Kyle Han, 113799110

ESE280 Laboratory 11

Section 3

Bench 9

Sensors, Basic Analog-to-Digital Conversion, and the 12bit ADC

1. What is the significance (particular usefulness) of the internal 2.5V reference voltage?

Because it is hard to store decimals (Especially analogous "When do they stop!" voltages, that refer to a very specific analog value, we can use a different method from directly storing them. We instead store them as a discrete fraction of a certain reference voltage.

Therefore, instead of saying "1.25V", we can say "½VREF", where Vref is 2.5V. This fraction is infinitely easier to store, over calculating the binary representation of a floating point value, and storing that decimal point.

Essentially, it turns a continuous analog voltage into a discrete voltage that our microcontroller can actually use via a scaling process.

2. When you single step your program in Task 2, is the analog-to-digital conversion process single stepped? Explain how you know this.

No, the ADC is not single stepped. We know this because each time we return to poll the ADC, it is immediately ready with a value (Since a practically infinite amount of time has elapsed since last checked, at least according to the microcontroller).

This also makes sense because the ADC is a different module than the CPU. Via single stepping, we only pause the CPU to look at the values, but all other modules are not stopped.

3. When your program polls to determine the end of a conversion, which flag bit must it poll and why?

It must poll bit 0 of the "ADC0_INTFLAGS" register. According to the datasheet, this register will be turned to 1 when a result is ready. Thus, to see if a conversion is done, just poll this bit.

4. When measuring a fixed voltage in Task2, do any of the least significant bits of the output change? If so, to what do you attribute these changes.

Yes, they change quite a bit. I attribute these changes to noise in the system, since nothing is perfect. Even the multimeter shows us it's least significant value changing slightly. Thus, our microcontroller's value will be changing, and so will the value being read.

5. What is the range of temperatures that can be measured using the MCP9700A and internal 2.5V reference? What is the smallest temperature change that can be detected. Show your calculations.

The temp range that can be measured is -38C-125C. The datasheet tells us that these voltages are between 0V and 2V, both voltages that our ADC/Microcontroller are built to handle. Thus, we can access this whole range.

