Justin Carlson

3/19/2023

Prof. Mealy

CPE 233-07

Experiment 9: MCU Timer Counter

Executive Summary:

This experiment saw the implementation of the timer counter module to help assist our firmware. The display multiplexing from the last experiment was redone with interrupts being triggered on a timer rather than an IO signal. Rather than use a hardware debounce and oneshot, a firmware debouncing solution was implemented.

Experiment Demo: https://youtu.be/qY_JxTvuqD8

Wrapper Version: Otter_Wrapper_TC_Driver

Assembly Code:

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

.text
main:
init:   li     x12,0x11008004  # button port
        li     x13,0x1100C004  # seg port
        li     x14,0x1100C008  # an port
        li     x10,0x1100D000  # CSR
        li     x11,0x1100D004  # TC In
        li     x15,7           # Ones anode code
        li     x16,11          # Tens anode code
        li     x17,15          # Disabled anode code
        la     x5,sseg         # Seven-seg LUT
        la     x6,ISR          # mtvec address
        li     x29,3           # used to check if count should be reversed
        li     x22,9           # used to check if regrouping is needed
        mv     x20,x0          # Direction of count (0 for up, 1 for down)
        csrrw x0,mtvec,x6     # store address as interrupt vector CSR[mtvec]
        li     x7,0x8FF        # delay for TC in
        sw     x7,0(x11)       # load delay to TC in
        li     x7,1            # TC CSR without prescale
        sw     x7,0(x10)       # Enable TC

unmask: li     x8,0x8          # Flip mie to 1

loop:   csrrw x0,mstatus,x8   # Enable interrupts after returning from ISR
        lb     x18,0(x12)      # Loads button data
        bnez   x18,debounce    # If button actuated, go to debounce
        j loop

debounce: lb   x18,0(x12)      # Checks initial button press state
        call   delay           # Waits ~5ms
        lb     x19,0(x12)      # Checks button state again
        bne    x18,x19,count   # If not equal, button has been pressed
        j      debounce        # Else debounce again

delay:  li     x23,0x6FFFF      # load delay count for debounce
dloop:  beq    x23,x0,exit      # leave if done
        addi   x23,x23,-1       # decrement count
        j      dloop            # rinse, repeat
exit:   ret

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

        or     x28,x30,x31    # ors to see if bcd count is zero
        beqz   x28,lowedge    # if count 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      loop                  # return to loop

Cntdown: beq   x30,x0,regroupdown    # check if decimal value needs regrouping
        addi    x30,x30,-1           # subtract one from ones
```

```
        j       loop                # return to loop

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

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


#-------------------------------------------------------------------------------
#- ISR
#-------------------------------------------------------------------------------

ISR:    beq     x7,x0,tens          # If tens anode needs to be driven, drive it

ones:   sw      x17,0(x14)          # Disable anodes
        add     x9,x5,x30           # Find address for sseg
        lb      x9,0(x9)            # Load sseg byte
        sw      x9,0(x13)           # Display sseg value
        sw      x15,0(x14)          # Enable ones anode
        xori    x7,x7,0x1          # Toggles flag to display tens next if necessary
        j leave

tens:   beq     x31,x0,ones         # If tens = 0, drive ones instead for lead zero blank
        sw      x17,0(x14)          # Disable anodes
        add     x9,x5,x31           # Find address for sseg
        lb      x9,0(x9)            # Load sseg byte
        sw      x9,0(x13)           # Display sseg value
        sw      x16,0(x14)          # Enable tens anode
        xori    x7,x7,0x1          # Toggles flag to display ones next

