Experiment #9C

More Interrupts on the RISC-V MCU

CPE 233

Bryan Mealy

06/06/2022

Danny Dang

Dylan Sandall

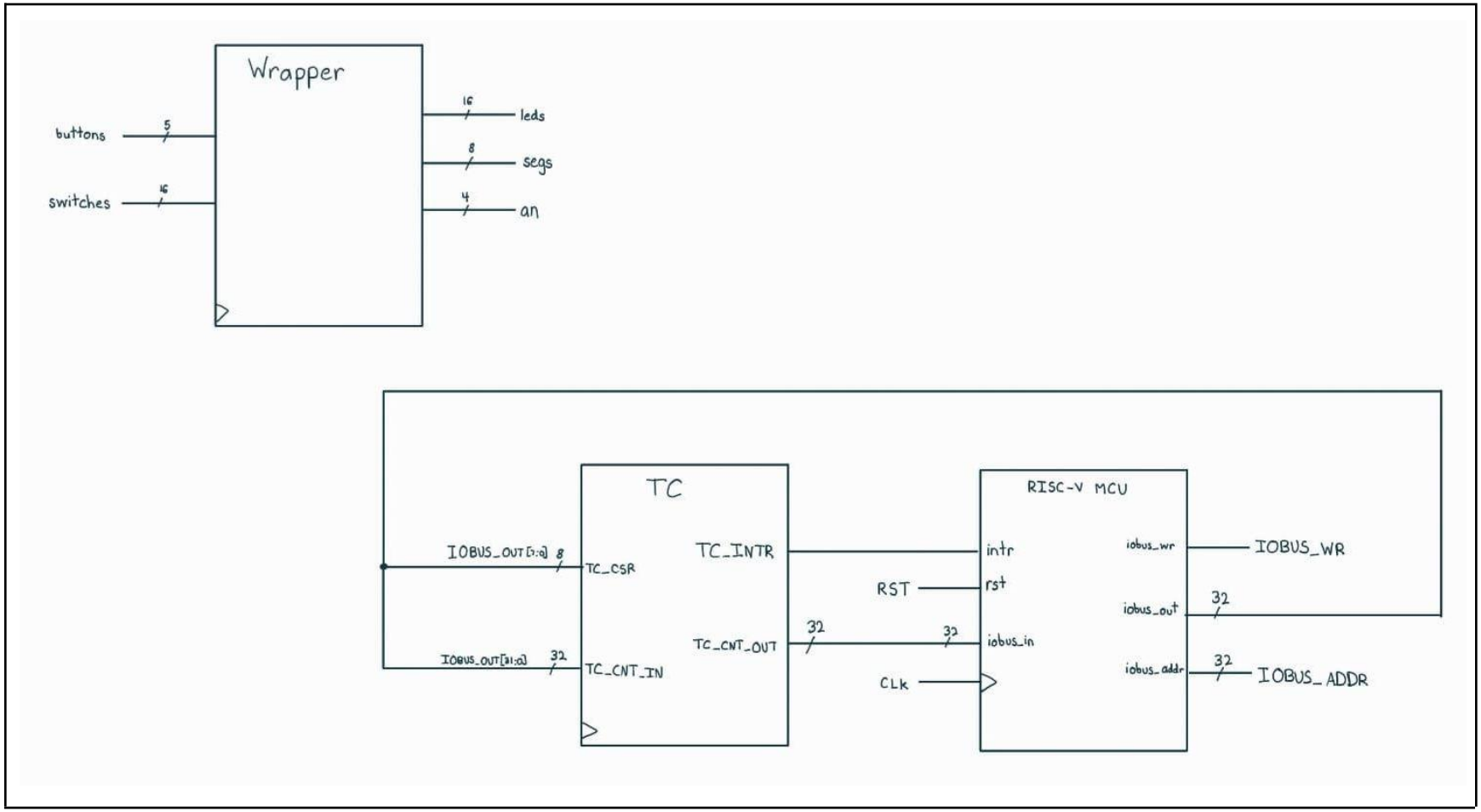
Felix Demharter

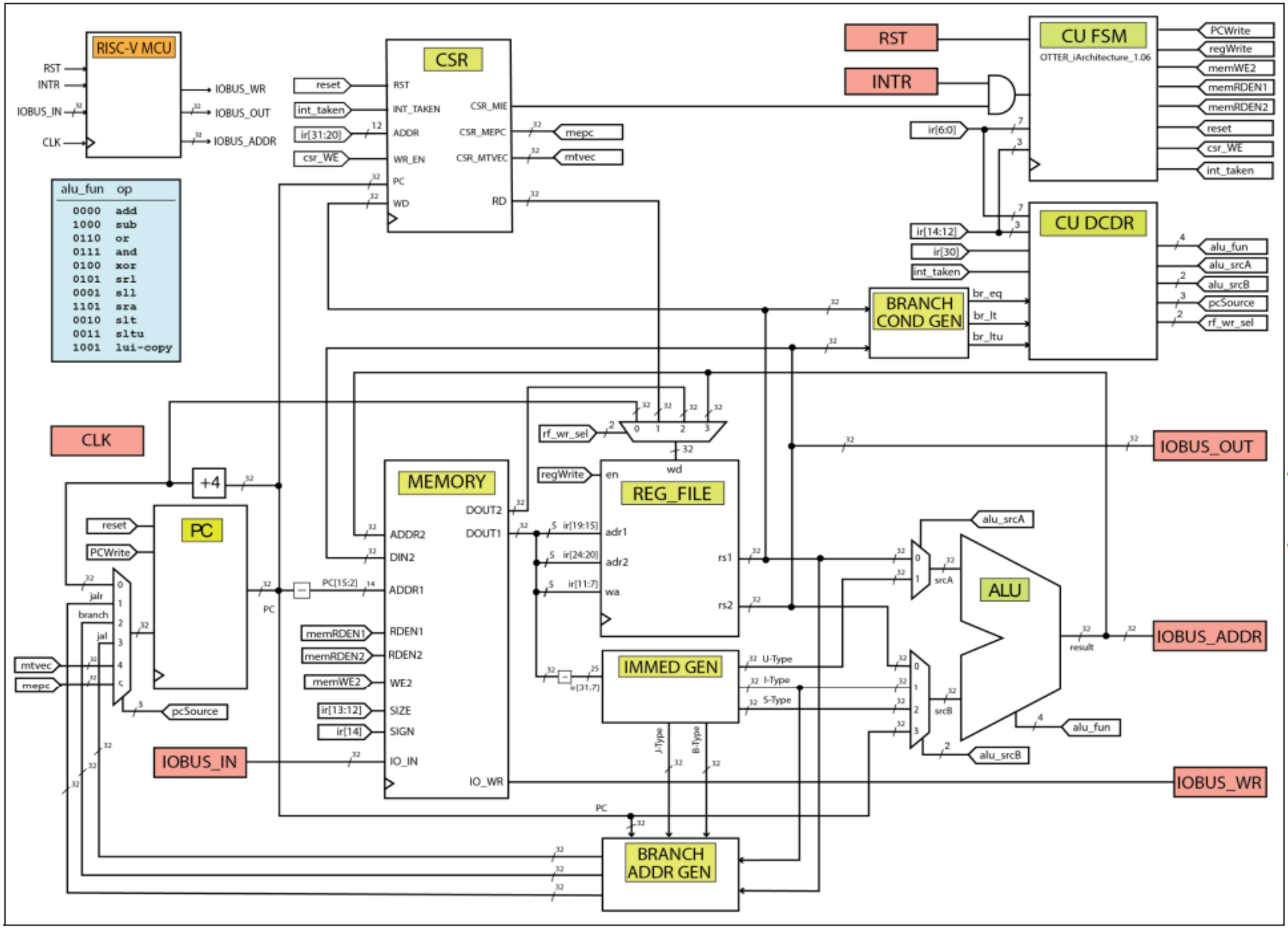
Summary:
We implemented the Timer/Counter module, and wrote accompanying assembly code. 9C directed us to write code for a button press counter that goes from 0-49, outputs to the 7 segment display using time delay and lead-zero blanking, as well as debouncing the input button in firmware.

