CPE 233 Lab #9 – Timer-Counter

Astrid Augusta Yu

June 3, 2020

Contents

1 Questions

1. **Briefly describe why it is "more efficient" to use a timer-counter peripheral to blink an LED than to use a dumb loop (delay loop) to blink an LED.**

   It is more efficient because the CPU could be doing other things instead of blinking the LED. Additionally, it would be more accurate because the counter is independent from the CPU's execution, which is not always predictable.

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

   (a) Every clock cycle it increments its 4-bit prescaler counter, r_ps_count.

   (b) When that overflows, it starts incrementing its 32-bit counter, r_counter32b.

   (c) When that reaches the target number stored in tc_cnt_in, it resets the counter and raises s_pulse.

   (d) A shift register-based pulse extender, pos_pulse_reg is used to extend that single-cycle s_pulse to 3 clock cycles. The result of that pulse extension is the output.

3. **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.**

   Assuming a 50% duty cycle made by inverting the LED's on state every interrupt, a single blink period is 2 interrupt pulses. Thus, it is $2 \cdot 10\text{ms} = 20\text{ms}$. Therefore, the maximum frequency would be $f = \frac{1}{20\text{ms}} = 50\text{Hz}$.

4. **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. For this problem, assume the timer-counter module uses the system clock.**

   The clock prescaler effectively acts like a clock divider for the system clock (we'll call the frequency f). If the prescaler is set to 2, then the new maximum clock frequency becomes $\frac{f}{2}$ and values above that are impossible to reach without readjusting the prescaler.

5. **Changing frequencies of the timer-counter can possibly create a timing error for one clock period. Briefly describe what causes this one hiccup.**

   On the clock cycle that the TC is being written to, its internal counter gets reset to 0. Thus, the TC misses the pulse of the last frequency it was set to.

6. **The timer-counter provided for this experiment provided 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 how and when this overhead can interfere with the frequency output of the blinking LED.**

   The overhead would basically be in calculating the state of the PWM output. Changing the frequency of the timer-counter causes a timing error, so the actual PWM frequency may be greater than the expected frequency.

7. **Briefly describe how you would use the timer-counter module to "time" the length of time a given signal is asserted.**

   The timer-counter/ISR needs to be enabled on or before the positive edge of the signal. Then, in the ISR, on every timer-counter interrupt:

   (a) Check if the signal is asserted

      i. If it is, increment a counter that tracks about how long it's been

      ii. Otherwise, a negative edge has been detected. Disable the ISR. The value of the counter will be proportional to the width of the signal pulse.

8. **If you tied the output of the timer-counter 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.**

   It would depend on the length of the pulse width that the timer-counter is configured to. However, in most cases, you cannot generate an interrupt this way because the debounce is a low-pass filter that removes short-width signals, while the timer-counter outputs a 2-tick signal that is too short, and gets filtered out by the debounce.

2 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 received an interrupt to the LEDs. Assume there are eight LEDs and eight switches. Interpret the eight-bit switch values are an unsigned binary number.

```
.eqv PORT_LEDS, 0x1100C000
.eqv PORT_BTN, 0x11008004
.eqv PORT_SEG, 0x1100C004
.eqv PORT_AN, 0x1100C008
.eqv PORT_SW, 0x11008000
.eqv PORT_TC_CNT, 0x1100D004
.eqv PORT_TC_CSR, 0x1100D000

.text
main:
    li s1, 0 # Value last interrupt
    li s2, 0 # Value 2 interrupt ago
    li s3, 0 # Value 3 interrupts ago
    li s4, 0 # Interrupted flag

    # Initialize LEDs
    la t0, PORT_LEDS
    sb zero, 0(t0)

    # Set ISR address
    la t0, ISR
    csrrw zero, mtvec, t0

enableInterrupts:
    # Enable interrupts
    li t0, 1
    csrrw zero, mie, t0

loop:
    beq s4, zero, loop
onInterrupted:
    li s4, 0 # Clear interrupted flag

    # Shift LED history
    mv s3, s2
    mv s2, s1

    # s1 = latest switch value
    li t0, PORT_SW
    lbu s1, 0(t0)

# Perform a linear search for max through 3 elements

    # Arbitrarily choose max value
    mv t3, s1

    # If s2 > s0 then s0 = s2
    bgt s0, s2, maxNotS2
    mv s0, s2
maxNotS2:
    # If s3 > s0 then s0 = s2
    bgt s0, s3, maxNotS3
    mv s0, s3
maxNotS3:
    # Write LEDs
    la t0, PORT_LEDS
    sb t3, 0(t0)

    j enableInterrupts

ISR:
    li s4, 1 # Raise interrupted flag
    mret
```