leave: mret
```

Hardware Schematic:

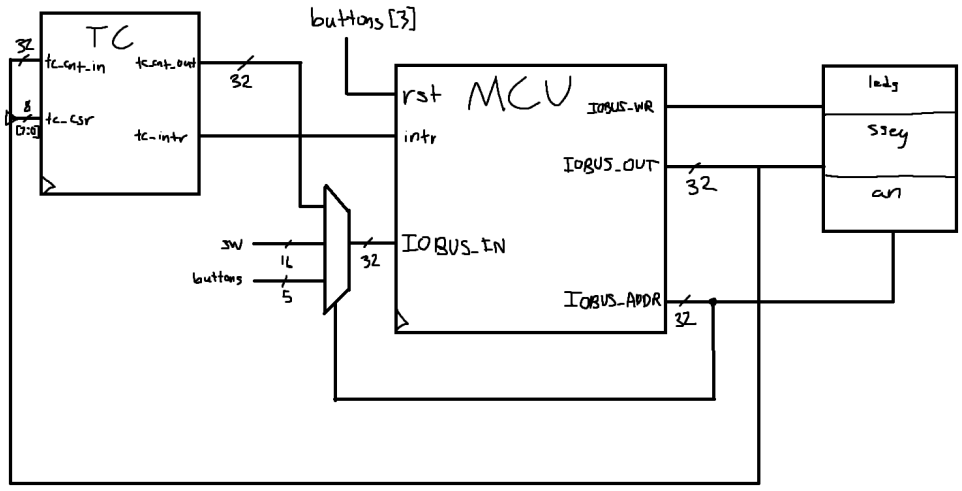
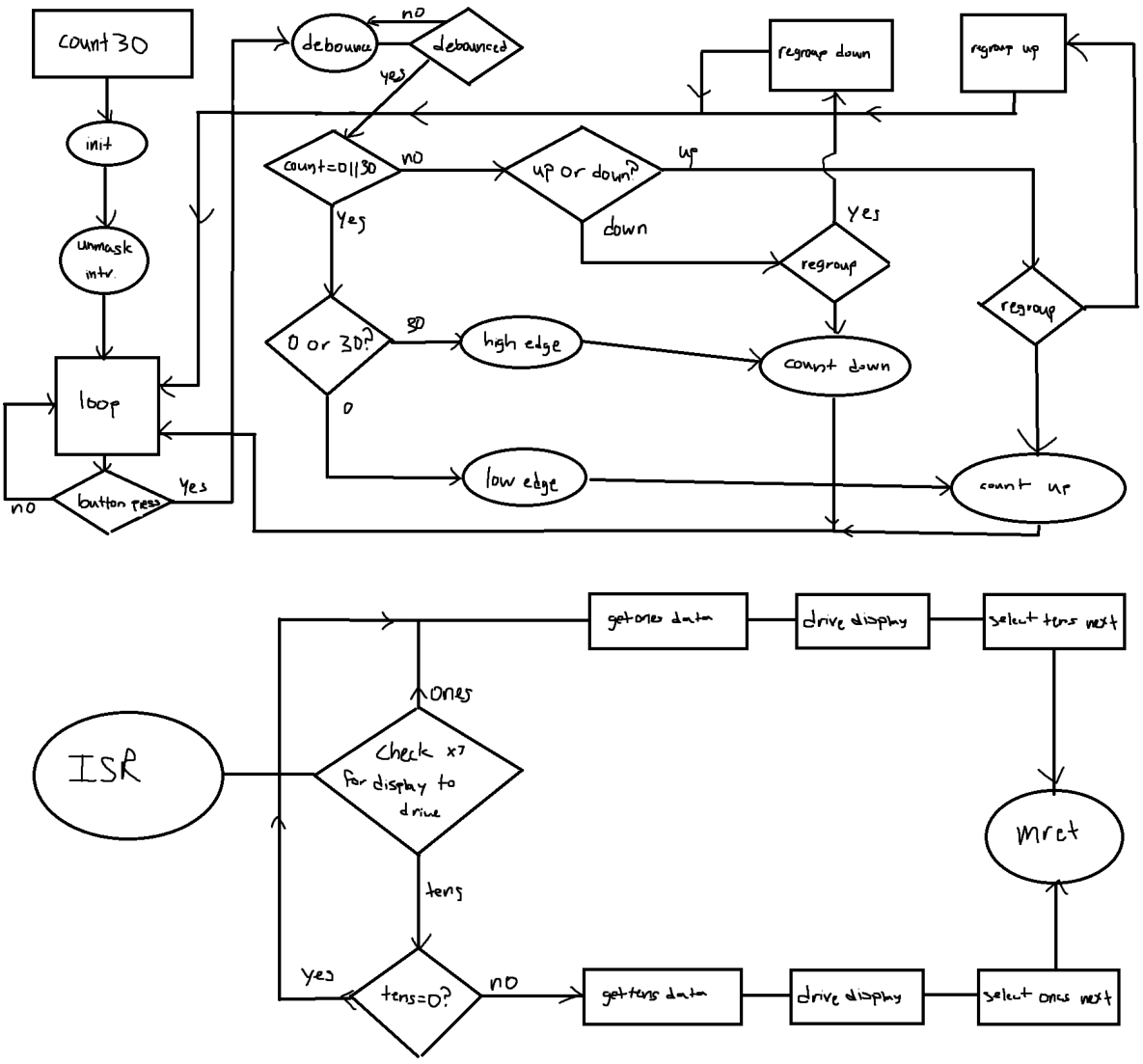


Figure 1 (MCU Schematic with TC)



Figures 2 & 3 (Assembly Flowchart & ISR Flowchart)

Rationale for TC Settings: 0x8FF was chosen for the TC_in value, and the TC prescale was 0. We don't need a prescale more than zero since our timer count does not exceed the 32-bit capacity. You told me in class that 0xFF was an acceptable delay for the display multiplexing, but this didn't work for my case. The displays slightly ghosted with the value on the other segment, and this problem was fixed by multiplying the delay by 9. According to the simulation, the delay is for about 164 microseconds which leads to a frequency of 6kHz. Although this is certainly higher than the display needs to function properly, I had no reason to change it further for this experiment.

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.

Since delay loops are incredibly inefficient as they occupy the MCU’s processing capabilities when running, having a passive component to do the counting for you such as the timer-counter frees up the MCU to do more meaningful things.

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

The hardware will automatically mask the interrupts upon entering the interrupt state to prevent nested interrupts. Our MCU can be changed to not have this behavior and it can be done under program control inside the ISR instead, as long as the interrupt pulse remains a certain width. Unmasking the interrupts can only be done under program control, as we first need to load the mtvec register and enable the mie bit manually before we can use interrupts.

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.

When enabled, the timer counter will increase by one for every clock edge. The tc_in input designates how high the count should go to before it gets reset. This is a 32-bit value completely configurable to the user. The prescaler can also be used to divide it even further by waiting a specific number of clock cycles before it increments the internal count, which is controlled by the upper 4-bits in the tc_csr register. Once the internal count matches the tc_in value, the timer-counter is reset to zero and it pulses the tc_intr output for n pulses based on the parameter.

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.

Given a 10ms interrupt and assuming the interrupt toggles the LED for each interrupt, the frequency is going to be 1/10ms * 0.5 which is 50Hz. Since a full cycle is considered turning on and turning back off the LED, a full cycle needs two interrupts to complete hence the 0.5 factor.

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 prescaler is a 4 bit value, and based on whatever it is the TC will only increase the internal count after the corresponding amount of clock cycles based on the prescale. This effectively means we can divide our timer-counter by any whole number from 1 to 16. The only constraint is that we can’t do decimal clock divisions or anything greater than 16.

6. Changing frequencies of the timer-counter can possibly create a timing error for one clock period. Briefly describe what two issues can causes 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.

When the timer-count value or the prescaler changes whatever value is inside the internal counter will not clear, which will lead to the count being wrong for a full TC cycle as the duration until the output pulse will not be the intended amount. After the output pulse is asserted and the TC is cleared, this problem will not happen again. The assembly related problem would be the extra instruction needed to load large values into a register. If the tc_in is greater than 12-bits, then an lui and an addi instruction need to be completed before the counter maximum is stored. This will create an error for a timer-counter cycle like the other problem.

