Justin Carlson

3/5/2023

CPE 233-02

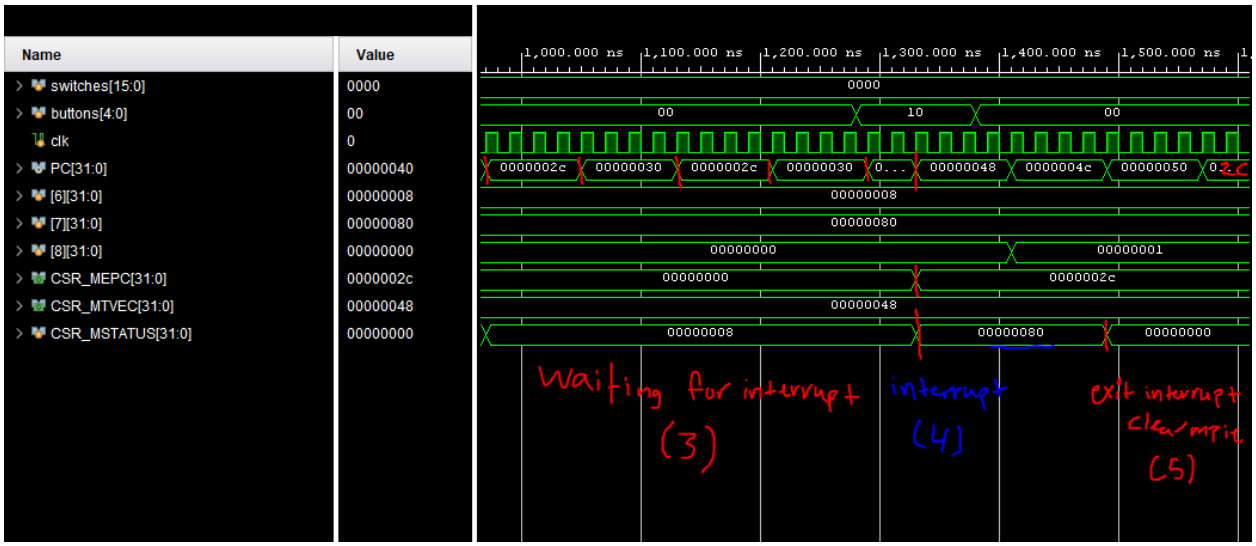
Prof. Mealy

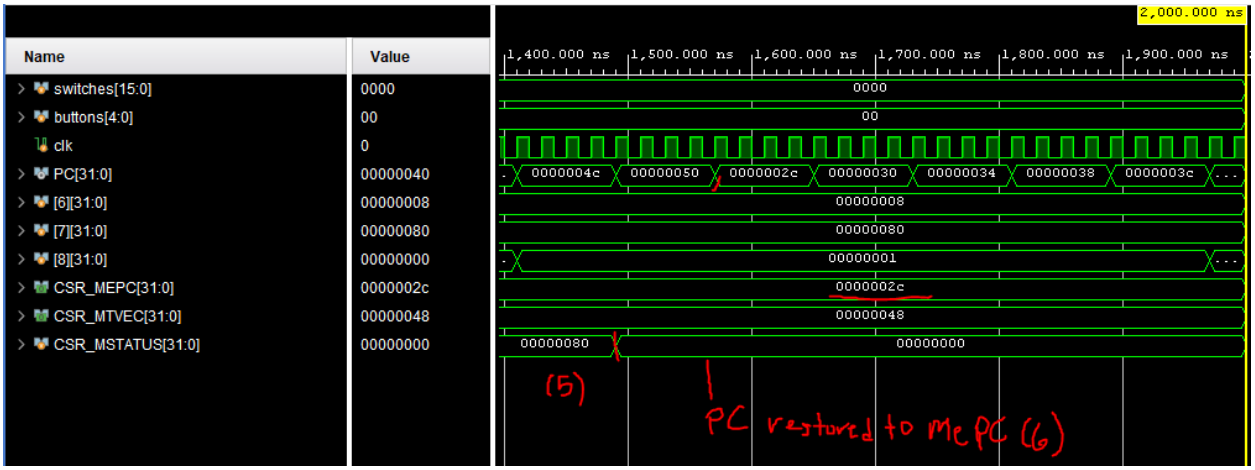
Experiment 7: MCU Interrupt Architecture