3 Hardware Assignment

You want to add the following instruction to the RISC-V OTTER MCU. This is a conditional return from subroutine problem when the instruction takes the return if the value in rs2 is non-zero; otherwise, the instruction has not affect other than to change the PC.

```
jalrc rs2,rs1,imm  # jump if rs2 is non-zero, rs1 & imm same as jalr
```

a. I would reuse the opcode and format of JALR for this operation, but instead of func3 = 000, JALRC would have a unique func3 (I will use 001 here).

   Additionally, I would add a “JALR condition generator” module to differentiate between the behaviors of JALR and JALRC while preserving generally the same structure inside the FSM.

   It has inputs `func3` and `rs2` as well as outputs `in_jalr` (asserted only when we are executing JALR) and `jalr_mask` (asserted only when we are executing JALR or when JALRC’s condition is true). The following truth table describes its behavior:

func3	rs2 $\neq 0$	in_jalr	jalr_mask
000 (jalr)	T	T	T
000 (jalr)	F	T	T
001 (jalrc)	T	F	T
001 (jalrc)	F	F	F

   It can be implemented in Verilog like so:

```
in_jalr = func3 == 000;
jalr_mask = in_jalr | (rs2 != 0);
```

   The CU_FSM will be modified to accept the new signals as follows:

```
case (PS)
    st_EX: begin
        pcWrite = 1;
        case (OPCODE)
            JALR: begin
>                   regWrite = in_jalr;
    ...
```

   In addition, the decoder would need to be modified as follows:

```
case (opcode)
    ...
    JALR: begin
        rf_wr_sel = 2'd0;   // next pc
>       pcSource = jalr_mask
>           ? 3'd1       // jalr
>           : 3'd0       // pc_inc
    end
    ...
```

b. The assembler would need to be modified to support jalrc and any pseudoinstructions using it, but fundamentally, not much else.

c. There wouldn't be any additional memory requirements or registers associated with this modifciation.

d. This operation can be very useful because it effectively combines two instructions into one. Using the base instruction set, this instruction can be implemented by

```
        beq x0, [rs2], notZero    # Skip over the next instruction
        jalr [rs1], [imm]
    notZero:
```

