Justin Carlson

3/12/2023

Prof. Mealy

CPE 233-07

Experiment 8: Using Interrupts on the RISC-V MCU

Executive Summary:

In this experiment, a button debouncer and oneshot hardware were added to the wrapper in order to start using interrupts in RISC-V without causing hardware problems. To test this functionality, a test program was written in RISC-V using a BCD count triggered by MCU interrupts.

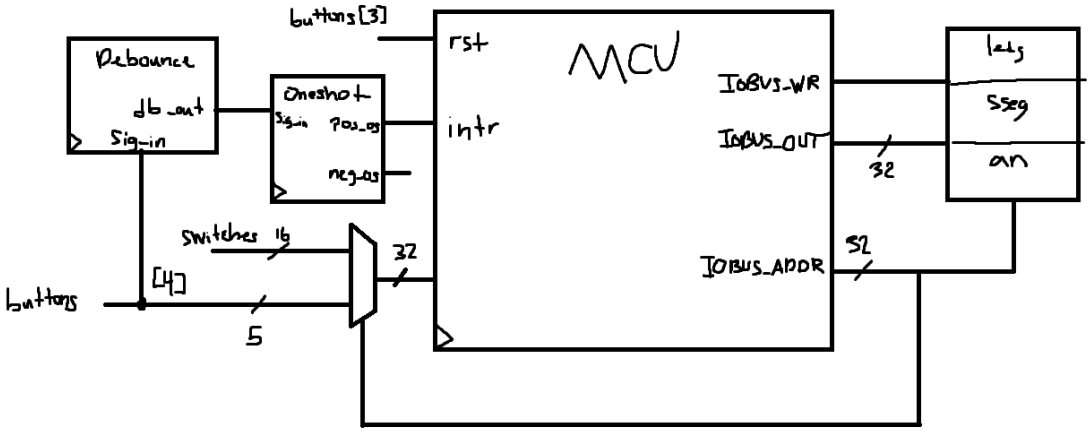
Experiment Demo: https://youtu.be/SMlfEZMapIs

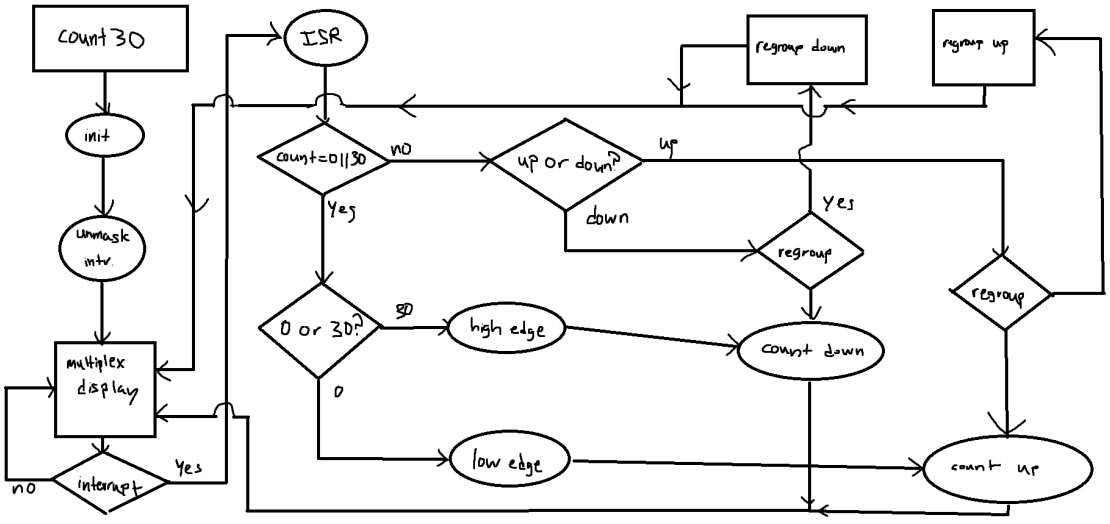
Wrapper Used: OTTER_Wrapper 1.07

RISC-V Code for Multiplexing:

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

.text
main:
init:   li     x15,0x1100C004  # seg port
        li     x16,0x1100C008  # an port
        li     x17,7           # ones anode display code
        li     x18,11          # tens anode display code
        li     x19,15          # blank anode display code
        li     x29,3           # used to check if count should be reversed
        li     x22,9           # used to check if regrouping is needed
        mv     x31,x0          # count tens place
        mv     x30,x0          # count ones place
        mv     x11,x0          # interrupt status flag for display
        mv     x20,x0          # count direction flag
        la     x5,sseg         # LUT address
        la     x12,ISR         # load address of ISR into x12
        csrrw  x0,mtvec,x12   # store address as interrupt vector CSR[mtvec]

unmask: li     x10,0x8         # set bit[3] value in x10
        csrrs  x0,mstatus,x10 # enable interrupts: set MIE in CSR[mstatus]

loop:  sw      x19,0(x16)     # clear anodes
       add     x8,x5,x30      # find absolute address of sseg data
       lb      x8,0(x8)       # load sseg data
       sb      x8,0(x15)      # display sseg
       sw      x17,0(x16)     # enable anodes for ones
       call   delay           # run delay to brighten display
       sw      x19,0(x16)     # clear anodes
       add     x8,x5,x31      # find absolute address of sseg data
       lb      x8,0(x8)       # load sseg data
       sb      x8,0(x15)      # display sseg
       sw      x18,0(x16)     # enable anodes for tens
       call   delay           # run delay to brighten display
       li      x10,0x8        # set bit[3] value in x10
       csrrs  x0,mstatus,x10  # enable interrupts: set MIE in CSR[mstatus]
       j loop                 # wait for interrupt

delay:         li      x23,0xFF       # load count
dloop:         beq   x23,x0,exit      # leave if done
               addi  x23,x23,-1       # decrement count
               j       dloop          # rinse, repeat
exit:          ret

#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
#- ISR
#-------------------------------------------------------------------------------
ISR:           beq     x31,x29,highedge # if interrupts is 30, reverse direction of count

               or      x7,x30,x31     # ors to see if bcd count is zero
               beqz    x7,lowedge     # if interrupts is 0, reverse direction of count

               beq     x20,x0,cntup   # checks counter direction flag to either count up or down
               j cntdown

highedge:     li  x20,1              # if interrupts is 30, count down instead
              j   cntdown

lowedge:      mv  x20,x0             # if interrupts is 0, count up
              j cntup
```

```
cntup: beq     x30,x22,regroupup  # check if decimal value needs regrouping
       addi    x30,x30,1          # add one to ones
       j       done               # return to loop

cntdown:  beq     x30,x0,regroupdown # check if decimal value needs regrouping
          addi    x30,x30,-1          # subtract one from ones
          j       done                # return to loop

regroupup: mv       x30,x0           # clear ones register
           addi     x31,x31,1        # increase tens register
           j        done             # return to loop

regroupdown: mv      x30,x22         # sets ones register to 9
             addi    x31,x31,-1      # decrease tens register
             j       done            # return to loop