Executive Summary:

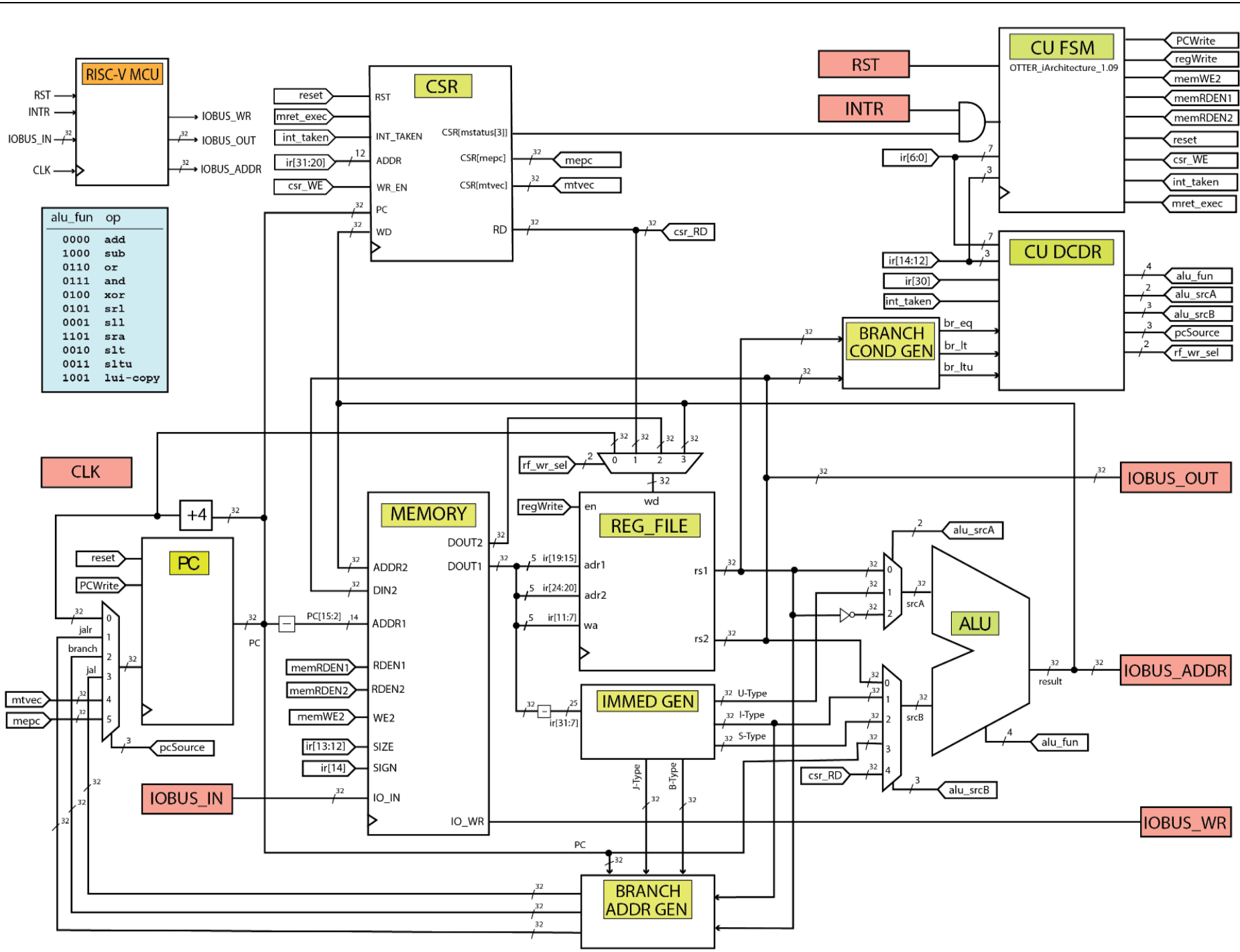
In this experiment, hardware was added to the MCU in order to support the handling of interrupts, where our MCU could enter a new subroutine at any desired time. The CSR register saves important information about our interrupt and how to return, and new instructions built into the FSM and control decoder control the interrupt functionality.

