Justin Carlson

2/22/23

CPE 233-07

Prof. Mealy

Experiment 6 Submission

Executive Summary:

This experiment involved implementing nearly the entire RISC-V instruction set for the MCU. Instructions were added by modifying the FSM and decoder files to include cases for values of func3, func7, and the opcodes.

Experiment Demo: https://youtu.be/tljQosNzo3w

Questions:

1. Briefly explain whether the use of enumerated types in SystemVerilog increases the size of the synthesized design.

Looking through the synthesized design, there are no indicators for the enumerated types anywhere in the diagram which leads me to believe it doesn't affect the size. My guess would be that it's read during synthesis in order to compile the program, and when Vivado builds the design it uses the plain values of these types to create the board.

2. 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.

The address in the memory tells the MCU whether it belongs in the memory space or the IO space. Memory occupies the address range 0x0 to 0xFFFF while IO occupies the space between 0x10000 and 0xFFFFFFFF.

3. Briefly but completely describe why the load-type instructions (lb, lbu, lh, lhu, and lw) require three cycles to execute.

The fetch cycles behaves like all other instructions as normal, but the load instructions have different timings. The execute state for the load instructions are only capable of enabling the memory read on that t-cycle. A simultaneous read + write cannot be done on the same execute cycle, so for load instructions we have a writeback state after the execute that actually writes the memory value into a register.

4. 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.

Since the IO space of the MCU falls into the range in memory between 0x00010000 and 0xFFFFFFFF, the maximum of addressable unique bits would be $(2^{32} – 2^{16})$ * 32 bits.

5. 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.

As the MCU IO space is shared for inputs and outputs, the number of output bits will be the same as the number of input bits, or $(2^{32} – 2^{16})$ * 32 bits.

6. Can you use the same I/O port address for both inputs and outputs in the same complete RISC-V MCU implementation? Briefly explain.

No, using the same address for inputs and outputs would cause you to overwrite the data at that memory location. Different ports must be chosen for all IO in your MCU.

7. If the "memory" portion of RISC-V MCU memory changed from 2^16 x 8 to 2^10 x 8, what would be the new maximum number of unique input and/or output addresses that the hardware could use? Answer this problem with an equation. Read the problem very carefully.

Since the number of IO addresses is the total memory capacity minus the space dedicated for memory, the number of unique addresses is $(2^{32} – 2^{10})$.

8. 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. addi x0, x0, 0
2. jal x0, next

next: (next instruction)

3. beqz x0, next

next: (next instruction)

4. mv x0, x0

9. Briefly describe whether the FSM used in the RISC-V OTTER MCU is a Mealy or a Moore-type FSM.

The FSM is a Moore type, as none of the inputs directly trigger any specific FSM outputs. This FSM is only sensitive to changes in the reset and opcode inputs, and no Mealy outputs are present in any of their respective state changes.

10. Briefly explain what attribute of the RISC-V MCU is preventing you from executing all 2-cycle instructions in one clock cycle.

Program memory reads are synchronous, so one t-cycle is required to actually read the memory then another is needed to actually execute it.

11. Can I possibly use the RISC-V to implement a program that utilizes 40 stacks? 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.

Memory segments can be allocated for each stack like in the normal MCU, where we break our data segment for example into a collection of stacks. We can use the set of registers as stack pointers, if we're willing to fit multiple stack pointers into each register since their address will be small enough.

