Because the input/output instructions are different, there will be no confusion. If an address is used with a load instruction, it is assumed to be an input address, and if it is used with a store instruction, it is assumed to be an output address.

6. If the “memory” portion of RISC-V MCU memory changed from 2^16 x 8 to 2^12 x 8, what would be the new maximum number of unique input or output bits that could be addressed? Write an equation; don’t generate the final number.
32 * (2^32 - 2^12)

7. Briefly describe whether the FSM used in the RISC-V OTTER MCU was a Mealy or a Moore-type FSM.
The FSM used in the RV MCU is a Moore-type, because the states have outputs that depend purely on the current state. If the outputs within each state were controlled by a variable other than the state itself, it would be Mealy. If you combined the FSM and DCDR, you could consider the new module to be a Mealy type FSM.

8. 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 they always block. Briefly state why this approach represents excellent HDL coding style.

One advantage to defining each variable within each state is that it makes the code much easier to read - all values are directly written in the state.

9. How much memory do the control units in this experiment contain?

FSM - uses registers, and the number of registers is dependent on how many states the FSM has - 4 states (F, E, WB, init, and no interrupt state) - 2^2 is 4, so **2 registers.**
DCDR - the decoder is combinatorial, and has no memory.

10. Which signal(s) in the control units represent the memory elements for the control unit.

The only memory elements in the control unit are for recording the current state in the FSM - there are no control/status lines directly tied to the state, but the outputs of the FSM are determined by the state (PCwrite, regWrite, memWE2, memRDEN1, memRDEN2, and reset).

11. Briefly state why the nop in the RISC-V ISA is a pseudoinstruction and not a base instruction.

There’s not really much reason to encode a NOP base instruction if it can be accomplished using pseudoinstructions. You might want a NOP base instruction for efficiency (no need for fetch or execute with a NOP type instruction), but considering the NOP instruction is intended for use as a delay, that could actually make NOP worse.

12. 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 MCU is a pseudoinstruction. Show at least four different ways you can implement a nop instruction in RISC-V assembly language.

1. mv x0, x1
2. addi x10, x10, 0
3. add x10, x10, x0
4. andi x10, x10, FFFF_FFFF
5. ori x10, x10, 0000_0000
6. neg x12, x12
   neg x12, x12

13. My particular application requires 45 stacks. Would this be possible using the current RISC-V MCU? If so, briefly but completely explain how this can be done using the ISA and not changing hardware. Assume the amount of memory is not an issue for this problem.

Maybe this could be done in some clever way, but the biggest obstacle is a lack of registers to keep up with all 45 stacks. Each stack would need a stack pointer, and there are only 32 (31 usable) registers.

**Programming Assignment:**

Write a RISC-V program that produces a heartbeat-type output on the LED display. For this problem one LED turns on and bounces back and forth on the eight lower LEDs on the development board at about a 1-2Hz rate. LEDs are either all on or all off, but you can simulate various levels of LED intensity by adjusting the percentage of the time the LED is on. The notion of a heartbeat LED means that it slowly goes on then slowly goes off. This is an open-ended problem; you main task is to write this program and show that it works on the Basys3 board. Make it look good to perfect. Note that there is a demonstration of a completed project on Canvas. Also, for this problem, include a flowchart describing your algorithm, and a complete written description of the algorithm you used in your solution. Also, for this problem, you must use the wrapper associated with Experiment 5, not the one you used in this Experiment.