Annotated Timing Diagram:





Figures 1, 2, & 3 (Simulation Diagrams)



RISC-V OTTER MCU Architecture Diagram with Interrupts

Verilog Code:

New Interrupt OPCODE in FSM & Decoder:

```
INTERRUPT = 7'b1110011;
```

Interrupt FSM Instructions:

```
INTERRUPT:
       begin
              case(func3)
                           3'b000: // MRET instruction
                             begin
                                    mret_exec = 1'b1;
                             end

                             3'b001: // CSRRW instruction
                             begin
                                    csr_we = 1'b1;
                                    regWrite = 1'b1;
                                    if (intr)
                                    begin
                                    NS = st_INTR;
                                    end
                                    else
                                    begin
                                    NS = st_FET;
                                    end
                             end

                             3'b010: // CSRRS instruction
                             begin
                                    csr_we = 1'b1;
                                    regWrite = 1'b1;
                                    if (intr)
                                    begin
                                    NS = st_INTR;
                                    end
                                    else
                                    begin
                                    NS = st_FET;
                                    end
                             end

                             3'b011: // CSRRC instruction
                             begin
                                    csr_we = 1'b1;
                                    regWrite = 1'b1;
                                    if (intr)
                                    begin
                                    NS = st_INTR;
                                    end
                                    else
                                    begin
                                    NS = st_FET;
                                    end
                             end

                                    endcase
```

New FSM State:

```
st_INTR: // interrupt state
        begin
              pcWrite = 1'b1;
              int_taken = 1'b1;
              NS = st_FET;
        End
```

Added to non-load type opcodes:

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

Decoder Interrupt Instructions:

```
INTERRUPT:
               begin
                      case(FUNC3)
                      3'b000: // MRET instruction
                      begin
                      pcSource = 3'b101;
                      end

                      3'b001: // CSRRW instruction
                      begin
                      pcSource = 3'b000;
                      rf_wr_sel = 2'b01;
                      alu_srcA = 2'b00;
                      alu_srcB = 3'b000;
                      alu_fun = 4'b0000;
                      end

                      3'b010: // CSRRS instruction
                      begin
                      pcSource = 3'b000;
                      rf_wr_sel = 2'b01;
                      alu_srcA = 2'b00;
                      alu_srcB = 3'b100;
                      alu_fun = 4'b0110;
                      end

                      3'b011: // CSRRC instruction
                      begin
                      pcSource = 3'b000;
                      rf_wr_sel = 2'b01;
                      alu_srcA = 2'b10;
                      alu_srcB = 3'b100;
                      alu_fun = 4'b0111;
                      end

                      endcase
```

Added to end of always_comb:

```
if (int_taken)
       begin
              pcSource = 3'b100;
       end
```

Questions

1. Briefly describe how the AND gate in the RISC-V MCU schematic helps control the ability of the MCU to process interrupts.

In order for the interrupt to happen, information about the return address of the ISR must be given before the interrupt to be asserted. Once the CSR has the requisite information it needs to enter an interrupt, it triggers one side of the AND gate and waits for the user to assert the other side of the gate whenever they would like to cause the interrupt.

2. Based on the state diagram in Figure 22, estimate (in terms of clock cycles) how long the interrupt signal could be asserted but not having the MCU enter the interrupt signal. Briefly but completely explain your estimate. Assume the interrupt is unmasked.