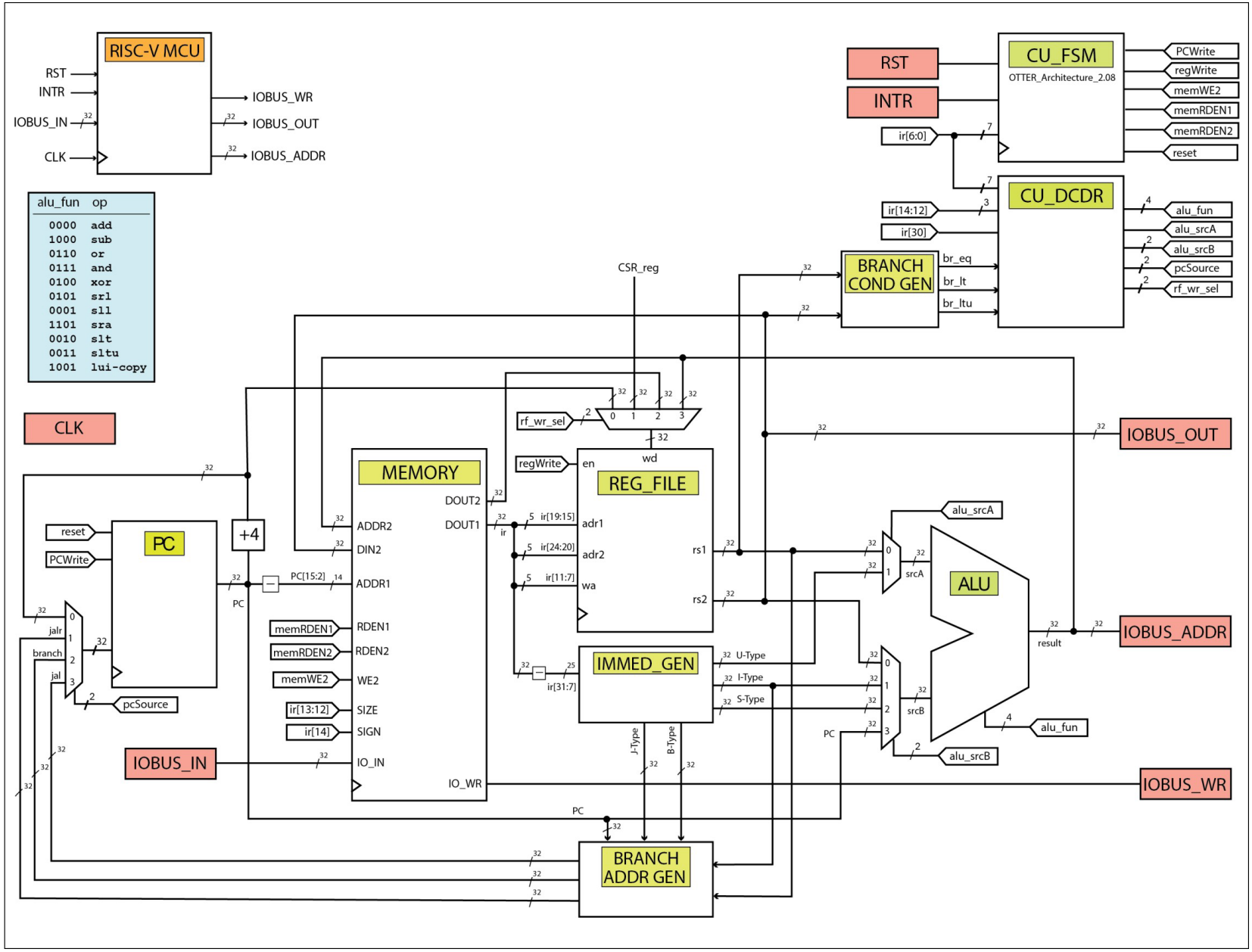
Otherwise, we can also put the stack pointers into a known place in memory and load/store them when we need to access each stack and operate on it.

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

The memory of the control units is determined by the size of the state registers, and since we have 4 states, our CU memory can be fit into 2 bits.

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

The signal in charge representing the memory elements in the CU is the state that the PS variable is assigned to.


RISC-V MCU
RST
INTR
IOBUS_IN
CLK
IOBUS_WR
IOBUS_OUT
IOBUS_ADDR
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
RST
INTR
CU_FSM
OTTER_Architecture_2.08
PCWrite
regWrite
memWE2
memRDEN1
memRDEN2
reset
ir[6:0]
CU_DCDR
ir[14:12]
ir[30]
alu_fun
alu_srcA
alu_srcB
pcSource
rf_wr_sel
BRANCH COND GEN
br_eq
br_lt
br_ltu
CSR_reg
CLK
IOBUS_OUT
MEMORY
REG_FILE
regWrite
rf_wr_sel
wd
DOUT2
DOUT1
ADDR2
DIN2
ADDR1
RDEN1
RDEN2
WE2
SIZE
SIGN
IO_IN
IO_WR
adr1
adr2
wa
rs1
rs2
ir
ir[19:15]
ir[24:20]
ir[11:7]
reset
PCWrite
PC
+4
PC[15:2]
memRDEN1
memRDEN2
memWE2
ir[13:12]
ir[14]
IOBUS_IN
jalr
branch
jal
pcSource
IMMED_GEN
ir[31:7]
U-Type
I-Type
S-Type
J-Type
B-Type
alu_srcA
srcA
ALU
srcB
alu_srcB
alu_fun
result
IOBUS_ADDR
IOBUS_WR
BRANCH ADDR GEN