**Assembly Code:**

```
# ---------------------------------------------------
# SUBROUTINE: heartbeat pong - exp 6
# LED           - x5  - contains LED address 0x110000F0
# CurrLED       - x7  - contains current LED to be lit, as a byte
# LED8          - x8  - 1000 0000
# LED1          - x9  - 0000 0001
# UP            - x10- up or down, for direction heartbeat is moving
# TIMERlen      - x11- contains length of timer
# LEDSTATE      - x13- contains LED state
# TIME          - x20- counter, for duty cycle
# DUTY          - x30- duty cycle of LED, out of TIMERlen
# DUTYlim       - x31- checks if duty cycle is done
# ----------------------------------------------------
Lab6_heartbeat:
init:           li   x5,  285212912       # x5 = led addr
                li   x7,  2               # Current LED = 0000 0010
                li   x8,  128             # x8 = 1000 0000
                li   x9,  1               # x9 = 0000 0001
                mv   x10, x0              # UP = 0
                li   x11, 2047            # value for timer length

loop:           beq  x7, x8, flip         # flip if x7 = 10000000
                beq  x7, x9, flip         # flip if x7 = 00000001
                j    shift
flip:           not  x10, x10             # flip UP (0000 or 1111)
shift:          call shift_led            # call shift_led routine

initon:         mv   x30, x0              # x30 = DUTY = 0
                mv   x31, x11             # DUTYlim = TIMERlength
on:             call Timer_heartbeat      # call timer routine
```

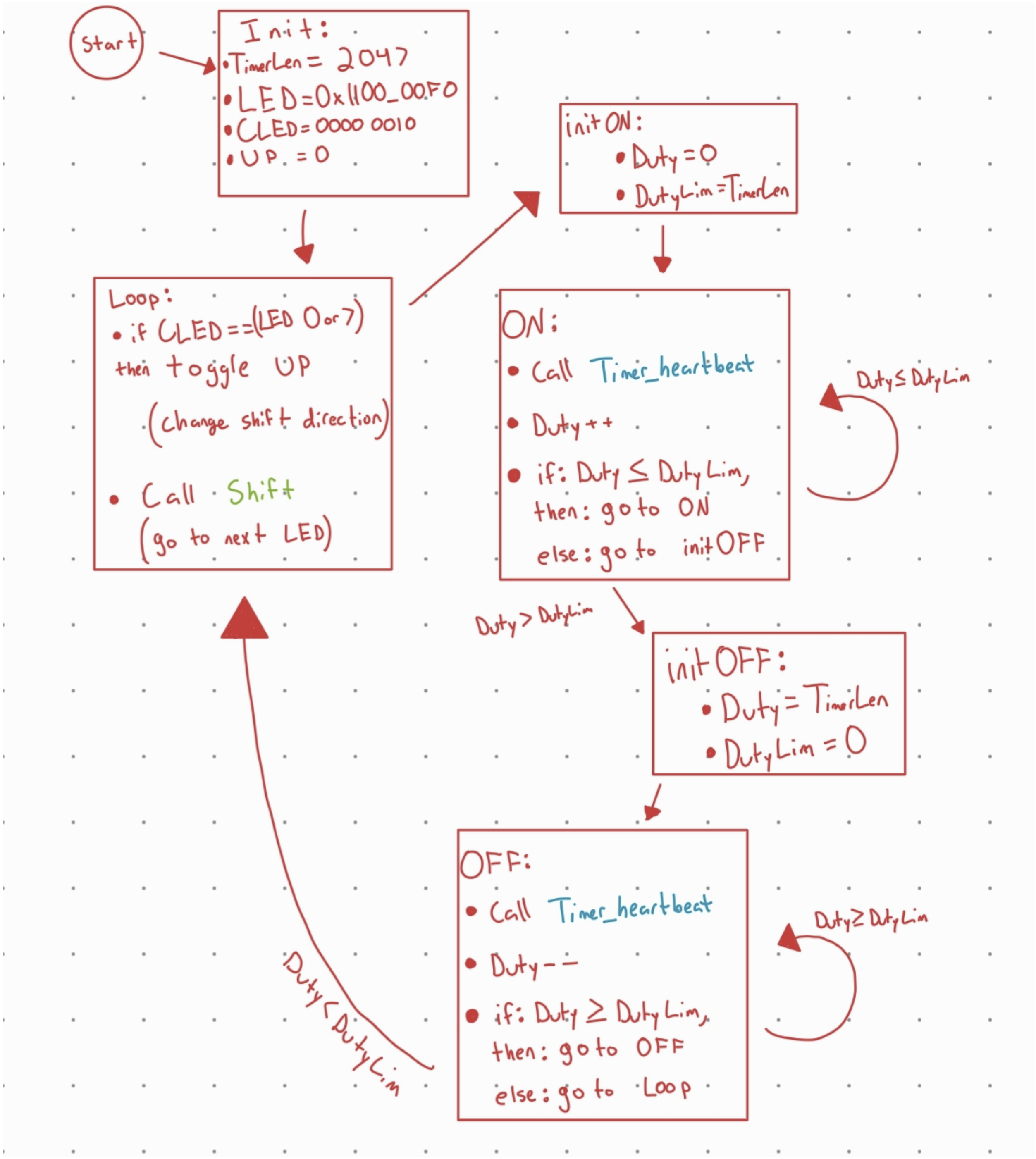
```
                addi x30, x30, 1           # increment DUTY level
                ble  x30, x31, on          # loop if DUTY within lim

initoff:        mv   x30, x11              # x30 = DUTY = TIMERlength
                mv   x31, x0               # x31 = DUTYlim = 0
off:            call Timer_heartbeat       # call timer routine
                addi x30, x30, -1          # decrement DUTY level
                bge  x30, x31, off         # loop if DUTY within lim

                j    loop                  # repeat main loop

# ----------------------------------------------------
# SUBROUTINE: Timer V2 - exp 6
# LED           - x5  - contains LED address
# TIMERlen      - x11- contains length of timer
# LEDSTATE      - x13- contains LED state
# TIME          - x20- counter, for duty cycle
# DUTY          - x30- duty cycle of LED, out of TIMERlen
# -----------------------------------------------------
Timer_heartbeat:
                mv   x13, x0               # LEDSTATE = off
                mv   x20, x11              # time = TIMERlength
                sw   x13, 0(x5)            # init LEDs as off

timeloop:       beq  x30, x20, timeon      # branch if duty = time
                j    timead                # otherwise jump to admin
timeon:         xor  x13, x13, x7          # enable current LED
                sw   x13, 0(x5)            # output LEDSTATE to LEDs
timead:         addi x20, x20, -1          # decrement time
                bnez x20, timeloop         # repeat until time = 0

                ret

# ----------------------------------------------------
# SUBROUTINE: shift_led - exp 6
# CurrLED       - x7  - contains current LED to be lit, as a byte
# UP            - x10- up or down, for direction heartbeat is moving
# -----------------------------------------------------
shift_led:      beq  x10, x0, shiftr       # if UP = 0, shift right
shiftl:         slli x7, x7, 1             # else, shift left
                j    shiftad
shiftr:         srli x7, x7, 1             # shift current LED right
shiftad:        ret
```