During a load instruction that takes three clock cycles, if the interrupt signal is asserted right after the transition to the fetch state, the MCU must finish this instruction before the interrupt can be received. This means that it's possible for the signal to be asserted for nearly three clock cycles and not be seen by the MCU if it's disabled before the FSM leaves the writeback state.

3. Interrupt architectures generally always automatically mask interrupts upon receiving an interrupt. Briefly but completely describe why this is a good approach, and a better approach than attempting to rely on masking the interrupts under program control.

If the programmer has to manually disable the bit each time to prevent nested interrupts, there's a possibility that it gets forgotten once which would cause the program to break and might be hard to diagnose. If the bit is controlled by the hardware instead, the machine's not going to forget it which makes it a more dependable solution.

4. If the RISC-V Otter MCU hardware did not automatically mask the interrupt, there could be a big problem if the MCU could only mask the interrupts under program control. For this question, briefly but completely describe the problem.

When the mie bit is being cleared under program control, the cssrc instruction needs to be run in order to remove the bit. At the end of the execute cycle before the MCU clears the mie bit, the bit will still be 1 which means on the next clock edge, the MCU will see this as another interrupt and go back to the beginning of the ISR.

5. The problem described in the previous problem can actually be avoided, thus making it possible to disable interrupts under program control without compromising the overall operation of the MCU. Describe how the situation can be avoided when the MCU acts on an interrupt.

If we can guarantee that the interrupt pulse is short enough and turned off as soon as we predict the MCU reaches the ISR, then we can mask the FSM interrupts with the external input this time by unasserting the intr signal. Therefore the MCU could not start the interrupt again regardless of the state of the mie bit.

6. For the RISC-V MCU, there is only one state associated with the interrupt cycle, which means that the FSM only requires one clock cycle to complete the interrupt cycle. Briefly but completely describe in general what dictates how many states (or clock cycles) a given MCU requires for the interrupt cycle. This question considers “states” and “clock cycles” as the same thing.

Depending on how big the ISR is, the number of clock cycles needed to exit the interrupt would vary. Generally though, ISR’s should be as short as possible.

7. We generally consider interrupts “asynchronous” in nature. However, the RISC-V MCU processes everything synchronously. Briefly describe what it means for the interrupts to be asynchronous and how exactly the MCU processes them in a synchronous manner.

The interrupt signal can be asserted at any time, which exhibits asynchronous behavior. The MCU can only process the interrupt signal on a clock edge during a state change though, since it has to complete the instruction it’s currently working on before it can leave for the interrupt state.

8. Briefly but completely describe the major problem with the RISC-V MCU receiving an interrupt while the MCU is in the act of processing an interrupt. For this problem, consider the interrupts masked when the MCU receives the interrupt.

Any interrupt signals asserted while the interrupts are masked would not be recognized by the MCU, and if the interrupt was intended then the program wouldn't work as intended.

9. Briefly but completely describe the major problem with the RISC-V MCU receiving an interrupt while the MCU is in the act of processing an interrupt. For this problem, consider the interrupts unmasked when the MCU receives the interrupt.

If the MCU is running code somewhere in the ISR when the interrupt signal is asserted again, the MCU will go back to the beginning of the ISR and run some of the code which was already run, which could be disastrous for the program that's being run.

10. Word on the street is that polling is bad because it makes your programs operate "less good". My RISC-V application uses a few different polling constructs; does this make me a bad programmer? Briefly but completely explain.

No this is not necessarily the case, as using polling can sometimes be very useful and practical for a program. If at a moment, the MCU isn't doing any meaningful work and waiting for a specific input then polling is completely fine to do since nothing more important needs to be done.

11. The default action for interrupts in the RISC-V Otter MCU is that they are masked upon entry to the ISR, and later unmasked upon exiting the ISR; this means that without intervention, the interrupts returned unmasked. Briefly describe the procedure of how programmers return from interrupts with the interrupts disabled.

