Experiment #8

Using Interrupts on the RISC-V MCU

CPE 233

Bryan Mealy

06/06/2022

Danny Dang

Dylan Sandall

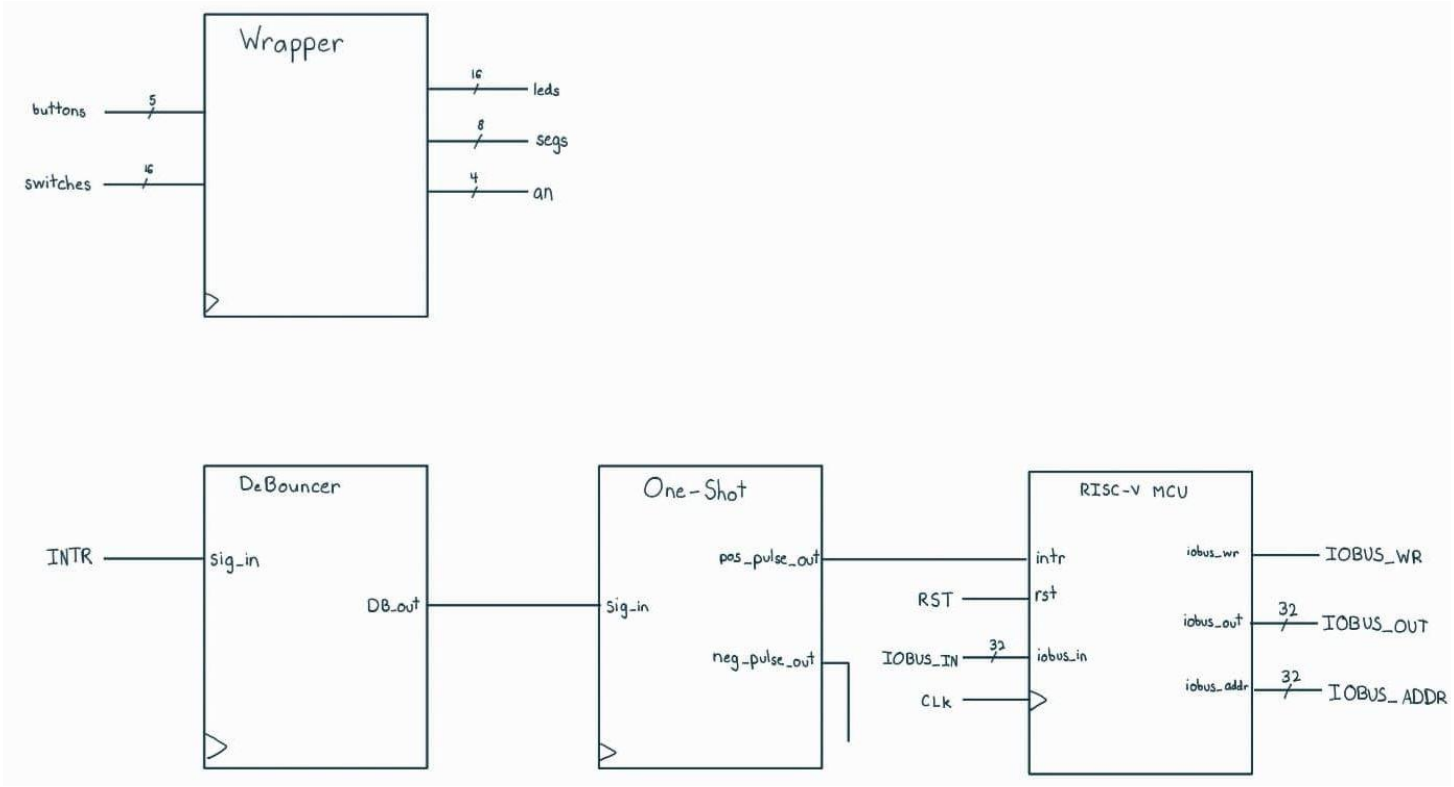
Felix Demharter

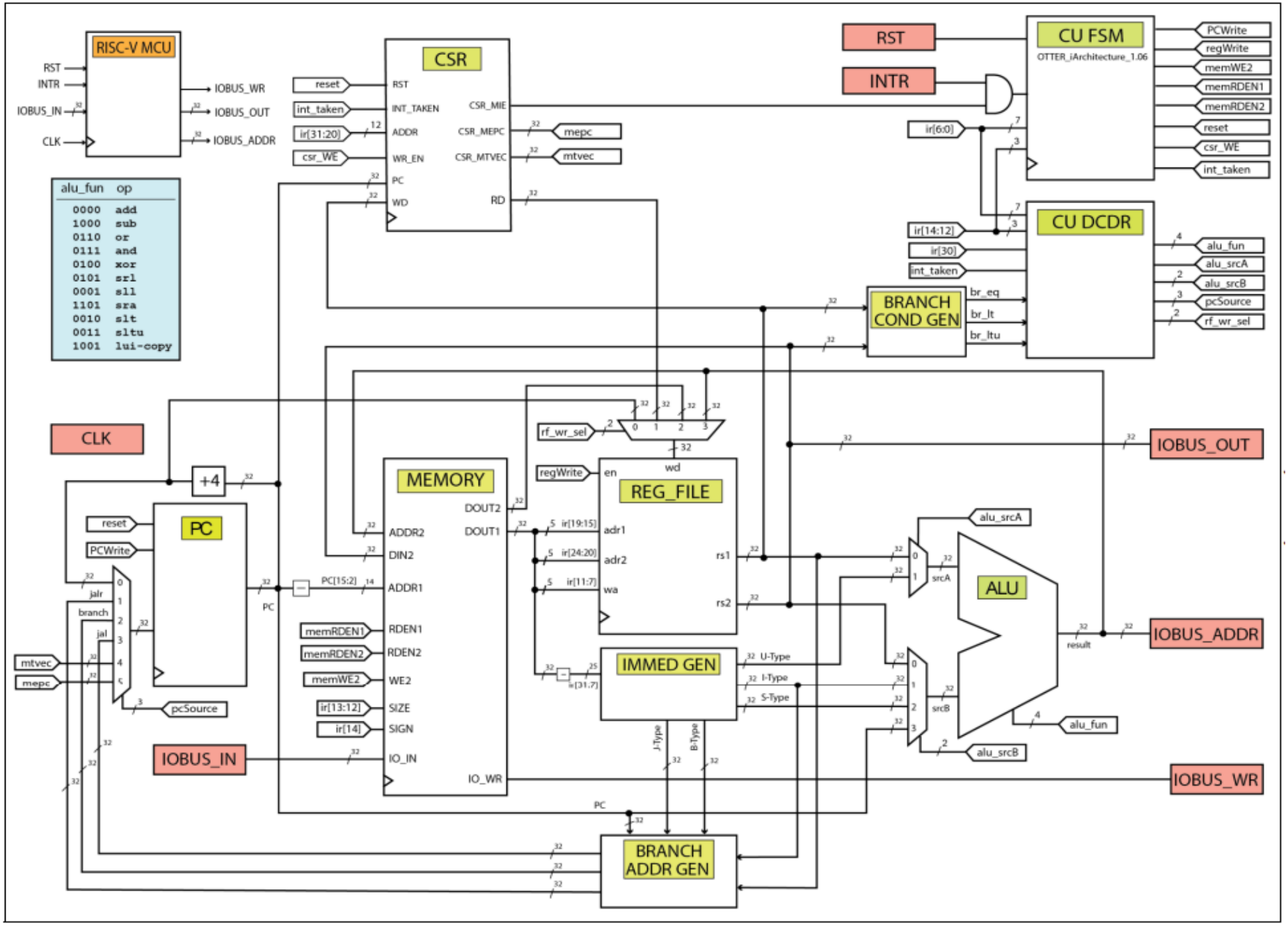
Summary:
Implemented a DeBouncer and One-Shot module in the hardware that asserted the interrupt signal for an appropriate amount. Programmed an assembly code that counted how many times an interrupt was pressed up until 49 and used a multiplexed seven segment with lead zero blanking to display the interrupt count.