Verilog Code:

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: Modified by Justin Carlson
//
// 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,
        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;  alu_srcB = 2'b00;    rf_wr_sel =
                                        2'b00;
                                        alu_srcA = 1'b0;   alu_fun  = 4'b0000;

                                        case(OPCODE)
                                                LUI:
                                                begin
                                                 rf_wr_sel = 2'b11;
                                                alu_srcA = 1'b1;
                                                alu_fun  = 4'b1001;
                                                end

                                                AUIPC:
                                                begin
                                                 rf_wr_sel = 2'b11;
                                                alu_srcA = 1'b1;
                                                alu_srcB = 2'b11;
                                                end

                                                JAL:
                                                begin
                                                        pcSource = 2'b11;
                                                end

                                                JALR:
                                                begin
                                                        pcSource = 2'b01;
                                                end

                                                LOAD:
                                                begin
                                                        alu_srcA = 1'b0;
                                                        alu_srcB = 2'b01;
                                                        rf_wr_sel = 2'b10;
                                                end
```

```
                                STORE:
                                begin
                                        alu_srcA = 1'b0;
                                        alu_srcB = 2'b10;
                                end

                                BRANCH: // Individual instructions labeled
          for clarity despite their glaring similarities
                                begin
                                    alu_fun = 4'b0000;
                                    alu_srcA = 1'b0;
                                     rf_wr_sel = 2'b00;
                                    alu_srcB = 2'b11;
                                    case(FUNC3)
BEQ:
begin
    if (br_eq) pcSource = 2'b10;
end

BNE:
begin
    if (~br_eq) pcSource = 2'b10;
end

BLT:
begin
    if (br_lt) pcSource = 2'b10;
end

BGE:
begin
    if (br_eq || ~br_lt) pcSource = 2'b10;
end

BLTU:
begin
    if (br_ltu) pcSource = 2'b10;
end

BGEU:
```

```
begin
    if (br_eq || ~br_ltu) pcSource = 2'b10;
end
                                    endcase
                                end

                                OP_IMM:
                                begin
                                    rf_wr_sel = 2'b11;
                                    alu_srcA = 1'b0;
                                    alu_srcB = 2'b01;
                                       case(FUNC3)
                                               3'b000: // ADDI
                                               begin
                                                   alu_fun = 4'b0000;
                                               end

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

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

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

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

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

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

                3'b101: // SRLI & SRAI
                begin
                    case(func7)
                       1'b0: // SRLI
                       begin
                       alu_fun = 4'b0101;
                       end

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

                default:
                begin
                        pcSource = 2'b00;
                        alu_fun = 4'b0000;
                        alu_srcA = 1'b0;
                        alu_srcB = 2'b01;
                        rf_wr_sel = 2'b11;
                end
        endcase
end

OP_RG3:
begin
    alu_srcA = 1'b0;
    alu_srcB = 2'b00;
    rf_wr_sel = 2'b11;
    case(FUNC3)

        3'b000: // ADD & SUB
        begin
```

```
    case(func7)

        1'b0: // ADD
        begin
            alu_fun = 4'b0000;
        end

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

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

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

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

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

3'b101: // SRL & SRA
begin
    case(func7)

        1'b0: // SRL
        begin
            alu_fun = 4'b0101;
```

```
                end

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

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

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

        default:
               begin
                       alu_srcA = 1'b0;
                       alu_srcB = 2'b00;
                       rf_wr_sel = 2'b11;
               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
```

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
//     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] {
       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)
        if (RST == 1)
            PS <= st_INIT;
```

```
        else
            PS <= NS;

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

            st_FET: //fetch state
            begin
                memRDEN1 = 1'b1;
                memRDEN2 = 1'b0;
                NS = st_EX;
            end

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

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

```
    BRANCH:
       begin
            NS = st_FET;
       end

    LUI: // U-type
       begin
            regWrite = 1'b1;
           NS = st_FET;
       end

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

    OP_IMM: // I-type arithmetic
       begin
          regWrite = 1'b1;
          NS = st_FET;
       end

OP_RG3: // R-type instructions
   begin
      regWrite = 1'b1;
      NS = st_FET;
   end