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 TC has a slight delay when it handles prescaled values due to extra time needed to configure and read the counter value. This propagation delay is noticeable in small time measurements, but is incredibly negligible on a seconds level. If the frequency of the LED blink is fast enough though, the timer-counter will not quite be at the proper frequency.

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.

By enabling the timer-counter with the tc_csr[0] bit at the same time the pulse signals, you can wait for the pulse to end and disable tc_csr[0] at the same time. The tc_cnt_out output will tell you the duration of the pulse. A large enough tc_in value should be loaded as to not cause it to overflow. If the pulse-width exceeds the 32-bit max size, a tc_intr output can be used to increment an additional counter for the number of overflows. This second counter output can be multiplied by the tc_in size and added to the current tc_cnt_out to find the width of long pulses.

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.

With how my firmware debounce was set up, this would technically be possible, but there would be a massive delay before the interrupt signal is asserted. My debounce worked by loading the value of a button, waiting a duration of time, loading it again, and lastly comparing the two and deciding if the bouncing is done. The delay cycle took nearly 5 ms though to account for a large button bounce, and the tc_intr pulse is meant to only be a few cycles of 20 ns long. Although it would work in some cases, this might be impractical for a number of others given the debounce takes hundreds of thousands of times longer than a timer-counter pulse.

Hardware 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

Since we need to do multiple instructions, we'll have to add another execute FSM state to handle the additional arithmetic operation. Our first execute state will add the two registers as normal and store the result in the location at rs2. The second state will handle the division based on the immediate value and store it back in rs2. Since we need to connect the addr2 input of the RegFile to the wa input, we need a mux to select between ir[11:7] and the rs2 address. If our instruction is called, the FSM needs to enable a new signal to write the resulting data back to the address at the rs2 register. The handling of the ALU operations and generation of immediate values should be the same so the FSM and the mux should be the only hardware changes we need to make.

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

The assembler needs to be changed to handle our new instruction. Since the instruction takes in two source registers and an immediate value, we need to make a new opcode considering we don't have one that matches those requirements (unless we share opcodes with branch or store instructions). A func3 value must also be assigned for our new instruction.

c) changes in RISC-V MCU memory requirements

No changes were made to the PC, memory, RegFile, or CSR. We added an additional FSM state but since our current FSM size allows for 8 states and we have 5 currently, adding a new state does not occupy additional memory.

d) specifically, why this modification would be useful

If this addition and division operation needs to be done a number of times, adding this instruction helps us lose a second fetch cycle on the second instruction call improving our performance at this task.

Programming Assignment:

Write an interrupt driven RISC-V MCU assembly language program that does the following. The program outputs the average of the four most recent values that were on the switches when the RISC-V MCU received 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