done:   mret                         # return from interrupt
#-------------------------------------------------------------------------------
```





Figures 1 & 2. (Wrapper Schematic & Program Flowchart)

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 what determines whether your program requires a stack or not.

A stack is absolutely required for programs that need to call other subroutines, as the stack stores information to restore to the location after call instructions. If you don’t require subroutines to be called, then you don’t need a stack for your code to work.

2. You used a debounce modules 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.

It took at a minimum 4 clock cycles to get a clear, debounced output. The flip flops r_dff1 and r_dff2 take one and two cycles respectively from the interrupt being asserted until both of them are on. When both are on, the s_reset turns off which allows the r_count to become 1 after another clock cycle. Finally, the output flip flop is triggered high after one more clock cycle where the high r_count and r_dff2 are both 1 which assigns the debounced output to be one, hence taking four total cycles.

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

Longer ISRs decrease the response time of the system, so shortening them ensures that interrupts will not be delayed and other interrupt signals won’t be missed.

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

The entire point of the ISR is that the programmer might need to run a subroutine at any given point in time, and the interrupt gives them the ability to stop whatever they were doing and run the ISR code.

Since us as the programmer is willing to pause whatever code we were previously running and run the ISR, it makes sense that we treat it with a higher priority compared to whatever was being done before.

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.

Without a LUT, you would have to use a K-map to generate the boolean expressions for each cathode on the display. Then, these expressions need to be simplified into logic gates like ANDs and ORs that can be handled in a RISC-V program.

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.

Once the ISR entered, in order to return from the ISR with interrupts masked, we must use a load instruction to fill a register with a value for mstatus. Then, one of the CSR write instructions must be used to clear the mpie bit in the CSR status register. Lastly, the ISR must be returned from with the mret instruction, so in total it takes three instruction cycles.

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?

If the ISR is really short as it's meant to be, then it's possible that with too long a pulse length the ISR can run again after the ISR returns and interrupts are re-enabled. Having the pulse be short enough along with the hardware interrupt masking provides multiple levels of protection that the ISR is run the desired amount of times.

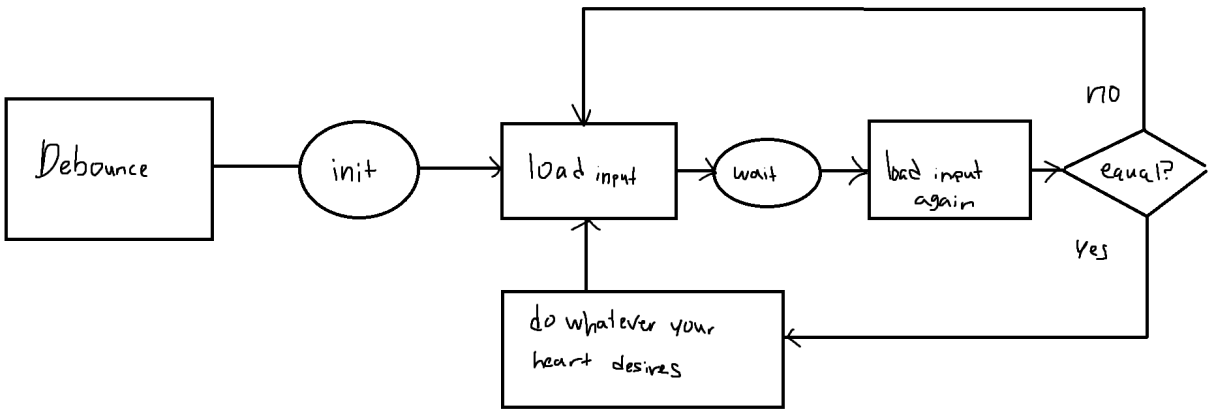
8. Nested subroutines are quite common in assembly language programming. Accordingly, there is

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?

If you wanted nested interrupts on the MCU, you would just have to re-enable interrupts while inside the ISR.

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.

```
init: li x10,0x1100_8004    # Buttons

loop: lb x20,0(x10)          # look at first button input
      nop                    # wait for a given time
      lb x21,0(x10)          # load another button input
      bne x20,x21,loop       # compare the two inputs and if they are the same, then the input
is debounced otherwise return to polling loop

      # do whatever you desire here
      j loop                 # re-enter polling loop
```



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

After writing it, I'm seriously inclined to think I did in fact just write "tuppaware". On a more serious note, this experiment sort of involved both software and firmware depending on your interpretation. The firmware could be identified as the display multiplexor that drives the Basys board to work given our hardware. The counting sequence itself was more software related since it involved an arbitrarily given problem. We could do a counting sequence in an infinite number of ways but we can only create a display multiplexor that does its one job.

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 MCU is certainly a RISC processor, as it's been designed to run the RISC-V instruction set. Our MCU runs much more simple/barebones instructions than those in the CISC ISA, instructions in RISC take only one clock cycle to complete (on execute cycles, except for load instructions), and CISC has less of a memory footprint since complicated operations can be fit into a single instruction.

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.

Yes, it's completely fine to run subroutines inside the ISR as long as the return address for each call is specified in the stack. Returning from a subroutine inside an interrupt would behave the exact same way as returning from a normal subroutine, and we would need the return address to backtrack to our previous PC value.

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

Lead zero blanking is useful in that no unnecessary data is displayed to the user that would be useless to the user. Without lead zero blanking, then the display multiplexor would have to drive each anode all the time which would be inefficient and useless if a limited number of possible anodes are being utilized. In embedded systems where clock cycles are incredibly important, we'd be effectively saving time by not displaying this unimportant information.

14. If the RISC-V Otter MCU hardware did not automatically mask the interrupt, it would be problematic to do so under program control. Briefly describe how you could “add some hardware” to the RISC-V Otter MCU that would ensure that not disabling interrupts upon acting on an interrupt (and thus relying on disabling interrupts under program control) could actually work.

If we put a register or another synchronous element in front of the interrupt input, then we could ensure that we can delay additional interrupts long enough to mask the interrupt inside the program.

Hardware Problem:

You must modify the RISC-V MCU such that it supports two new instructions.