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

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

 default:
       begin
```

```
                                                                    NS = st_FET;
                                                                end

                endcase
            end

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

            default: NS = st_FET;

        endcase //- case statement for FSM states
    end

endmodule
```

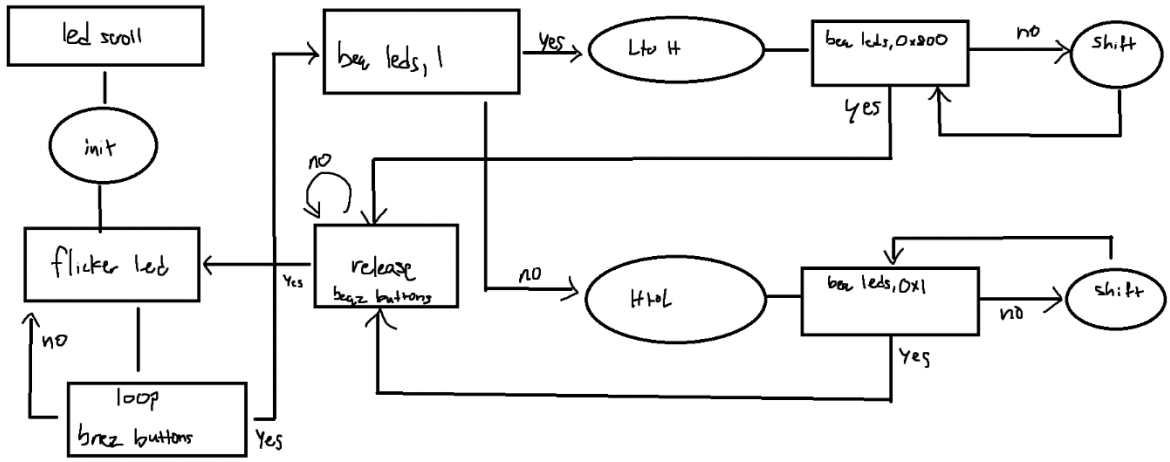
Programming Assignment:

Write a RISC-V program that implements a unique display on the development board. Upon programming the FPGA, the right-most LED on the development board will blink at about a 2Hz rate. When the user presses the left-most button on the dev board, a single LED lights as it moves from the right-most LED to the left-most LED; the left-most LED then blinks at the 2Hz rate waiting for a button release and then a button press. When the user releases the button and presses it again, the "lit" LED moves one lit LED at a time until it reaches the right-most LED. This functionality happens continuously.

```
init: li    x20,0x11008004  # button input port addr
      li    x21,0x1100C000  # LED output port addr
      li    x9, 0x8000      # constant upper value
      li    x10, 1          # value of the led
      li    x11, 1          # constant lower value

loop: sw x10, 0(x21)        # activates light
      lw x25, 0(x20)        # loads button IO
      sw x0, 0(x21)         # deactivates light
      bnez x25, pick        # checks if button pressed
      j loop

unpress: sw x10, 0(x21)     # activates light
      lw x25, 0(x20)        # loads button IO
      sw x0, 0(x21)         # deactivates light
      beqz x25, loop        # checks if button unpressed
      j unpress

pick: beq x10, x11, LtoH    # decides whether to change leds from low to high or high to low
      j HtoL

HtoL: beq x10, x11, unpress # checks if done
      srli x10, x10, 1      # shifts value to right
      sw x10, 0(x21)        # reassigns led
      j HtoL

LtoH: beq x10, x9, unpress  # checks if done
      slli x10, x10, 1      # shifts value to left
      sw x10, 0(x21)        # reassigns led
```

```
j LtoH
```

This program works by first initializing all the values and putting the program into a loop. This loop waits for a button press to be made, and the light flickers on and off until it waits. When it senses a button change, based on its current LED value, the program determines whether to shift it left or right. It then shifts it until its on the far side either to the left or right and then waits for the button to be released while flickering the LED before entering into the original loop again when the buttons are unpressed.



Demo: https://youtu.be/0FM4n2m2xKI

Hardware Assignment:

You need to the RISC-V MCU of clear alternating output values, meaning that output operations output a value, then output zero, then outputs a value, etc. on successive output operations. For this problem, describe a modification to the RISC-V schematic to implement this change and show a schematic for your added hardware. All modifications for this problem should be on the top schematic level; don't change any of the current modules in the RISC-V Otter schematic.

In order to only display a value every other time the MCU writes out, we can use a D-type flip flop with it's not output assigned to its input, D, to toggle our output. Now, IOBUS_WR acts as a clock for the flip flop, and on the rising edge it will toggle the current value in the flip flop. If we pass this output to the selector for a 2:1 mux, and have inputs of our IOBUS_OUT and ground, then our data will only display on the board for every other time the IOBUS_WR signal is set to high.