Demo: https://youtu.be/nvTc9bCn8oI

Schematics:

Higher Level Schematic




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
alu_fun
alu_srcA
alu_srcB
pcSource
rf_wr_sel
BRANCH COND GEN
br_eq
br_lt
br_ltu
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
DOUT2
DOUT1
ADDR2
DIN2
ADDR1
memRDEN1
memRDEN2
memWE2
ir[13:12]
ir[14]
IO_IN
IO_WR
RDEN1
RDEN2
WE2
SIZE
SIGN
IOBUS_IN
rf_wr_sel
regWrite
REG_FILE
wd
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
IOBUS_OUT
alu_srcA
srcA
ALU
srcB
result
alu_fun
alu_srcB
IOBUS_ADDR
IOBUS_WR
BRANCH ADDR GEN
PC

Source Code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:  Danny Dang, Dylan Sandall, Felix Demharter
//
// Create Date: 11/14/2018 02:46:31 PM
// Design Name:
// Module Name: OTTER_Wrapper
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Otter Wrapper: With the addition of the Debouncer and Oneshot modules
//
// Dependencies:
//
// Revision:
//
//////////////////////////////////////////////////////////////////////////////////

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

   //- 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

   //- Signals for connecting OTTER_MCU to OTTER_wrapper
   logic s_interrupt;
   logic s_reset;
   logic s_clk = 1'b0;            // divided clock

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

   //- 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)
```

```
 // Signals for the DeBouncer
 logic DB_out;

 // Signals for the One-Shot
 logic pos_pulse_out;
 logic neg_pulse_out;


assign s_interrupt = buttons[4];  // for btn(4) connecting to interrupt
assign s_reset = buttons[3];      // for btn(3) connecting to reset

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

 DBounce #(.n(5)) my_dbounce(
     .clk    (s_clk),
     .sig_in (s_interrupt),
     .DB_out (DB_out)   );

 one_shot_bdir  #(.n(3)) my_oneshot (
     .clk           (s_clk),
     .sig_in        (DB_out),
     .pos_pulse_out (pos_pulse_out),
     .neg_pulse_out (neg_pulse_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];
       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;
         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
```

```
# ---------------------------------------------------
# ISR: buttoncntr - exp 8
# ---------------------------------------------------
.data
sseg:         .byte 0x03,0x9F,0x25,0x0D,0x99,0x49,0x41,0x1F,0x01,0x09,0xFF
              # LUT for 7-segs (includes blank char)

.text         li     x8, 0x1100C008      # x8 = AN
              li     x9, 0x1100C004      # x9 = SEG
              li     x10, 10
              li     x11, 5
              la     x13, sseg           # get address of first sseg
              li     x24, 1

              la     x5, ISR             # get ISR addr
              csrrw  x0, mtvec, x5       # load ISR addr

wait:         csrrw    x0, mie, x24      # enable interrupts

set10:        mv     x5, x18
              bnez   x5, nz              # if zero:
              li     x5, 10              # set to char10 (blank)
nz:           add    x5, x13, 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

              call    delay

set1:         add    x5, x13, 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

              call     delay

              j    wait                  # repeat

# ---------------------------------------------------
# ISR: buttoncntr - exp 8
# upon interrupt, increments counter (0-49) that is stored as a 2 digit BCD
# in registers x18 and x19
# ---------------------------------------------------
ISR:   addi         x19, x19, 1          #     increment 1's
       blt          x19, x10, ISRret     #     if x19 is 10 or higher:
       addi         x18, x18, 1          #     add one to tens
       mv           x19, x0              #     clear ones
```

```
       bne           x18, x11, ISRret     #     if reached 50:
       mv            x18, x0              #     clear
       mv            x19, x0              #     clear