Demo Video

Schematics:

Higher Level Schematic




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
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
ir[6:0]
PCWrite
regWrite
memWE2
memRDEN1
memRDEN2
reset
csr_WE
int_taken
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
CLK
+4
PC
reset
PCWrite
jalr
branch
jal
mtvec
mepc
pcSource
PC[15:2]
MEMORY
ADDR2
DIN2
ADDR1
RDEN1
RDEN2
WE2
SIZE
SIGN
IO_IN
IO_WR
DOUT2
DOUT1
memRDEN1
memRDEN2
memWE2
ir[13:12]
ir[14]
IOBUS_IN
regWrite
REG_FILE
en
wd
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
rf_wr_sel
IOBUS_OUT
alu_srcA
srcA
ALU
srcB
result
alu_fun
alu_srcB
IOBUS_ADDR
IOBUS_WR
PC
BRANCH ADDR GEN

Source Code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:  Danny Dang, Dylan Sandall, Felix Demharter
//
// Create Date: 03/08/2020 02:46:31 PM
// Design Name:
// Module Name: OTTER_Wrapper
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Otter Wrapper for timer-counter reference design.
//
// Dependencies:
//
// Revision:
//
// Additional Comments: modified/extended version of Exp 5 Otter Wrapper
//
//////////////////////////////////////////////////////////////////////////////////

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

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

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


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

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


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

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

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


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

 logic [31:0] s_tc_cnt_out;
 logic s_tc_intr;

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

 //- Instantiate RISC-V OTTER MCU
 OTTER_MCU  my_otter(
    .RST         (s_reset),
    .intr        (s_tc_intr),
    .clk         (s_clk),
    .iobus_in    (IOBUS_in),
    .iobus_out   (IOBUS_out),
    .iobus_addr  (IOBUS_addr),
    .iobus_wr    (IOBUS_wr)   );

timer_counter #(.n(3))  my_tc (
   .clk        (s_clk),
   .tc_cnt_in  (r_tc_cnt_in),
   .tc_csr     (r_tc_csr),
   .tc_intr    (s_tc_intr),
   .tc_cnt_out (s_tc_cnt_out)  );