**Written Description and Demo:**

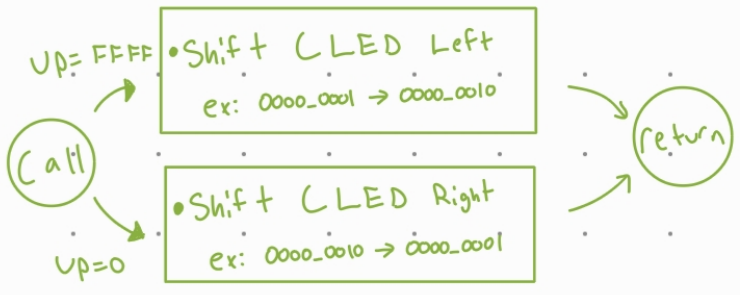
**Demonstration Video**

The above assembly code is designed to fade an LED on, fade it off, then go to the next LED in an 8 LED group. The program works by first initializing values (we changed the LED address in the wrapper as well as the assembly code), then calling the main loop. The main loop handles the movement of the LED along the board. It does this by determining the direction the LED needs to move, and then calling shift. The shift subroutine moves a single 1 in a byte around, representing the current LED. After shift has run, values are initialized for the rising fade of the LED. It starts with a Duty value of 0 and a DutyLim value of 2047. Timer_heartbeat is now called, and will loop 2048 times, with the LED on for 0 out of 2048 times. Timer_heartbeat finishes, Duty is incremented, and Timer_heartbeat is called again. This will happen 2048 times, with the brightness (Duty out of DutyLim) of the LED increasing from 0 to 100 percent. Great, now the LED is on! To fade out the LED, the same process is used, but with slightly different logic. The Duty instead starts at max, and the limit at 0. Timer_heartbeat is called, Duty is now *decremented*, and this repeats until Duty reaches 0. Once this happens, return to the main loop, and shift/fade your heart out.