ISRret:       mret                         #     return

# ----------------------------------------------------
# SUBROUTINE: delay - exp 8
# tweaks x7
# ----------------------------------------------------
delay:        li            x7, 999
dela:         addi          x7, x7, -1
              bgez          x7, dela
              ret
```

Questions:

1. Here is a quote from this experiment: “If you write your program in such a way that your program requires a stack”… So you often have a choice of when you use a stack or not. Briefly explain when your program requires a stack or not.

Your program will require a stack if you call subroutines recursively, or wish to save context conveniently. Most concepts can probably be done without the stack, but making use of the stack is a handy construct for storing data in memory, and can be used recursively and consistently. If your program is very simple, you may not need a stack, but for anything requiring lots of nested code, it’s very convenient.

2. You used a debounce module in this experiment. This module won’t catch all possible signals, however. What is the minimum signal duration that this debounce is guaranteed to pass along as a valid signal (as opposed to noise)? State your answer in system clock cycles. Fully explain your answer; you’ll need to look at the Verilog model for the debounce to answer this question.

The minimum signal duration is 8 clock cycles. The MCU will see eight 0 to 1 transitions and will only take into account the final 0 to 1 transition after 8 clock cycles, given enough time for the interrupt signal to be asserted.

3. Briefly but completely explain why, in general, keeping your ISRs are short as possible is the best approach.

An ISR should be as short as possible so that it is less likely that another interrupt will be missed when it is executing. If the interrupt is too long we could end up having an ISR that would take a much longer time to execute than the non-interrupt code which would cause huge delays.

4. We often consider the ISR code as having a “higher priority” than the non-interrupt code. Briefly but fully explain this concept.

It is best to handle interrupts quickly (with priority, as well as low processing time), as masking interrupts for long periods can lead to longer response time, and higher chance of subsequent interrupts being missed. If the ISR is quickly processed, and interrupts are reenabled, this is not as much of a problem.