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

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

```
    end

    //- MUX to route input devices to I/O Bus
       //-   IOBUS_addr is the select signal to the MUX
       always_comb
    begin
        IOBUS_in=32'b0;
        case(IOBUS_addr)
            SWITCHES_PORT_ADDR: IOBUS_in[15:0] = switches;
                    BUTTONS_PORT_ADDR:  IOBUS_in[4:0]  = buttons;
                    TMR_CNTR_CNT_OUT:   IOBUS_in[31:0] = s_tc_cnt_out;
            default: IOBUS_in=32'b0;
        endcase
    end

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

endmodule
```

```
#lab9 button counter to 49

.data
sseg:          .byte   0x03,0x9F,0x25,0x0D,0x99,0x49,0x41,0x1F,0x01,0x09,0xFF
# LUT for 7-segs (includes blank char)

.text
main:
init:  la     x3, sseg             # load LUT
       li     x7, 0x11008004       # button address
       li     x8, 0x1100C008       # x8 = AN
       li     x9, 0x1100C004       # x9 = SEG
       li     x10, 10
       li     x11, 5
       li     x17, 256             # 256 = debounce length
       li     x26, 1               # easily accessible 1
                                   # init timer
       li     x22, 0x1100D000      # timer counter CSR port address
       li     x23, 0x1100D004      # timer counter count port address
       li     x5, 0x01             # init TC CSR
       sw     x5, 0(x22)           # no prescale, turn on TC
       li     x5, 0x00044444       # for ~90Hz blink rate
       sw     x5, 0(x23)           # init TC count

                                   # init interrupts
       la     x5, ISR              # load address of ISR into x5
       csrrw  x0, mtvec, x5        # store address as interrupt vector CSR[mtvec]

clear: li     x31, 0               # init x31 accumulator

loop:  csrrw  x0, mie, x26         # enable interrupts
       lbu    x15, 0(x7)           # get button data
       andi   x15, x15, 1          # mask LSB
       beqz   x15, clear           # if button is off, clear and continue polling
       addi   x31, x31, 1          # accumulate
       bne    x31, x17, loop       # loop if button has not been on for req length

press: addi   x19, x19, 1          # increment 1's
       blt     x19, x10, rclear    # if x19 is 10 or higher:
       addi   x18, x18, 1          # add one to tens
       mv     x19, x0              # clear ones
       bne    x18, x11, rclear     # if reached 50:
       mv     x18, x0              # clear

rclear: li    x31, 0               # init x31 accumulator

rloop: csrrw  x0, mie, x26         # enable interrupts
       lbu    x15, 0(x7)           # get button data
       andi     x15, x15, 1        # mask LSB
       bnez    x15, rclear         # if button is on, clear and continue
       addi    x31, x31, 1         # accumulate
       bne    x31, x17, rloop      # loop if button has not been on for req length

       j    clear
```

```
#ISR
ISR:   not     x4, x4              # toggle current digit
       beqz     x4, set1           # if flag = 0, display 1's
                                   # else, set 10's
                                   # set 10's (with lead 0 blanking!)
set10: mv     x5, x18
       bnez   x5, nz              # if zero:
       li     x5, 10              # set to char10 (blank)

nz:    add    x5, x3, x5          # offset LUT
       lbu    x5, 0(x5)           # get digit seg values, place in x5
       sb     x5, 0(x9)                  # push digit to seg
       li     x5, 0xb             # address for dig1

       sb     x5, 0(x8)           # set AN to selected
       j      iret

# set 1's
set1:  add    x5, x3, x19         # offset LUT
       lbu    x5, 0(x5)           # get digit seg values, place in x5
       sb     x5, 0(x9)           # push digit to seg
       li    x5, 0x7              # address for dig0
       sb     x5, 0(x8)           # set AN to selected

iret:    mret
```

Questions:

1. Briefly describe why using a timer-counter peripheral to blink an LED is “more efficient” than using a dumb loop (delay loop) to blink an LED.

In this experiment, the time-counter uses an interrupt to blink the LED which means that it is running as a background task which then allows us to be able to run a foreground task with more processing power. If we were to use the delay loop to blink the LED, all the processing power would be focused on blinking the LED and that is not as efficient as doing it with a background task.

2. List all the possible ways that the RISC-V uses to mask and unmask the interrupt.

RISC-V makes use of CSR_MIE, a register that can enable or disable the receiving of an interrupt (going into the interrupt state in the FSM). This register is controlled both by the program, as well as hardware. Interrupts are disabled when received, and enabled/disabled under program control.

3. Examine the Verilog model for the timer-counter and briefly but completely describe how the module operates. Be sure to mention both the counter portion as well as the pre-scaler.

The counter portion of the timer- counter module operates very similarly to a regular counter where it will count up from zero until it reaches the allowed max count and then resets to zero and repeats. The main difference is that the module outputs a pulse when its count is equivalent to the count in the TCCNTx registers which are the 8-bit registers that hold the terminal counts. The duration of the output pulse is two system clock periods.

The prescaler portion of the timer count module is controlled by a 4-bit signal that prescales the clock input to the timer counter. It allows the user to reduce the input clock frequency and takes effect when the timer counter reaches the current prescale value.

4. If you configured the timer-counter to generate an interrupt on the RISC-V MCU every 10ms, what is the highest frequency blink rate of an LED using that interrupt? Briefly explain your answer.

The highest frequency blink rate LED would be 50hz because the period of the LED blink cycle would be 20ms and the frequency is the inverse of the period (1/20ms) is equal to 50 Hz.

5. Briefly but completely explain how using the “clock prescaler” will prevent the firmware programmer from blinking the LED at all possible frequencies lower than the system clock frequency. Keep in mind you can still blink at some frequencies lower than the system clock frequency, but not all of them. For this problem, assume the timer-counter module uses the system clock.