```
mret_m      # interrupts are masked when instruction executes
mret_u      # interrupts are unmasked when instruction executes
```

For this problem, describe the following:

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

Since these instructions behave very similarly to the csrrw instruction to mask and unmask interrupts, we can use its functionality in adding these new instructions, but it will require some modifications. Since csrrw requires a source register, our new instructions will have to load the proper bits into a register with an li instruction before it can be used in the csrrw instruction. Since we're effectively doing two instructions, we'll need another FSM state to handle both the load to a register then the CSR write.

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

In our interrupt-type opcode, we'll have to dedicate two new func3 values for these instructions then bake their functionality into the assembler so that they can be used in the MCU.

c) changes in RISC-V MCU memory requirements

The PC, memory module, and CSR were not changed for these instructions. Since we need a free register to handle the load part, we require 32 bits of register space. The call pseudoinstruction uses the x6 register to jump and link, and this instruction would also require the extra register in this manner. We also added an FSM state to the design to handle the two necessary operations, but since we didn't have to increase the bit width of the FSM parameters, the FSM still occupies the same amount of memory.

d) why this modification would or could be useful

Rather than manually having to keep track of the mstatus register and do the CSR write operations, having instructions to mask and unmask interrupts makes the code cleaner and saves the programmer a little bit of time.

Programming Problem:

Write an interrupt driven RISC-V MCU program that drives 8 LEDs. Each time there is an interrupt, the program reads a 4-bit value from the input port. After reading 16 values (16 interrupts), the program averages those values and outputs the LED pattern corresponding to the average to the LEDs. The LEDs thus represent a visual representation of the average of the value read from the input when each time there was an interrupt. Assume an external device connected to the RISC-V MCU generates an interrupt and provides an input value for reading. You must use the provided LUT for this problem. The dark circles in the diagram below represent the "on" LEDs. Note the difference in average value and the patter on the LEDs that need to be output.

- LED port address = 0x1100C000
- Input port address = 0x11004444
- Don't use I/O instructions in the interrupt service routine
- Minimize the number of instructions in your solution

```
.data                 # 16 total values
my_lut:   .byte       0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x08, #LED output patterns
       .byte  0x0C, 0x0E, 0x0F, 0x10, 0x18, 0x1C, 0x1E, 0x1F

.text
main:
init:   li     x14,0x11004444  # input port
        li     x15,0x1100C000  # led port
        mv     x31,x0          # number of interrupts done
        li     x16,16          # constant value to compare against number of interrupts
        mv     x20,0           # set accumulator
        la     x5,my_lut       # load address of LUT
        la     x6,ISR          # load address of ISR into x6
        csrrw  x0,mtvec,x6     # store address as interrupt vector CSR[mtvec]

unmask: li     x10,0x8         # set bit[3] value in x10
        csrrs  x0,mstatus,x10  # enable interrupts: set MIE in CSR[mstatus]

loop:   lb     x21,0(x14)      # loads switch value while waiting for interrupt
        beq    x8,x0,loop      # wait for interrupt

        beq    x31,x16,average # if interrupts is 16, go to averaging part
        mv     x8,x0           # clear flag
        add    x20,x20,x21     # accumulate input value

        csrrs  x0,mstatus,x10  # enable interrupts: set MIE (bit3) in CSR[mstatus]
        j      loop            # return to loopville

average: csrrw   x0,mstatus,x0   # disable future interrupts
         srli  x20,x20,4       # divide by 16
         add   x8,x5,x20       # find address for LUT
         lb    x8,0(x8)        # load led value
         sb    x8,0(x15)       # display leds
         ret                  # done


#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
#- The ISR: sets bit x8 to act as flag to task code.
```

```
#-------------------------------------------------------------------------------
ISR:    li    x8,0x1         # set flag to 1
        addi   x31,x31,1     # increments interrupt counter by 1

        li    x9,0x80        # set bit7 in x9
        csrrc  x0,mstatus,x9   # clear bit7 (MPIE) in CSR[mstatus]

done:   mret                 # return from interrupt
#-------------------------------------------------------------------------------
```