This kind of construct can be used in situations where the

4 Assembly Code

```
.eqv BTN_DURATION, 100
.eqv INC_BTN_MASK, 0x10
.eqv PORT_LEDS, 0x1100C000
.eqv PORT_BTN, 0x11008004
.eqv PORT_SEG, 0x1100C004
.eqv PORT_AN, 0x1100C008
.eqv PORT_SW, 0x11008000
.eqv PORT_TC_CNT, 0x1100D004
.eqv PORT_TC_CSR, 0x1100D000

.data
    pressedDuration: .word BTN_DURATION
    bcdData: .half 0
    digitEncountered: .byte 0
    digitIndex: .byte 0
    sseg: .space 10 # Save space for 10-byte lookup table

.text
main:
    # Load 7-segment data
    la s7, sseg
    li t0, 0x03
    sb t0, 0(s7)
    li t0, 0x9F
    sb t0, 1(s7)
    li t0, 0x25
    sb t0, 2(s7)
    li t0, 0x0D
    sb t0, 3(s7)
    li t0, 0x99
    sb t0, 4(s7)
    li t0, 0x49
    sb t0, 5(s7)
    li t0, 0x41
    sb t0, 6(s7)
    li t0, 0x1F
    sb t0, 7(s7)
    li t0, 0x01
    sb t0, 8(s7)
    li t0, 0x09
    sb t0, 9(s7)

    # Load initial digit encounter
    la t0, digitEncountered
    sb zero, 0(t0)

    # Load initial digit index
    la t0, digitIndex
    li t1, 3
    sb t1, 0(t0)

    # Load initial pressed duration
    la t0, pressedDuration
    li t1, BTN_DURATION
    sw t1, 0(t0)

    # Load initial BCD data
    la t0, bcdData
    sh zero, 0(t0)

    # Load initial stack pointer
    li sp, 0x4000

    li s0, 0 # Pressed times count
    li s1, 1 # "Redraw" flag

    # Set up TC
    li t1, PORT_TC_CNT # timer counter count port address
    li t0, 50000
    sw t0, 0(t1)

    li t1, PORT_TC_CSR # timer counter CSR port address
    li t0, 0x01 # init TC CSR
    sw t0, 0(t1) # no prescale, turn on TC

    # Enable ISR
    la t0, isr
    csrrw x0, mtvec, t0

doMainLoop:
    li t0, 1
    csrrw x0, mie, t0

    # a2 = button pressed
    li t0, PORT_BTN
    lw a2, 0(t0)
    andi a2, a2, INC_BTN_MASK

    # Debug: display outputs on LEDs
    li t0, PORT_LEDS
    sw s0, 0(t0)

    call debounceStep

shouldIncrementCount:
    beq a0, zero, shouldDraw
incrementCount:
    addi s0, s0, 1
    mv a0, s0

    # Recalculate BCD and store
    call bcd
    la t0, bcdData
    sh a1, 0(t0)
shouldDraw:
    beq s1, zero, doMainLoop
    li s1, 0
    call draw
    j doMainLoop
isr:
    li s1, 1 # Set "redraw" flag
    mret


#-----------------------------------------------------------------
#- Subroutine for debouncing and positive edges
#-
#- Arguments:
#- - a2: button currently pressed flag
#- Returns:
#- - a0: positive edge detected
#- Tweaked registers - t0, t1, t2
#-----------------------------------------------------------------
debounceStep:
    # t0 = pressed duration
    la t1, pressedDuration
    lw t0, 0(t1)

    bne a2, zero, btnPressed
btnNotPressed:
    li t0, BTN_DURATION # Reset pressed duration
    li a0, 0 # No positive edge detected
    j cleanupDebounce
btnPressed:
    # Negative pressedDuration -> posedge processed
    bge t0, zero, posEdgeNotYetProcessed
    li a0, 0 # No positive edge detected
    ret
posEdgeNotYetProcessed:
    addi t0, t0, -1

    # Has it been pressed for long enough?
    beq t0, zero, positiveEdge
    j cleanupDebounce
positiveEdge:
    li a0, 1 # Positive edge detected
    li t0, -1 # pressedDuration = -1
cleanupDebounce:
    # Store pressed duration back into memory
    la t2, pressedDuration
    sw t0, 0(t2)
    ret


#-----------------------------------------------------------------
#- Subroutine for doing the draw step
#- Tweaked registers - t0, t1, t2, t3, t4, t5, t6
#-----------------------------------------------------------------
draw:
    # t0 = index
    la t5, digitIndex
    lb t0, 0(t5)

    # t2 = bcd
    la t5, bcdData
    lh t2, 0(t5)

    # t1 = index * 4
    slli t1, t0, 2

    # t1 = digit value = (bcd << 4*index) & 0xF
    srl t1, t2, t1 # t1 = bcd << 4*index
    andi t1, t1, 0xF

isLastDigitIndex:
    # Are we rendering the right-most digit?
    beq t0, x0, buildOutput
isZeroDigitVal:
    # Is the digit value zero?
    bne t1, x0, buildOutput
isDigitEncountered:
    # t2 = digit encountered
    la t5, digitEncountered
    lb t2, 0(t5)