Since the MCU is unmasked by whatever the mpie bit is inside the mstatus register, making sure that mpie is zero by clearing that bit ensures that the interrupt cannot be asserted when the ISR is complete.

Programming Problem:

Write a RISC-V program that implements a unique display on the development board. Upon programming the FPGA, the right-most 7-segment display on the development board blinks the value of "0" at about a 2Hz rate. When the user presses the left-most button on the dev board, a display counts upward (no blinking) until it hit "9"; the value of "9" 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 display counts down until it hits "0", which it once again blinks at about an 2Hz rate as it waits for a button release and a button push. This functionality happens continuously.

• Ensure your code checks for a button release button before it waits for the next button press.

• Design your program such that the output closely matches the demonstration of a working solution to this problem listed on Canvas.

• This is an open-ended problem; your main task is to write this program and show that it works on the Basys3 board so don't get too hung up on details.

• Use the wrapper associated with Experiment 5, not the wrapper you used in this experiment.

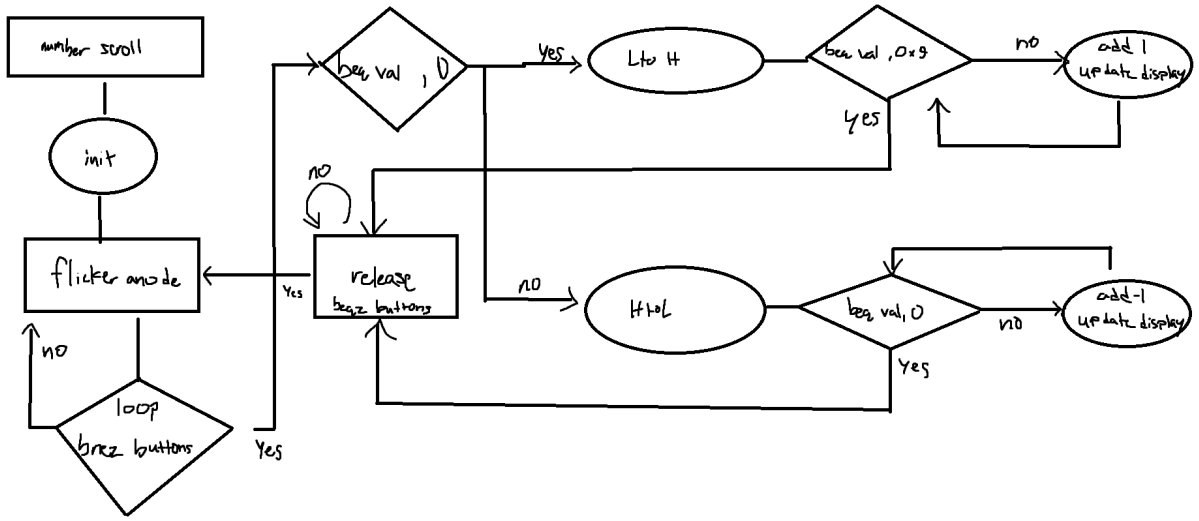
• Include a flowchart and a complete written description of the algorithm your solution uses.