The LEDs are prevented from being blinked at lower frequencies than the system clock because the clock prescaler sets the frequency to the multiplicand of the system clock and a natural number. The frequency can only be the system clock or the greater.

6. Changing frequencies of the timer-counter can possibly create a timing error for one clock period. Briefly describe what two issues can cause this one hiccup. HINT: one has to do with the count in the timer-counter; the other has to do with the RISC-V assembly code.

If we were to change the TCNTx numbers during a clock cycle, the clock may count up to the new values instead of the values set at the beginning of the clock cycle. This would cause the cycle to end at the wrong time.

7. The timer-counter provided for this experiment provides a means to easily blink an LED with a 50% duty cycle. Using the RISC-V MCU and the timer-counter, there is a programming overhead” associated with blinking the LED at an exact frequency if the duty cycle is not 50%. Briefly describe the cause of this overhead code, as well as how and when this overhead can interfere with the frequency output of the blinking LED.

The timer/counter sends a consistent, evenly spaced signal. In order to produce a signal that has a duty length other than 50%, these signals must not be evenly spaced. You could set the LED to turn on after A consecutive interrupts, and turn off after B consecutive interrupts, leading to an altered duty cycle, but this would also increase your required period, and add programming overhead. It would take additional operations to make these duty cycle calculations.

8. Briefly describe how you would use the timer-counter module to “time” the length of time a given signal is asserted. Note that your answer should have nothing to do with interrupts.

I would add an output so I can read the value of the timer’s count so I can start the timer when the signal is asserted and then when the signal is no longer asserted, I will be able to read the output of the timer to get the time.

9. If you tied the output of the timer-counter (as you configured the TC in this experiment) to a debounce and the output of the debounce to the ISR input of the RISC-V MCU, would you be able to generate an interrupt? Briefly explain your answer.

Yes, as long as the time the timer counter is on for (the pulse width when it is high) is long enough to trigger the debounce. If the pulse length is too short, the debounce will interpret the pulse as a bounce, and not assert the interrupt.

Programming Assignment:

Write an interrupt driven RISC-V MCU assembly language program that does the following. The program outputs the largest of the three most recent values that were on the switches when the RISC-V MCU receives an interrupt to the LEDs. Assume there are 16 LEDs and 16 switches. Interpret the 16-bit switch values as unsigned binary numbers. Use the port addresses associated with the standard RISC-V wrapper (not from Experiment 6) for this problem.

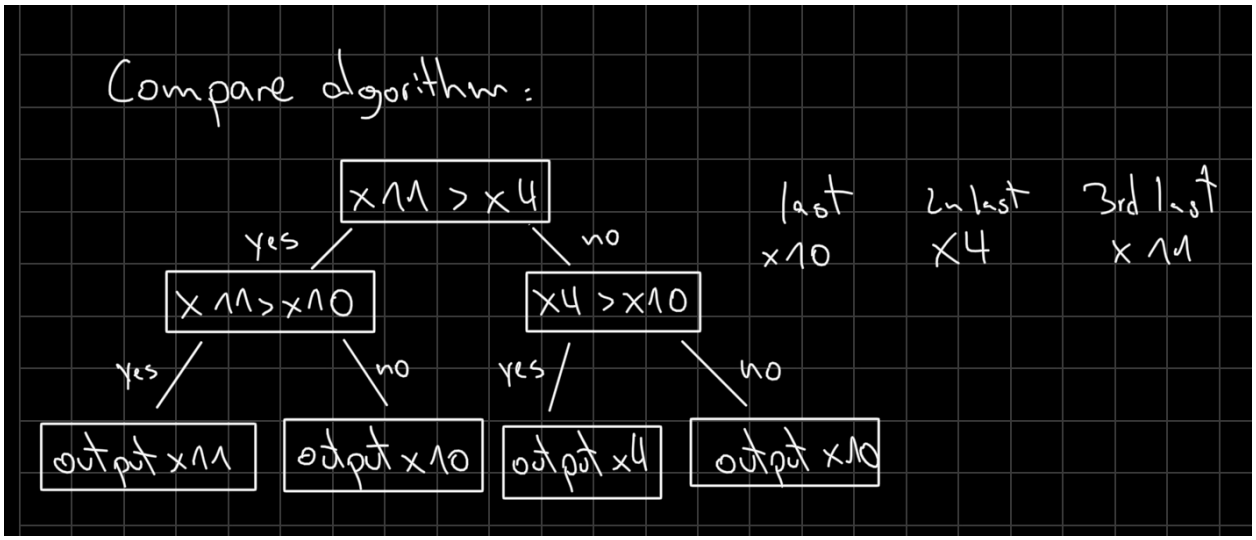
- Don't perform any IO in the interrupt service routine
- Assume an external device generates the interrupt
- Minimize the number of instructions in your solution