5. If you were not able to use a LUT for this experiment, briefly but completely describe the program you would need to write in order to translate a given BCD number into its 7-segment display equivalent.

Going from BCD to binary (for each digit, done separately) would be done the same way as with a LUT. Next you need to convert a binary number to its visual representation on the 7 seg. You could go about this in many ways. One is writing a case/switch statement, where the binary number is checked in a series of beq statements, and output values are set according to the case.

6. What is the minimum execution time that an ISR requires? This is the time from when the MCU switches to the interrupt state to when the processor begins executing the next intended instruction, which was scheduled to be executed before the interrupt processing started. Provide a concise answer with adequate description. Assume the interrupts return from the ISR in the disabled state.

It takes five clock cycles to go through the ISR which requires the interrupt state, fetch 0x3FF, execute 0x3FF, fetch ISR, and execute the ISR.

7. As you know, part of the interrupt architecture includes the RISC-V MCU automatically masking the interrupts. This being the case, why then is there a need to keep the overall output pulse length of the interrupt relatively short?

The MCU will only mask interrupts for long enough to complete the ISR, and return to the main code where MIE is set back to 1. If the pulse length of the interrupt is longer than the time the interrupts are masked, a single interrupt will be received, masked, unmasked, and received again.

8. Nested subroutines are quite common in assembly language programming. Accordingly, there is also the notion of nested interrupts. The RISC-V OTTER does not support nested interrupts, but if you wanted to break your program by allowing interrupts to nest, how would you do that under program control on the RISC-V MCU?

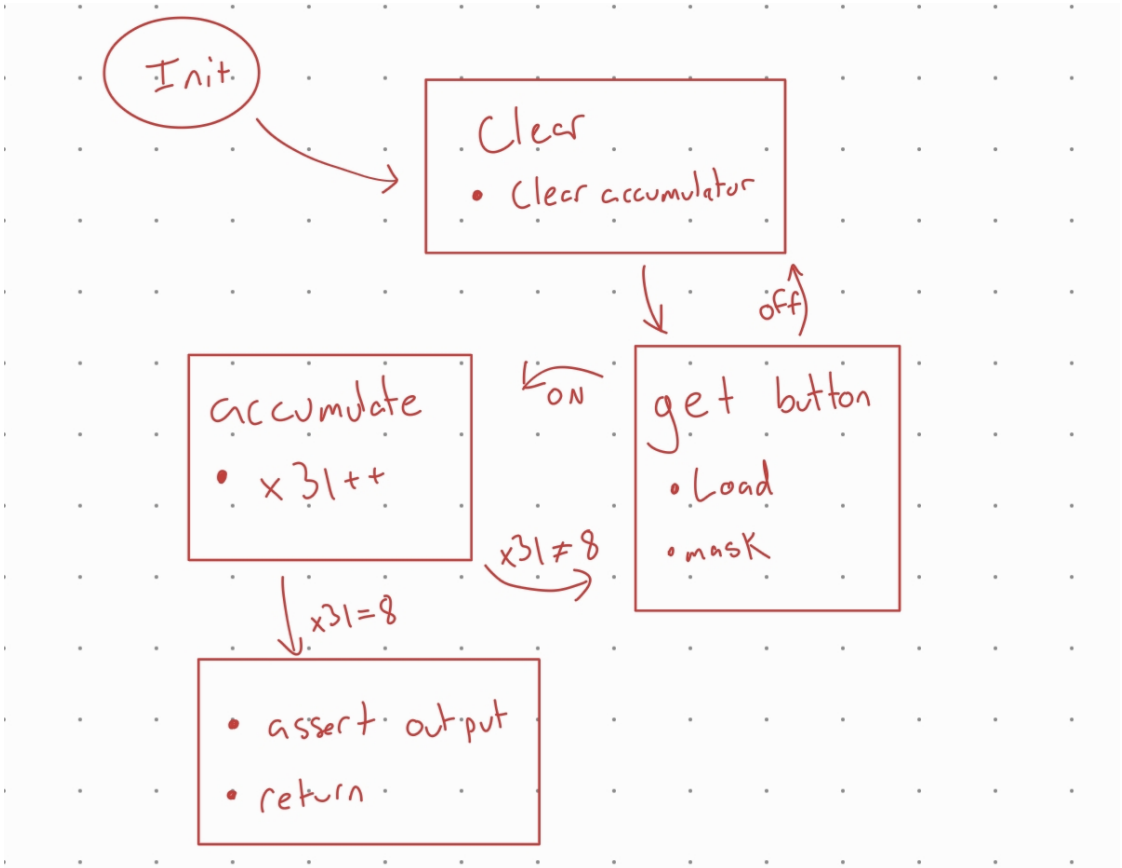
In order to allow interrupts to nest, you simply need to enable interrupts within the ISR. This can be done using “csrrw x0, mie, x24” where x24 = 1.

9. Write some RISC-V MCU assembly code that would effectively implement a firmware debounce for a given button. For this problem, assume the button you’re debouncing is associated with the LSB of buttons addressed by port 0x1100_8004. Make sure to fully describe your code with instruction comments and header comments. Also include a flowchart that models your code.