    # Has a non-zero digit been encountered?
    beq t2, x0, buildOutput

    # Clear display
    li t3, -1
    li t4, -1
    j doWrite7Seg
buildOutput:
    li t2, 1 # Set digit encountered

    # t3 = anode output = ~(1 << index)
    li t3, 1
    sll t3, t3, t0
    not t3, t3

    # t4 = segment output = sseg[digit value]
    add t5, s7, t1
    lbu t4, 0(t5)
doWrite7Seg:
    li t5, PORT_AN # Anodes
    li t6, PORT_SEG # Segments

    li t1, -1
    sw t1, 0(t5) # Clear anodes
    sw t3, 0(t5) # Write to anodes
    sw t4, 0(t6) # Write to segs

# Decrementing reached zero? We do this instead of a simple
# modular add because we need to clear digit encountered.
```

```
shouldLoopAround:
    beq t0, zero, doLooparound
doDecrement:
    addi t0, t0, -1
    j cleanup
doLooparound:
    li t0, 4 # Set digit to 4 (janky workaround?)
    li t2, 0 # Clear digit encountered
cleanup:
    # Store flags and data
    la t5, digitIndex
    sb t0, 0(t5)
    la t5, digitEncountered
    sb t2, 0(t5)
    ret

#-----------------------------------------------------------------
#- Converts a value less than 10000 into its 4-digit BCD
#- representation. Adapted from Lab 4.
#-
#- Parameters
#- a0 - value to convert
#- Returns
#- a1 - BCD representation
#- Tweaked registers - t0, t2, a1
#-----------------------------------------------------------------
bcd:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw a0, 4(sp)
    sw a2, 8(sp)

    mv a1, a0 # Dividend
    li a2, 1000 # Divisor
    call divide # a0 R a1 = a1 / a2

    mv t2, a0 # Quotient = thousands digit.
    slli t2, t2, 4 # Shift BCD

    li a2, 100 # Divisor. Modulus = new dividend.
    call divide # a0 R a1 = a1 / a2

    or t2, t2, a0 # Quotient = hundreds digit.
    slli t2, t2, 4 # Shift BCD.

    li a2, 10 # Divisor. Modulus = new dividend.
    call divide # a0 R a1 = a1 / a2

    or t2, t2, a0 # Quotient = ten’s digit.
    slli t2, t2, 4 # Shift BCD.
    or t2, t2, a1 # Modulus = one’s digit.

    mv a1, t2 # Return value
    lw ra, 0(sp)
    lw a0, 4(sp)
    lw a2, 8(sp)
    addi sp, sp, 12

    ret

#---------------------------------------------------------------
#- Implementation of binary division. Given a dividend n and
#- divisor p, runs in approximately log(n / p) time.
#-
#- Parameters
#- a1 - dividend
#- a2 - divisor
#- Returns
#- a0 - quotient
#- a1 - modulus/remainder
#- Tweaked registers - a0, a1, t0
#---------------------------------------------------------------
divide:
    blt a1, a2, iszero # If dividend < divisor return early

    # Store registers on stack
    addi sp, sp, -4
    sw a2, 0(sp)

    # Initialize registers
    li t0, 1 # divisor coefficient

# Double the divisor until it is greater than dividend
expand:
    slli t0, t0, 1
    slli a2, a2, 1
    bgt a1, a2, expand

    beq a1, a2, isPower2


beginSubtract:
    li a0, 0 # quotient
# Halve the divisor until we go under the original value
subtractloop:
    srai t0, t0, 1
    beq zero, t0, enddiv # End
    srai a2, a2, 1
    blt a1, a2, subtractloop # Don't subtract if a1 < a2

# Add to quotient if divisor falls below modulus/dividend
subtract:
    or a0, a0, t0
    sub a1, a1, a2
    bne a1, x0, subtractloop
enddiv:
    lw a2, 0(sp)
    addi sp, sp, 4
    ret
isPower2:
    mv a0, t0
    li a1, 0
    j enddiv
iszero:
    li a0, 0
    ret

end: j end
```

5 Timer-counter Calculations

We have the following assumptions:

1. The timer-counter is running at 50 MHz.
2. Every interrupt will increment the digit counter.
3. We want to draw at least 200 frames per second.
4. Due to a bug that is "fixed" by a workaround in the program, there are actually 5 virtual digits that the program iterates through. However, only digits 0-3 actually display stuff, and digit 4 does not.

Thus, the value for the timer-counter should be less than:

$$\frac{50 \times 10^6 \text{ clock}}{\text{s}} \cdot \frac{\text{s}}{200\,\text{displays}} \cdot \frac{\text{display}}{5\,\text{digits}} = 50000 \text{ clock} \tag{1}$$