Experiment #7

The RISC-V MCU with Interrupts

CPE 233

Bryan Mealy

07/06/2022

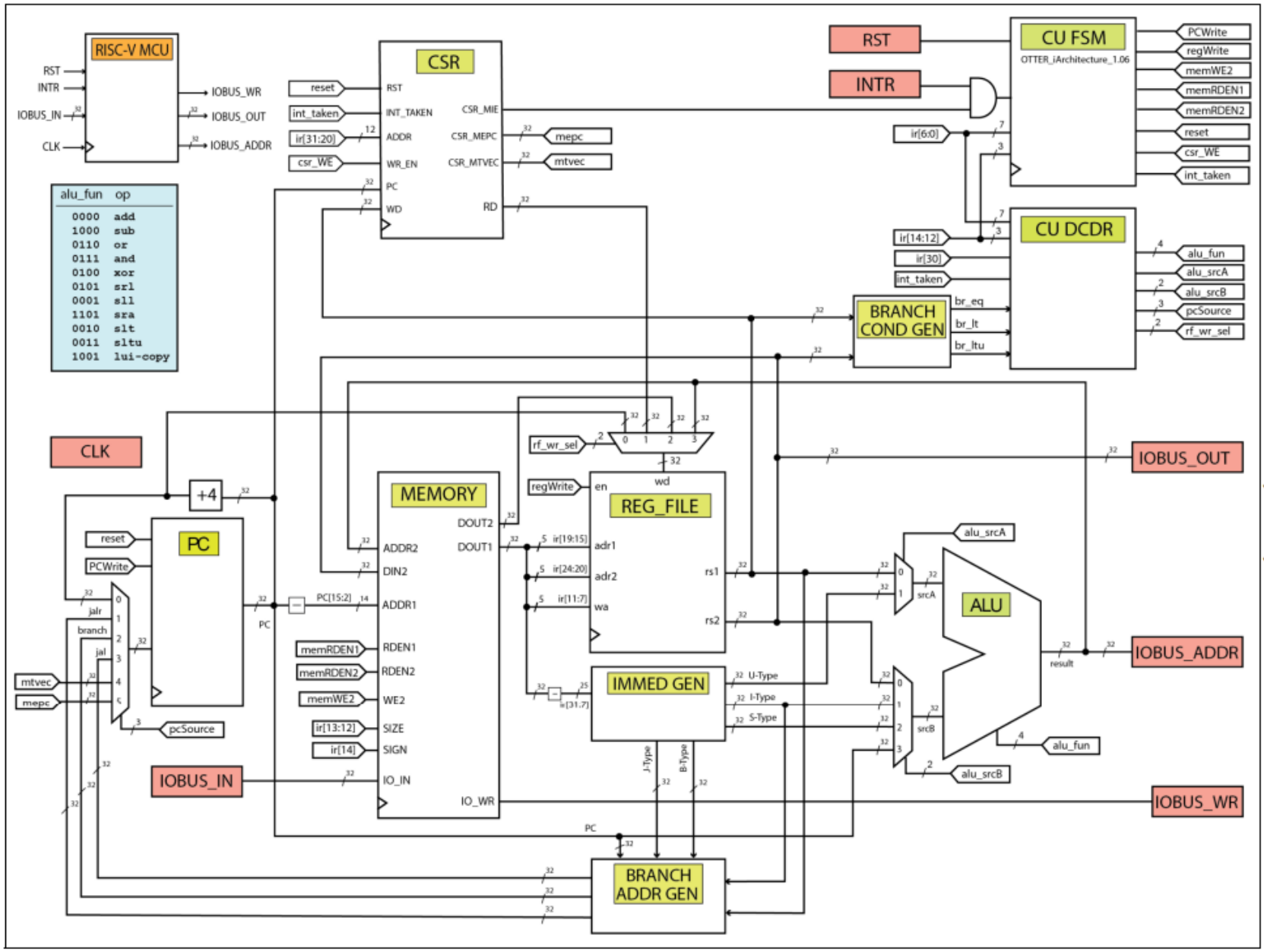
Danny Dang

Dylan Sandall

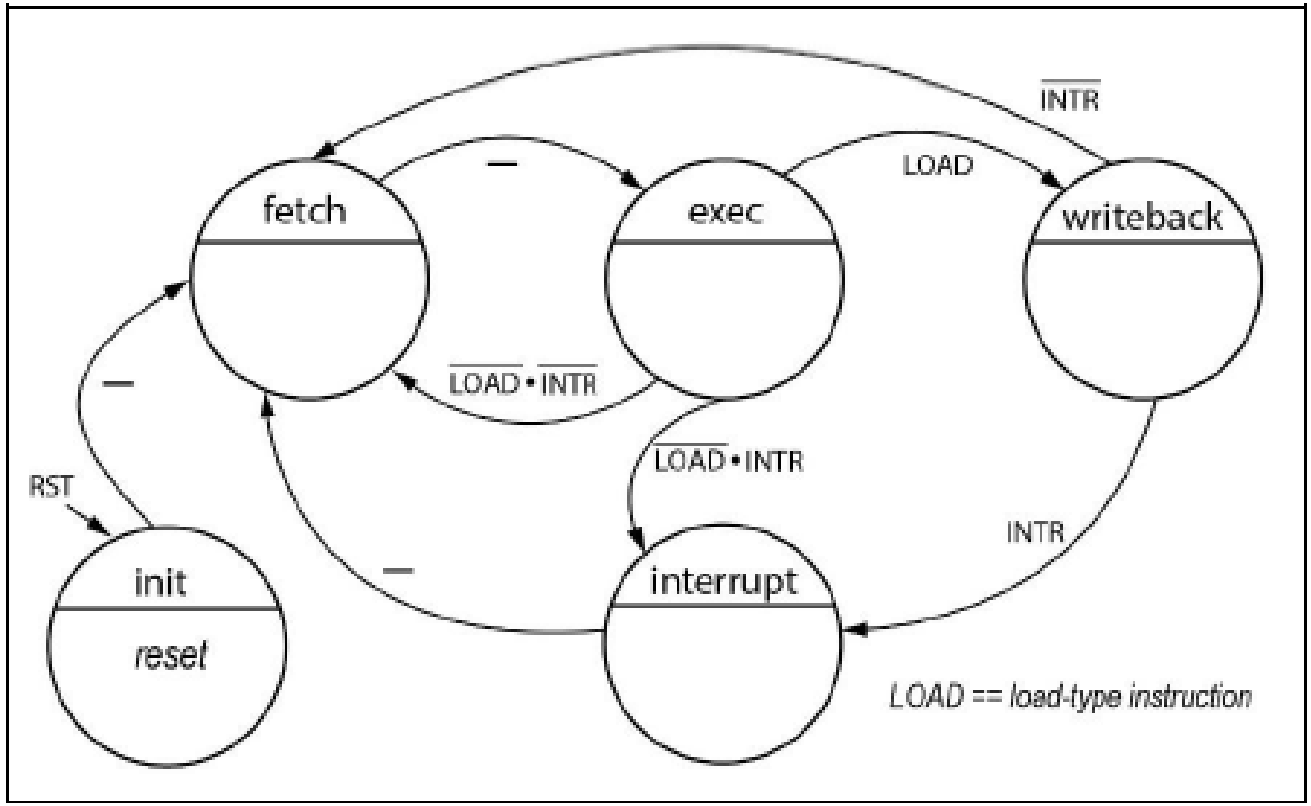
Felix Demharter

Summary: The control and status register was added to our RISC-V Otter MCU with the additions of more states for the