```
.text
main:
init:   li     x14,0x11008000 # switch port
         li    x15,0x1100C000 # led port
         li    x16,0x6000     # memory pointer
         li     x17,0x6003     # end of memory constant
         mv    x20,0           # set accumulator
         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:  lbu     x21,0(x14)       # loads switch value while waiting for interrupt
       beq     x8,x0,loop       # wait for interrupt

       csrrw   x0,mstatus,x0   # disable interrupts temporarily
       mv      x8,x0             # clear flag
       beq     x16,x17,move_pointer  # check if memory pointer is at end of range

store: sb      x21,0(x16)      # store switch data in memory
       addi    x16,x16,1       # increment memory pointer

average:  mv   x20,x0          # clear accumulator
       li      x25,0x6000      # load initial memory pointer location

loop2:  beq    x25,x17,exit    # check if all memory values are accumulated
       lb      x28,0(x25)      # load data
       add     x20,x20,x28     # accumulate
       addi    x25,x25,1       # increment pointer
       j       loop2

exit:  srli    x20,x20,2       # divide by 4
       sw      x20,0(x15)      # display on leds
       j       unmask          # return to loop

move_pointer: li x16,0x6000
       j store


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

#-------------------------------------------------------------------------------
#- The ISR: sets bit x8 to act as flag to task code.
#-------------------------------------------------------------------------------
ISR:     li     x8,0x1        # set flag to 1

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