Assembly Code:

```
#-------------------------------------------------
# SUBROUTINE: threerecent  - exp 9
# -------------------------------------------------
threerecent:
    li          x6, 0x11008004       #button address
    li          x9, 0x11008000            #switch address
    li          x8, 0x1100C000            #LED address
    lhu         x10, 0(x9)           #load initial switch position
    li          x4, 0                #initialize second last
    li          x11, 0               #initialize third last
    la          x7, ISR              #ISR address
    csrrw       x0, mtvec, x7        #into mtvec

initinterrupt:
    li          x27, 1               #set x27
    csrrw       x0, mie, x27         #enable interrupts

wait:
    lhu         x5, 0(x9)            #load current into x5
    bne         x5, x10, move        #if current==previous
    bnez        x27, wait

compare:
    bgtu        x11, x4, checkthird
    bgtu        x4, x10, issecond
    sh          x10, 0(x8)           #LED <-- largest
    j           initinterrupt

checkthird:
    bgtu        x11, x10, isthird
    sh          x10, 0(x8)           #LED <-- largest
```

```
    j           initinterrupt

issecond:
    sh          x4, 0(x8)             #LED <-- largest
    j           initinterrupt

isthird:
    sh          x11, 0(x8)            #LED <-- largest
    j           initinterrupt

move:
    mv          x11, x4               #move second last into third
last
    mv          x4, x10               #move last into second last
    mv          x10, x5               #move current into last
    j           wait

# ------------------------------------------------
# ISR
# ------------------------------------------------
ISR:
    mv          x27, x0                #int received
    mret
```

Diagram of compare algorithm:



Reasoning:

The reason why we use 4 registers (x5 current, x10 last, x4 second last, x11 third last) to keep track of the states of the switches is because if they don't change but the interrupt is called x5 and x10 would have the same value and thus we would only have 2 states to compare

Hardware Design Assignment:

My particular application does many of the same operations, so I want a new instruction to support them. The RTL below shows the new instruction; note that rs2 is both a source and destination operand for this instruction. For this instruction, do not make any changes to the ALU or the memory module, but minimize the hardware you use in your design. For this problem, fully describe the following:

```
adddiv     rs2,imm,rs1     # X[rs2] ← (X[rs1] + X[rs2]) >> imm
```

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

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

c) changes in RISC-V MCU memory requirements

d) specifically why this modification would be useful

Answer:

A. This new instruction seems to be the combination of the already existing “add” and “immediate shift right” instruction. We can achieve these two separate instructions with the one instruction above by adding a register after the ALU so we can store the output of the addition of rs1 and rs2. Then, we have to add a new state that will run the last instruction for us shifting to the immediate value right and have to increase the srcA mux size to 2 bits so we can add the new alu register output to it that will be controlled by the state machine. We can use an already existing type for this instruction such as B or S type as they deal with immediate values and rs1, rs2 but with a new opcode since the instruction is different from the currently existing instructions in those types but can reuse a func3 code.

B. We would need to let the assembler know of the new instruction so it can send out the correct opcodes.

C. We added a new register after the ALU, which would be an increase in memory. We did not touch the CSR, memory, PC and the old REG so no memory modifications there. We added a new state to the FSM but the state machine has 4 unused states so no need for more state registers.

D. A two in one instruction allows for more concise code.