state machine and cases for the decoder for the RISC-V to be able to handle interrupts. The compiled assembly code that is used for this experiment makes the LSB of the LED register toggles on and off whenever an interrupt is administered through a press of a button.

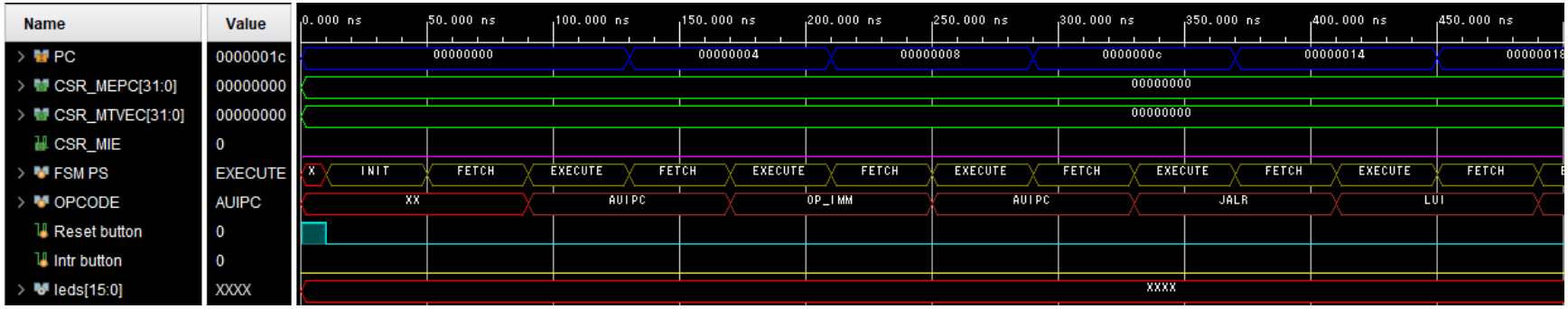
Schematics:

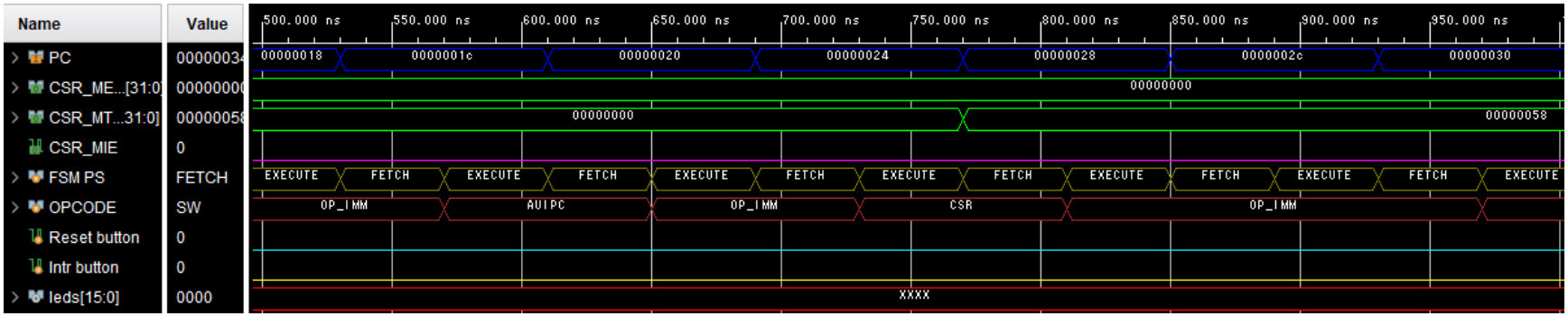

RISC-V MCU
RST
INTR
IOBUS_IN
CLK
IOBUS_WR
IOBUS_OUT
IOBUS_ADDR
CSR
reset
int_taken
ir[31:20]
csr_WE
RST
INT_TAKEN
ADDR
WR_EN
PC
WD
CSR_MIE
CSR_MEPC
CSR_MTVEC
RD
mepc
mtvec
RST
INTR
CU FSM
OTTER_iArchitecture_1.06
PCWrite
regWrite
memWE2
memRDEN1
memRDEN2
reset
csr_WE
int_taken
ir[6:0]
CU DCDR
ir[14:12]
ir[30]
int_taken
br_eq
br_lt
br_ltu
alu_fun
alu_srcA
alu_srcB
pcSource
rf_wr_sel
BRANCH COND GEN
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
CLK
+4
PC
reset
PCWrite
jalr
branch
jal
mtvec
mepc
pcSource
PC
PC[15:2]
MEMORY
ADDR2
DIN2
ADDR1
DOUT2
DOUT1
memRDEN1
memRDEN2
memWE2
ir[13:12]
ir[14]
SIZE
SIGN
RDEN1
RDEN2
WE2
IO_IN
IO_WR
IOBUS_IN
rf_wr_sel
wd
regWrite
REG_FILE
en
adr1
adr2
wa
rs1
rs2
ir[19:15]
ir[24:20]
ir[11:7]
IMMED GEN
ir[31:7]
U-Type
I-Type
S-Type
J-Type
B-Type
alu_srcA
alu_srcB
srcA
srcB
ALU
result
alu_fun
IOBUS_OUT
IOBUS_ADDR
IOBUS_WR
BRANCH ADDR GEN

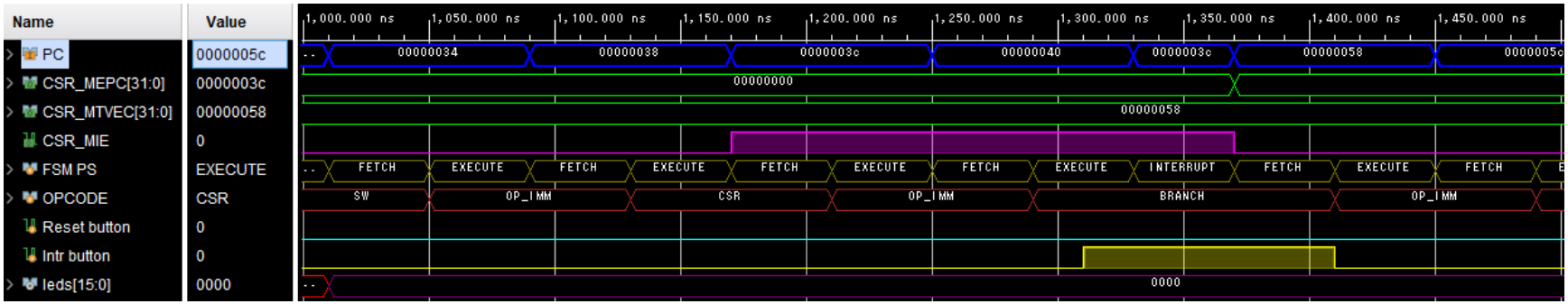
State Machine:

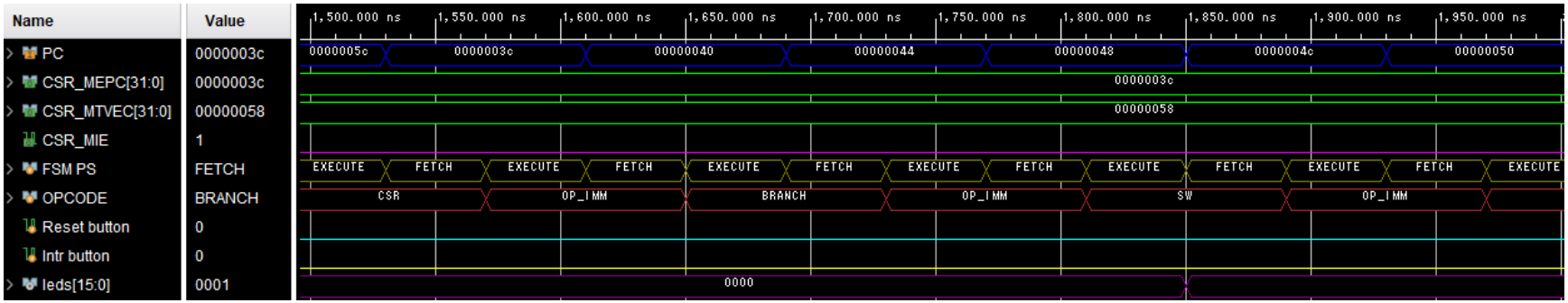


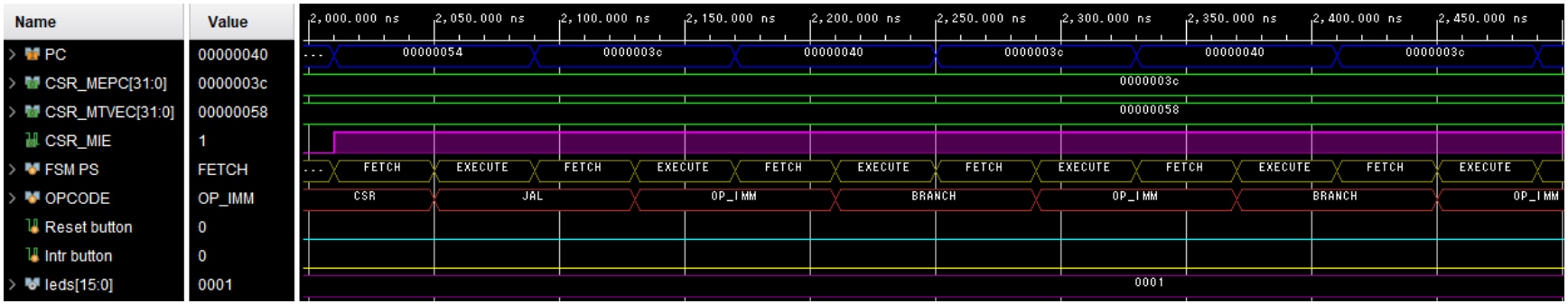
Timing Diagrams:

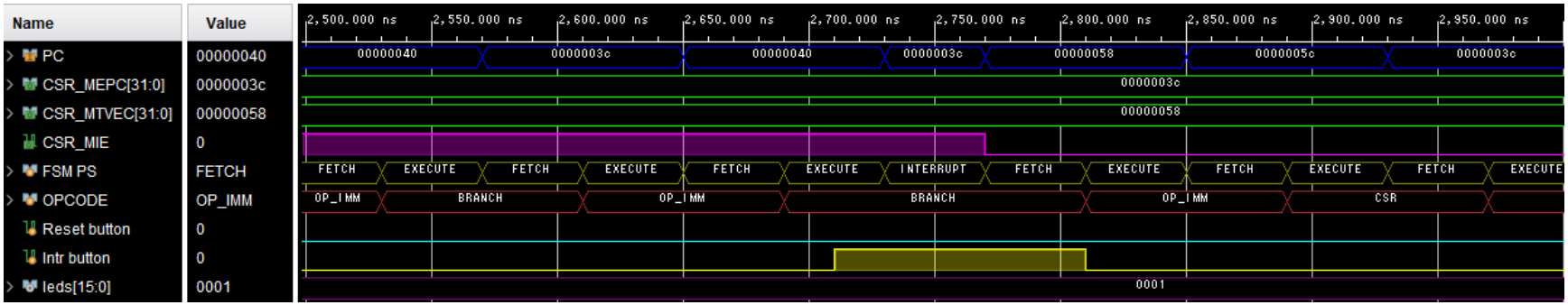


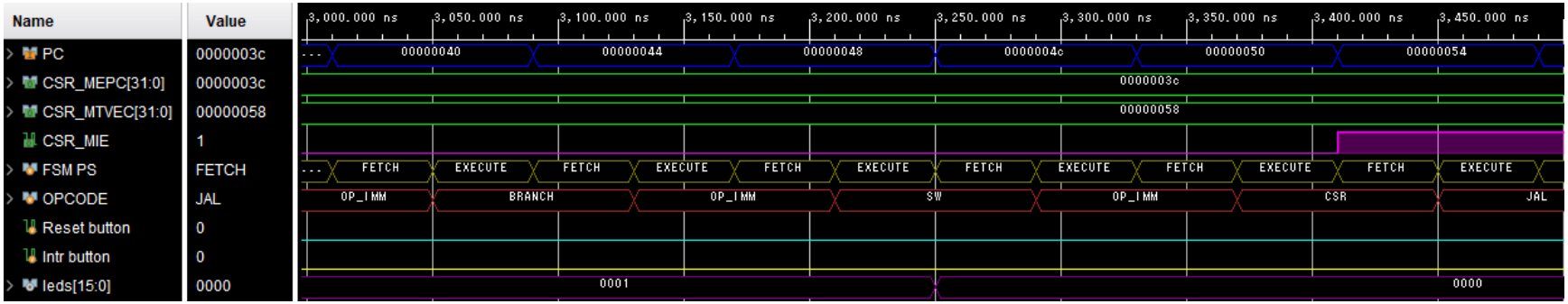



Name
Value
PC
0000005c
CSR_MEPC[31:0]
0000003c
CSR_MTVEC[31:0]
00000058
CSR_MIE
0
FSM PS
EXECUTE
OPCODE
CSR
Reset button
0
Intr button
0
leds[15:0]
0000
1,000.000 ns
1,050.000 ns
1,100.000 ns
1,150.000 ns
1,200.000 ns
1,250.000 ns
1,300.000 ns
1,350.000 ns
1,400.000 ns
1,450.000 ns
00000034
00000038
0000003c
00000040
0000003c
00000058
0000005c
00000000
00000058
FETCH
EXECUTE
INTERRUPT
SW
OP_IMM
CSR
BRANCH
0000


Name
Value
PC
0000003c
CSR_MEPC[31:0]
0000003c
CSR_MTVEC[31:0]
00000058
CSR_MIE
1
FSM PS
FETCH
OPCODE
BRANCH
Reset button
0
Intr button
0
leds[15:0]
0001
1,500.000 ns
1,550.000 ns
1,600.000 ns
1,650.000 ns
1,700.000 ns
1,750.000 ns
1,800.000 ns
1,850.000 ns
1,900.000 ns
1,950.000 ns
0000005c
0000003c
00000040
00000044
00000048
0000004c
00000050
0000003c
00000058
EXECUTE
FETCH
CSR
OP_IMM
BRANCH
SW
0000


Name
Value
PC
00000040
CSR_MEPC[31:0]
0000003c
CSR_MTVEC[31:0]
00000058
CSR_MIE
1
FSM PS
FETCH
OPCODE
OP_IMM
Reset button
0
Intr button
0
leds[15:0]
0001
2,000.000 ns
2,050.000 ns
2,100.000 ns
2,150.000 ns
2,200.000 ns
2,250.000 ns
2,300.000 ns
2,350.000 ns
2,400.000 ns
2,450.000 ns
00000054
0000003c
00000040
0000003c
00000040
0000003c
0000003c
00000058
FETCH
EXECUTE
CSR
JAL
OP_IMM
BRANCH
0001


Name
Value
PC
00000040
CSR_MEPC[31:0]
0000003c
CSR_MTVEC[31:0]
00000058
CSR_MIE
0
FSM PS
FETCH
OPCODE
OP_IMM
Reset button
0
Intr button
0
leds[15:0]
0001
2,500.000 ns
2,550.000 ns
2,600.000 ns
2,650.000 ns
2,700.000 ns
2,750.000 ns
2,800.000 ns
2,850.000 ns
2,900.000 ns
2,950.000 ns
00000040
0000003c
00000040
0000003c
00000058
0000005c
0000003c
0000003c
00000058
FETCH
EXECUTE
INTERRUPT
OP_IMM
BRANCH
CSR
0001


Name
Value
PC
0000003c
CSR_MEPC[31:0]
0000003c
CSR_MTVEC[31:0]
00000058
CSR_MIE
1
FSM PS
FETCH
OPCODE
JAL
Reset button
0
Intr button
0
leds[15:0]
0000
3,000.000 ns
3,050.000 ns
3,100.000 ns
3,150.000 ns
3,200.000 ns
3,250.000 ns
3,300.000 ns
3,350.000 ns
3,400.000 ns
3,450.000 ns
00000040
00000044
00000048
0000004c
00000050
00000054
0000003c
00000058
FETCH
EXECUTE
OP_IMM
BRANCH
SW
CSR
JAL
0001
0000

Source Code: Not full Verilog code for modules as it was specified to only show the modifications.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// 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: Changes to the CU_FSM with the addition of interrupts
//
//     //- 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]
    input [2:0] func3,
    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
);

    typedef  enum logic [2:0] {
        INIT,
        FETCH,
        EXECUTE,
        WRITEBACK,
        INTERRUPT
    }  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,
        CSR = 7'b1110011
    }

            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;
                        csr_WE = 1'b0;
                        int_taken = 1'b0;
                        if (intr)
                            NS = INTERRUPT;
                        else
                            NS = FETCH;
                    end

                    AUIPC:
                    begin
                        pcWrite = 1'b1;
```

```
    regWrite = 1'b1;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

BRANCH:
begin
    pcWrite = 1'b1;
    regWrite = 1'b0;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