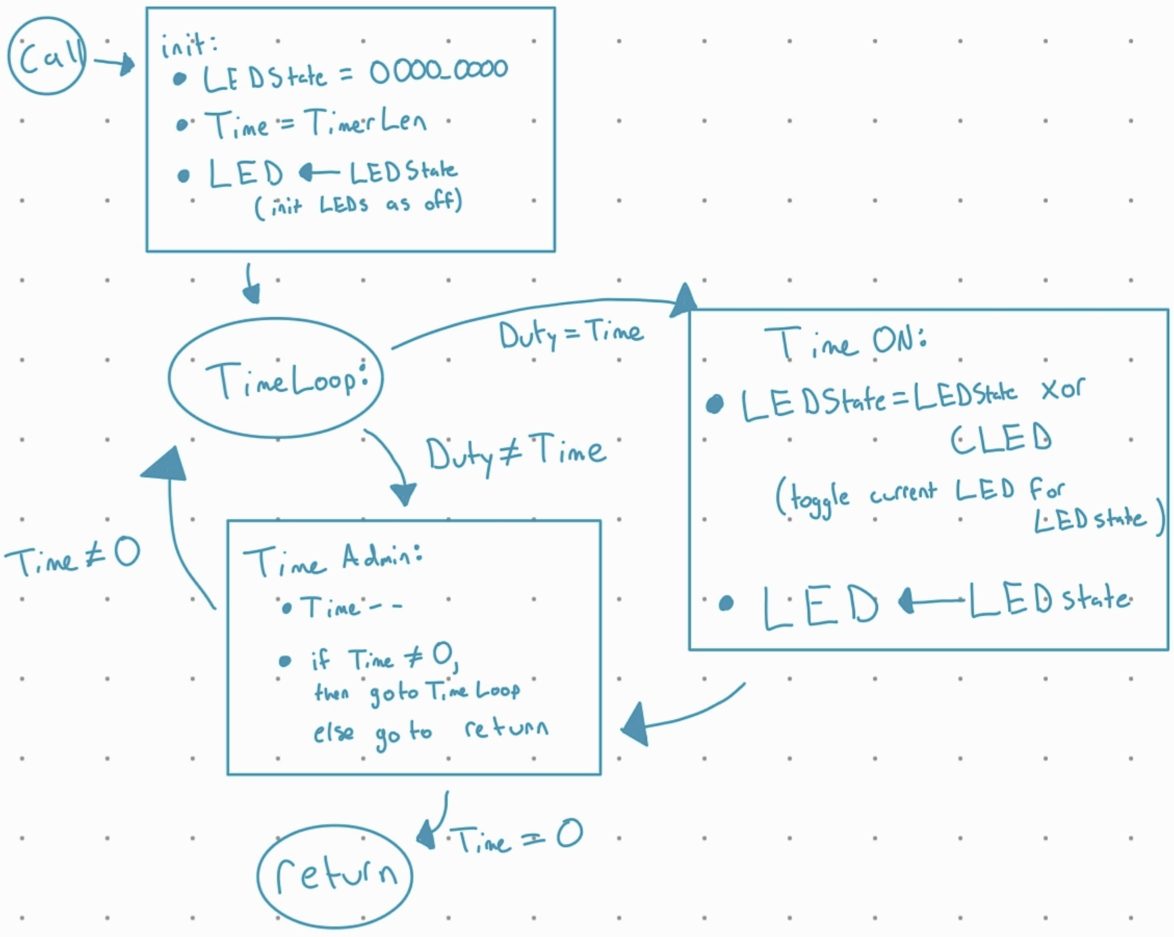
**Accompanying Flowcharts:**



## Shift



## Timer_heartbeat

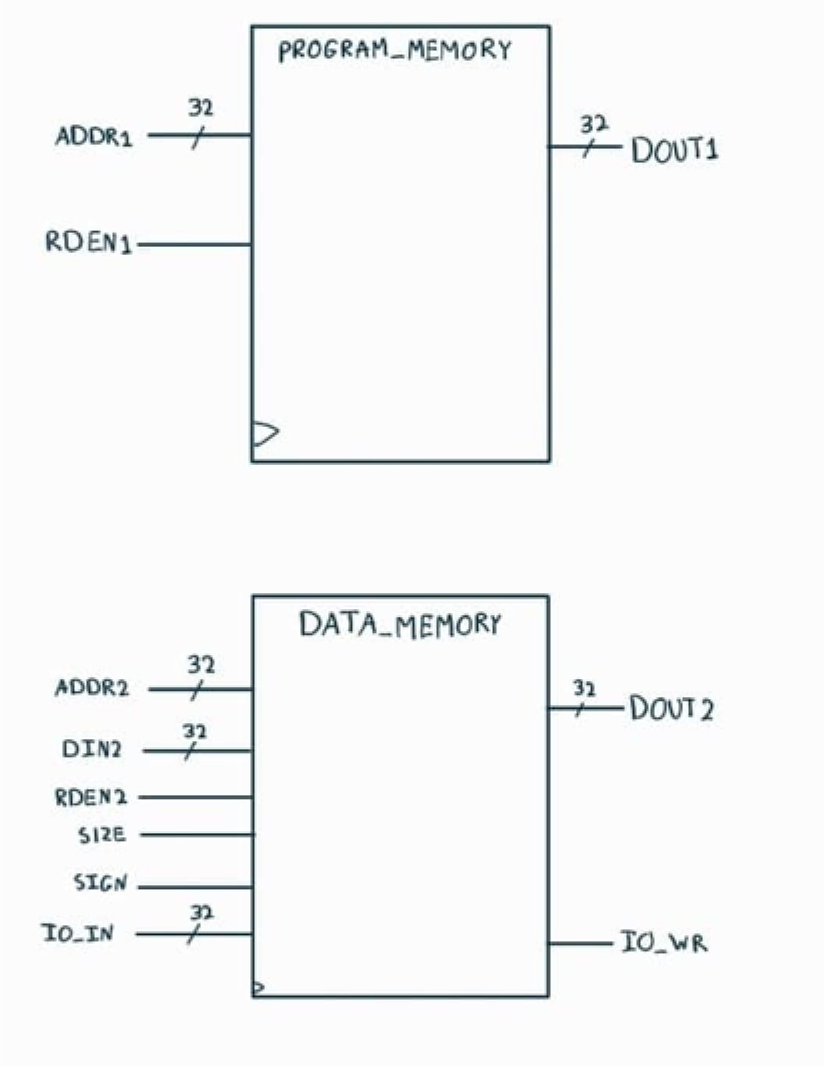


**Hardware Design Assignment:**

The RISC-V MCU memory uses only one block of memory to include both program memory and data memory. Describe the changes to hardware that you would need to make to separate data and program memory. Be sure to show a diagram for your solution. Also, briefly describe what advantage there would be to separate the memories. For this problem, use the RISC-V OTTER memory as the memory to separate, which means you must account for all the memory's inputs and outputs.

**Answer:**

This is actually easier than it sounds, thanks to the fact that the Memory module's control and status lines are already split between program and data with the 1 / 2 convention. You would need 2 single access memory modules, one for program and one for data. Connected to the new program memory would be ADDR1, RDEN1, DOUT1, and clock. Connected to the data memory would be ADDR2, DIN2, RDEN2, SIZE, SIGN, IO_IN, IO_WR, and DOUT2 (and clock). Advantages: Program memory could be made writable.


PROGRAM_MEMORY
ADDR1
32
RDEN1
32
DOUT1
DATA_MEMORY
ADDR2
32
DIN2
32
RDEN2
SIZE
SIGN
IO_IN
32
32
DOUT2
IO_WR