Demo: https://youtu.be/Lfr9LDLZyHE

```
.data
sseg:   .byte  0x03,0x9F,0x25,0x0D,0x99,0x49,0x41,0x1F,0x01,0x09 # LUT for 7-segs

.text
init: li       x20,0x11008004  # button input port addr
      li       x21,0x1100C004  # seg output port addr
      li       x22,0x1100C008  # an output port addr
      la       x5, sseg         # constant upper value
      li       x30, 7           # anode pattern when on
      li       x31, 15          # anode pattern when off
      li       x10, 0           # value of the segs
      li       x9, 9            # constant upper value

loop: add x15, x5, x10       # find address to search in LUT
      lb  x14, 0(x15)        # load byte from LUT
      sb x14, 0(x21)         # store byte to segs
      sw x30, 0(x22)         # activates an
      lw x25, 0(x20)         # loads button IO
      bnez x25, pick         # checks if button pressed
      sw x31, 0(x22)         # deactivates an
```

```
      j loop

unpress: add x15, x5, x10   # find address to search in LUT
         lb  x14, 0(x15)     # load byte from LUT
         sb x14, 0(x21)      # store byte to segs
         sw x30, 0(x22)      # activates an
         lw x25, 0(x20)      # loads button IO
         beqz x25, loop      # checks if button unpressed
         sw x31, 0(x22)      # deactivates an
         j unpress

pick: beq x10, x0, LtoH      # decides whether to change value from low->high or high->low
       j HtoL

HtoL: beq x10, x0, unpress  # checks if done
      sw x30, 0(x22)         # activates an
      addi x10, x10, -1      # decrements value
      add x15, x5, x10       # finds new address for LUT
      lb  x14, 0(x15)        # get new value
      sb x14, 0(x21)         # reassigns seg to new byte
      j HtoL

LtoH: beq x10, x9, unpress  # checks if done
      sw x30, 0(x22)         # activates an
      addi x10, x10, 1       # increments value
      add x15, x5, x10       # finds new address for LUT
      lb  x14, 0(x15)        # get new value
      sb x14, 0(x21)         # reassigns seg
      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 anode flickers on and off waiting for the input to be asserted. When it senses a button change, based on the display's current numerical value, the program determines whether to move the value up or down. It then repeatedly adds/subtracts while updating the display until its either 0 or 9. Then, it waits for the button to be released while flickering the anodes before entering into the original loop again when the buttons are unpressed.



Assembly Diagram

Hardware Assignment:

You must modify the RISC-V MCU to include three new instructions:

```
sb rs2,rs0(rs1) # store byte in rs2 at M[rs0 + rs1]
sh rs2,rs0(rs1) # store low 16 LSBs of rs2 at M[rs0 + rs1]
sw rs2,rs0(rs1) # load rs2 at M[rs0 + rs1]
```

For this problem, describe the following:

a) changes you need to make to the RISC-V MCU hardware

Since all the S-type instructions always require an immediate value instead of rs0, we likely need to dedicate a new opcode for these three instructions in order to use a register. If we take this approach, we'll need to modify the FSM and decoder to handle the new type. Since this instruction behaves very similar to both the R-type and the S-type though, we can use copy some of their behavior into our new type. Since we're doing memory stores just like we are normally, this opcode can retain the exact same behavior as our S-types in the FSM. In the decoder, since we need to add the rs1 and rs0 registers, we make the new opcode behave like a R-type that adds the two register values and passes this to the memory's ADDR2 input. This instruction doesn't interfere with any of the other modules like the PC, IMM_GEN, RegFile, or anything else so we only need to make the changes to the FSM/decoder.

b) changes you need to make to the RISC-V MCU assembler

The assembler needs to be told how to handle these instructions, so if we make a new instruction type, we'll have to add a new opcode to the assembler. The field should include this new opcode and three registers, rs0, rs1, and rs2 and additionally the 5-bit immediate value that we need to differentiate between the size of the storage element.

c) changes in RISC-V MCU memory requirements

We do not need to make any changes to the PC, memory module, and RegFile. No FSM states are added either and these new instructions could work entirely off of the hardware that we've already defined so there's no need to change the memory requirements for the new instructions.

d) if this change will work for both storing and outputting data

Although the new instructions will work well for storing data in our memory, no functionality was defined to output data as no memory reads were needed for these instructions.

e) why this modification would or could be useful

Instead of requiring an immediate value to use as a relative address pointer, having the ability to use a register makes things like for loops a lot more functional and intuitive. These instructions could be really

useful modifications to RISC-V as an alternative to loading an address and adding each time it needs to be changed.