SW:
begin
    pcWrite = 1'b1;
    regWrite = 1'b0;
    reset = 1'b0;
    memWE2 = 1'b1;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

LW:
begin
    pcWrite = 1'b0;
    regWrite = 1'b0;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b1;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    NS = WRITEBACK;
   if (intr)
       NS = INTERRUPT:
  else
       NS = FETCH;
end
```

```
OP_IMM:
begin
    pcWrite = 1'b1;
    regWrite = 1'b1;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

OP_RG3:
begin
    pcWrite = 1'b1;
    regWrite = 1'b1;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

JAL:
begin
    pcWrite = 1'b1;
    regWrite = 1'b1;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
end

JALR:
begin
    pcWrite = 1'b1;
    regWrite = 1'b1;
    reset = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        NS = INTERRUPT;
    else
        NS = FETCH;
```

```
    end

    CSR:
    begin
        pcWrite = 1'b1;
        regWrite = func3[0];
        reset = 1'b0;
        memWE2 = 1'b0;
        memRDEN1 = 1'b0;
        memRDEN2 = 1'b0;
        csr_WE = func3[0];
        int_taken = 1'b0;
        if (intr)
            NS = INTERRUPT;
        else
            NS = FETCH;
    end
    endcase
    end


WRITEBACK:
begin
    pcWrite = 1'b1;
    regWrite = 1'b1;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    reset = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b0;
    if (intr)
        begin
            NS = INTERRUPT;
        end
    else
        begin
            NS = FETCH;
        end
end



INTERRUPT:
begin
    pcWrite = 1'b1;
    regWrite = 1'b0;
    memWE2 = 1'b0;
    memRDEN1 = 1'b0;
    memRDEN2 = 1'b0;
    reset = 1'b0;
    csr_WE = 1'b0;
    int_taken = 1'b1;
    NS = FETCH;
end
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// 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: Changes to the CU_DCDR with the addition of interrupts
//
// 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 ()   );
//
//////////////////////////////////////////////////////////////////////////////////

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 alu_srcA,
    output logic [1:0] alu_srcB,
    output logic [1:0] rf_wr_sel   );

                    CSR:
                    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'b01;  // CSR_RD
                        if (func3 == 3'b000)
                            begin
                                pcSource = 3'b101; // mepc
                            end
                        else
                            begin
                                pcSource = 3'b000; // PC + 4
                            end
                    end

endmodule
```

Questions:

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

The AND gate is there to let MCU focus on other tasks rather than dividing its attention to always be waiting for interrupts. The AND gate lets the RISC-V MCU receive interrupts when it is convenient.

2. Based on the state diagram in Figure 22, would it be possible to assert the interrupt signal and not enter into an interrupt cycle? Briefly but completely explain. Assume the interrupt is unmasked.

It is possible to not go into the interrupt cycle temporarily, as being in the states of INIT and Fetch does not immediately enter the interrupt cycle when an interrupt signal is asserted but rather the interrupt is buffered to happen later if it is asserted during these states.

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.

It's generally a good idea to mask interrupts as soon as one is received, because you never want to leave an interrupt service routine unfinished. This could be a task left to the programmer, but when something is always done, it just makes more sense to put it in the hardware. Leaving something like this in program control leaves much room for error, and the advantages are not worth the risk.

4. 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.

When an interrupt is received, the only things that **must** be done are loading the vector address (ISR address) into the PC, and storing the next address for later. Because these actions can be completed in 1 clock cycle, only 1 state is needed to guarantee enough time has passed for these to finish.

5. 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.

Interrupts are asynchronous because they are externally controlled - they can be asserted for no discernable reason. Because the RISC-V MCU works synchronously, receiving interrupts directly can interrupt the flow of the firmware and instructions. In order to process the asynch interrupts to work with the MCU, the interrupt is passed through a debouncer and oneshot (if connected to a switch or button), then to a register. This register maintains the interrupt until the MCU has "received" it, and sets

it back to 0. When the FSM determines that an interrupt can be received, and the firmware has enabled interrupts, then and only then will the interrupt be received. The MCU then performs the interrupt

6. 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.

Assuming interrupts are masked when received, the key problem here is missing an interrupt. This problem is mitigated by saving (holding) the interrupt, and keeping it asserted until the MCU confirms that it has been recieved.

7. 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.

Programmers typically write their code with the notion that it will be run linearly, with each block finishing before it moves onto the next. An MCU receiving and handling an interrupt mid interrupt is like a child making a sandwich, who then decides they actually want mac and cheese, and leaves half a loaf of bread and a dirty knife on the counter. When their parents (later code) goes to use the kitchen, they will find things not in their place (registers or memory with unexpected values). The analogy starts to break down at this point, but the idea is that code cannot be interrupted at just any point, it has to finish the intended operation before moving onto the next (if you want it to operate properly).

8. 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.

The use of polling does not automatically make your program bad, as long as it’s used when no other meaningful work can be done. The problems with polling are 1- worse reaction time and 2- nothing can be computed while you are asking for a new input. As long as 1- minor reaction times are not a big deal and 2- there are no computations that can be run in the background before the signal is received, then polling is a fine solution.

Programming Assignment:

The programming assignment in the previous experiment required that you create a “bouncing” LED display, where the LEDs turned with a “heartbeat” style. You will repeat that heartbeat style with this programming assignment, but you will use it to count continuously from zero to nine on the dev board's 7-segment display. You must implement your solution on the dev board. For this problem, you must use a LUT for the counting in the problem. The count frequency should be in the range of 1-2Hz similar to the programming assignment in the previous experiment, which certainly means you should reuse a majority of your RISC-V code from that solution. Also for this problem, include a flowchart describing your algorithm, and a complete written description of the algorithm you used in your solution.
Note that Figure 26 seven-segment LUT data you need for this problem; you must use this data to drive the segments on the seven-segment displays. Since you are using a LUT for this problem, you must use the RARS assembler as well. Use the one on Canvas.

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

Assembly Code:

```
# ------------------------------------------------
# SUBROUTINE: heartbeat counter - exp 7
# SEG      - x7 contains 7S address
# Dseg     - x9 desired SEGs to turn on
# Len      - x11 granularity of PWM signal and length of timer
# LenLow   - x12 how long a SEG will stay off between numbers
#          - x13 current address
# LUTlim   - x14 max LUT entry (9)
# TIME     - x29 counter, for duty cycle
# DUTY     - x30 duty cycle of SEG, out of TIMERlen
# ------------------------------------------------

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

.text
Lab7_heartcount:
init:      li    x7, 0x1100C008  # x7 = AN
           sw    x0, 0(x7)       # set AN to all

           addi  x7, x7, -4      # AN - 4 = SEG = x7
           li    x11, 2047       # Len = 2047
           li    x12, -300       # LenLow = -300
           mv    x30, x0         # x30 = DUTY = 0
rstLUT:    la    x13, sseg       # get address of first sseg
           li    x14, 10         # LUTlim = 10

loop:      lbu   x9, 0(x13)      # load byte from address

on:        call  Timer_heartbeat # call timer routine
           addi  x30, x30, 1     # increment DUTY level
           ble   x30, x11, on    # if DUTY within Len, loop

off:       call  Timer_heartbeat # call timer routine
           addi  x30, x30, -1    # decrement DUTY level
           bge   x30, x12, off   # if DUTY > LenLow, loop

admin:     addi  x13, x13, 1     # increment LUT address
           addi  x14, x14, -1    # decrement LUTlim
           beqz  x14, rstLUT     # if LUTlim finished, reset LUT

           j     loop            # repeat main loop
```

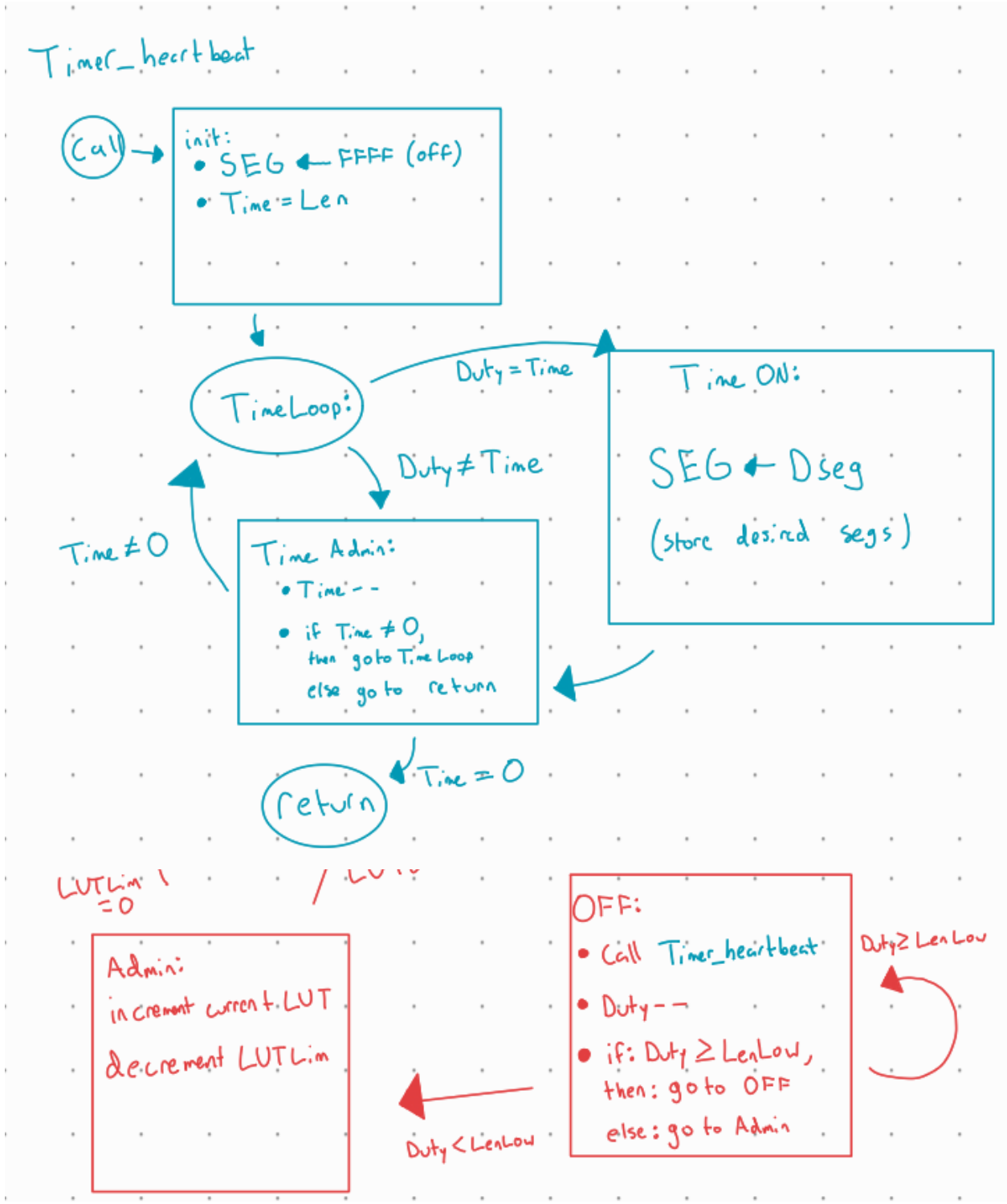
```
# ------------------------------------------------
# SUBROUTINE: Timer V3 - exp 7
# SEG      - x7  - i - contains 7S address
# Dseg     - x9  - i - desired SEGs to turn on
# Len      - x11 - i - granularity of PWM signal and length of timer
# LenLow   - x13 - i - how long a SEG will stay off between numbers
# TIME     - x29 -   - counter, for duty cycle
# DUTY     - x30 - i - duty cycle of SEG, out of TIMERlen
# ------------------------------------------------