```
# ------------------------------------------------
# Subroutine: debounce - exp 8
# This subroutine handles polling and debouncing of a button at the LSB of IO address 0x11008004.
# The button must be on for 5*(x8) clock cycles, and is checked (x8) times in this period.
# ------------------------------------------------
debounce:
        li      x7, 0x11008004 # button address
        li      x8, 8          # 8 = req length
        li      x9, 0          # clear output

clear:  li      x31, 0         # init x31
loop:   lbu     x5, 0(x7)      # get button data
        andi    x5, x5, 1      # mask LSB
        beqz    x5, clear      # if button is off, clear and continue polling
        add     x31, x31, x5   # accumulate
        bne     x31, x8, loop  # loop if button has not been on for req length

        li      x9, 1          # assert output
        ret
```



10. You wrote assembly code for this experiment. Briefly but completely explain whether this code was software, firmware, or Tupperware.

Definitely tupperware. The food storage capability of our code is unmatched.
Firmware - the assembly code written for this experiment will only work on the RISC-V Otter configured the same way we have it here. This code will not run on an arduino, and requires a board with a 7 seg module that is hooked up to the same I/O ports we chose in the code.

11. State whether the RISC-V MCU is a RISC or CISC processor. Support your statement with at least three characteristics regarding those two types of computer architectures.

The RISC-V MCU is an RISC as its name implies that it is a Reduced Instruction Set Computer. The RISC-V has the same number of clock cycles where all instructions are executed in two clock cycles with fetch and execute. It also has only simple arithmetic operations, logical bit operations, storage, program control, I/O which all only accomplish one basic task. It has a large register file with 32 register spaces instead of the CISC architecture that would have only 8 register spaces.

12. I wrote an interrupt service routine that always calls a single subroutine, though it calls that subroutine multiple times in a non-nested manner. Briefly but fully explain whether I should store the return address before my ISR makes those subroutine calls.

You shouldn't have to store the return address in this case. Saving the return address is critical for nested subroutines, as any subroutine called past the first will overwrite the first's return address. This is because they both use x1 to store the address. The ISR does not use x1 to store its return address, and will not be overwritten by the subroutines it calls. If it were to call a nested subroutine, the first subroutine it calls should store the return address in memory.

13. List at least two reasons why lead zero blanking is a great idea to use, particularly in embedded systems.

1. Lead zero blanking looks great, and more closely matches the way we write numbers as humans.
2. It saves inputs as each anode requires 8 pins when displaying so it would be more efficient to turn the anodes off when not needed.

Programming Assignment:

Write an interrupt driven RISC-V MCU assembly language program that does the following. The program outputs the most recent value on the switches to the LEDs when the MCU receives an interrupt. If the program outputs the same switch value on two consecutive interrupts, the MCU stops outputting switch values until BUTTON(0) (the LSB on the BUTTON input) is pressed (button press = 1), at which point the MCU returns to normal processing. Don't worry about button debounce in firmware.

- Don't input any data in the interrupt service routine
- Strive to keep your ISR as short as possible
- Assume the same switch values never appear more than two consecutive interrupts
- Assume an external device generates the interrupt
- Minimize the number of instructions in your solution
- Use the following I/O port addresses:

```
.EQU    LEDS,     0x1100_C000
.EQU    SWITCHES, 0x1100_8000
.EQU    BUTTONS,  0x1100_8004
```

Description of Algorithm

The below code is designed to do the following. After first performing necessary setup tasks, the program waits for an interrupt. Once an interrupt is received, the current status of the switches are compared to the previous reading. If they are equal, the MCU polls for a button press. If they are not equal or a button press is received, the current status of the switches is pushed to the LEDs, and the program resumes waiting for an interrupt.

Assembly Code:

```
# ---------------------------------------------------
# SUBROUTINE: readbtn  - exp 8
# ---------------------------------------------------
readbtn:
    li          x6, 0x11008004   #button address
    li          x9, 0x11008000   #switch address
    li          x8, 0x1100C000   #LED address
    lhu         x10, 0(x9)       #load initial switch position
    la          x7, ISR          #ISR address
    csrrw       x0, mtvec, x7    #into mtvec

sit:
    li          x27, 1           #set x27
    csrrw       x0, mie, x27     #enable interrupts
wait:
    bnez        x27, wait        #wait for interrupt
    lhu         x5, 0(x9)        #load current
    bne         x5, x10, ne      #if current==previous:

btnchk:
    lbu         x7, 0(x6)        #get button status
    andi        x7, x7, 1        #mask to LSB
    beqz        x7, btnchk       #repeat until it is received

ne: mv          x10, x5          #previous <-- current
    sh          x5, 0(x8)        #LEDs <-- current

    j           sit              #sit and wait

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

Hardware Design Assignment:

You must modify the RISC-V MCU such that the mret pseudoinstruction automatically unmasks the interrupts. Describe which RISC-V MCU modules need to change and what changes those modules require. Show the code you change as part of this solution. This is one of those problems that requires only a few minutes after you think about the problem for a while. You're going to need to look at the CSR module to solve this problem.

Answer:
In order to enable interrupts, MIE must be loaded with 1. This can be done by tweaking the CSR control lines ADDR, WR_EN, and WD to 304 (the address of MIE), 1, and 1. It can also be done in Verilog, by loading a 1 into MIE every time a return address is loaded from MTVEC.

```
module CSR();

    //- read from registers
       always_comb
       case(ADDR)
            MTVEC:
            begin
                RD = CSR_MTVEC;
                CSR_MIE = 1'b1;
            end
            MEPC:    RD = CSR_MEPC;
            MIE:     RD ={{31{1'b0}},CSR_MIE};
            default: RD = 32'd0;
       endcase

endmodule
```