Timer_heartbeat:
          not   x29, x0              # load x29 with FFFF
          sw    x29, 0(x7)           # init SEGs as off (active LOW)
          mv    x29, x11             # time = Len

timeloop:  beq  x30, x29, timeon     # branch if duty = time
          j     timead               #  else jump to admin
timeon:   sw    x9, 0(x7)            # turn on desired SEGs
timead:   addi  x29, x29, -1         # decrement time
          bnez  x29, timeloop        # repeat until time reaches lim
          ret
```

Written Description and Demo:

Demonstration Video Heartbeat Counter

The code begins by enabling all ANs and initializing values and parameters. Then, the LUT is loaded, and the LUT reset param is set. This allows the code to reset the LUT when it has reached the final element. The main loop now begins, by loading the byte in the current entry of the LUT (this is the segs to show the value 0 on the 7seg display). The segments will now fade on, calling the Timer_heartbeat subroutine. It starts with a Duty value of 0 and a len 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 (as this is what it is left at after reaching 100% duty), LenLow is -300. Timer_heartbeat is called, Duty is now *decremented*, and this repeats until Duty reaches -300. LenLow is set to -300 to allow for a short pause between number fades. Between duty = 0 and duty = -300, it is on for 0% of the time, and this manifests itself as a short delay before another digit is displayed. Now that our first digit has been turned on and off, we increment the digit to display by incrementing and reloading the LUT, as well as decrementing our counter to keep track of what digit we are at. This process will repeat 10 times, once for each digit, and then reset the LUT to digit 0 once again.

Accompanying Flowcharts:



Hardware Design Assignment:

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

```
lb      rd,rs2(rs1)          # load rd with the data at M[rs2 + rs1]
lh      rd,rs2(rs1)          # load 16 LSBs of rd at M[rs2 + rs1]
lw      rd,rs2(rs1)          # load rd at M[rs2 + rs1]
```

For this problem, describe the following:

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

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

c) changes in RISC-V MCU memory requirements

d) why this modification would or could be useful

A) These three new instructions are load types and I-types which already have similar instructions in our current MCU. There are currently 5 load-type instructions that exist in our MCU that all share the opcode of "0000011" and since there are only five and the 3-bit func3 opcode has three unused bit combinations, we can fit the three new instructions into there without changing much. There are other load instructions in the MCU but the main difference between those and the new ones are that the old ones choose an immediate value for the absolute address and the new one is that is chooses rs2 as it's address **"lw rd, rs2 (rs1)"** so then we would need modify the **CU_DCDR** to make sure that the selection on the **alu_srcB** mux is the correct signal for the new load instructions.

B) We would need to notify the assembler that there are three new instructions and the new opcodes that are to be added.

C) We did not change any memory requirements as we only touched the CU_DCDR.

D) The modification could be more useful because it allows assembly writers to have more options as they implement higher-level code. The previous address calculation only used one register and one immediate value while the new instructions uses two register values which will make the load-type instructions be extremely more useful.