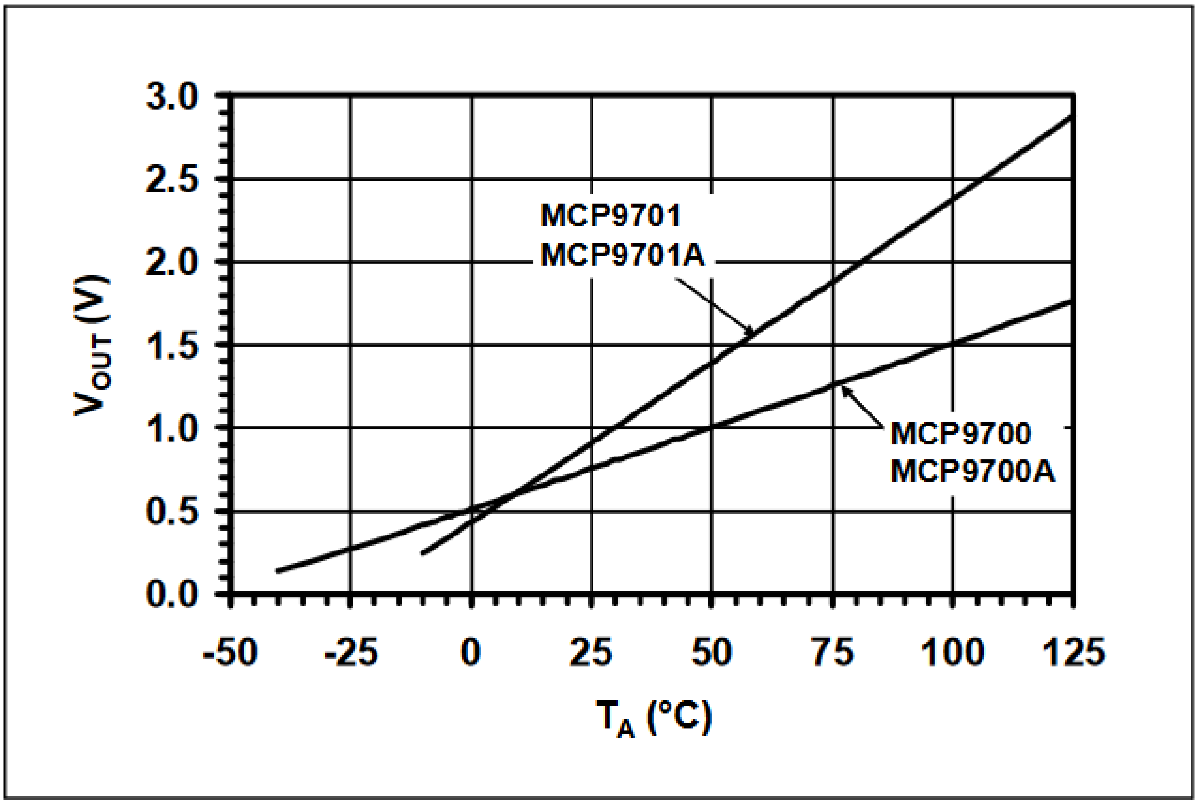
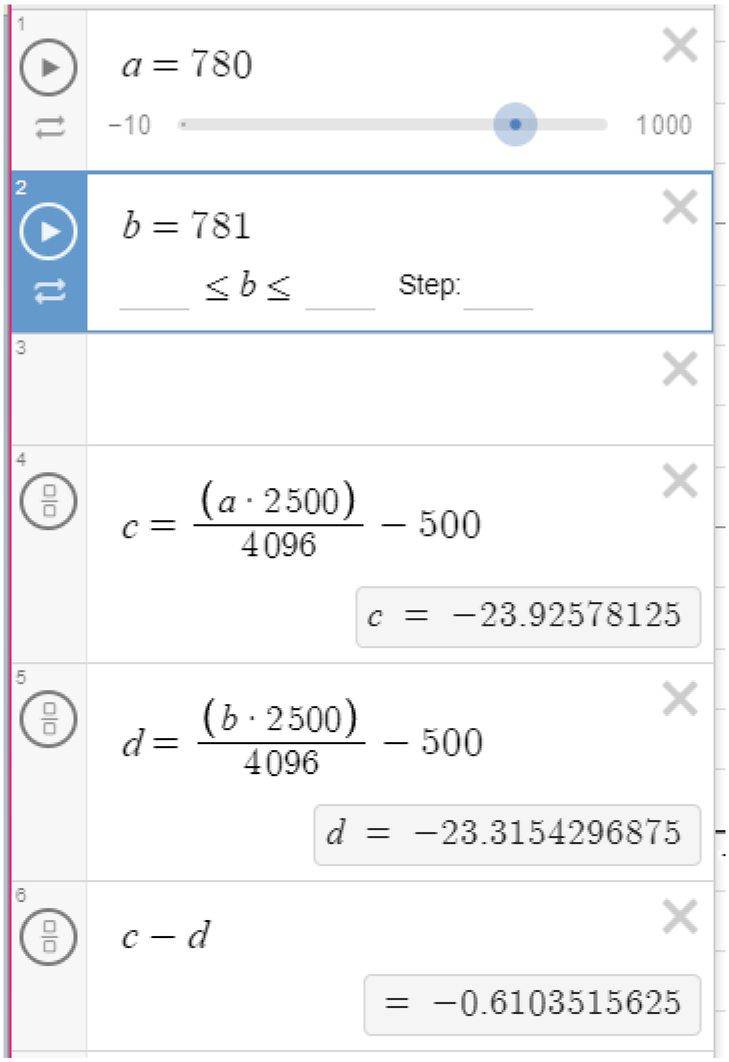


FIGURE 2-16: *Output Voltage vs. Ambient Temperature.*

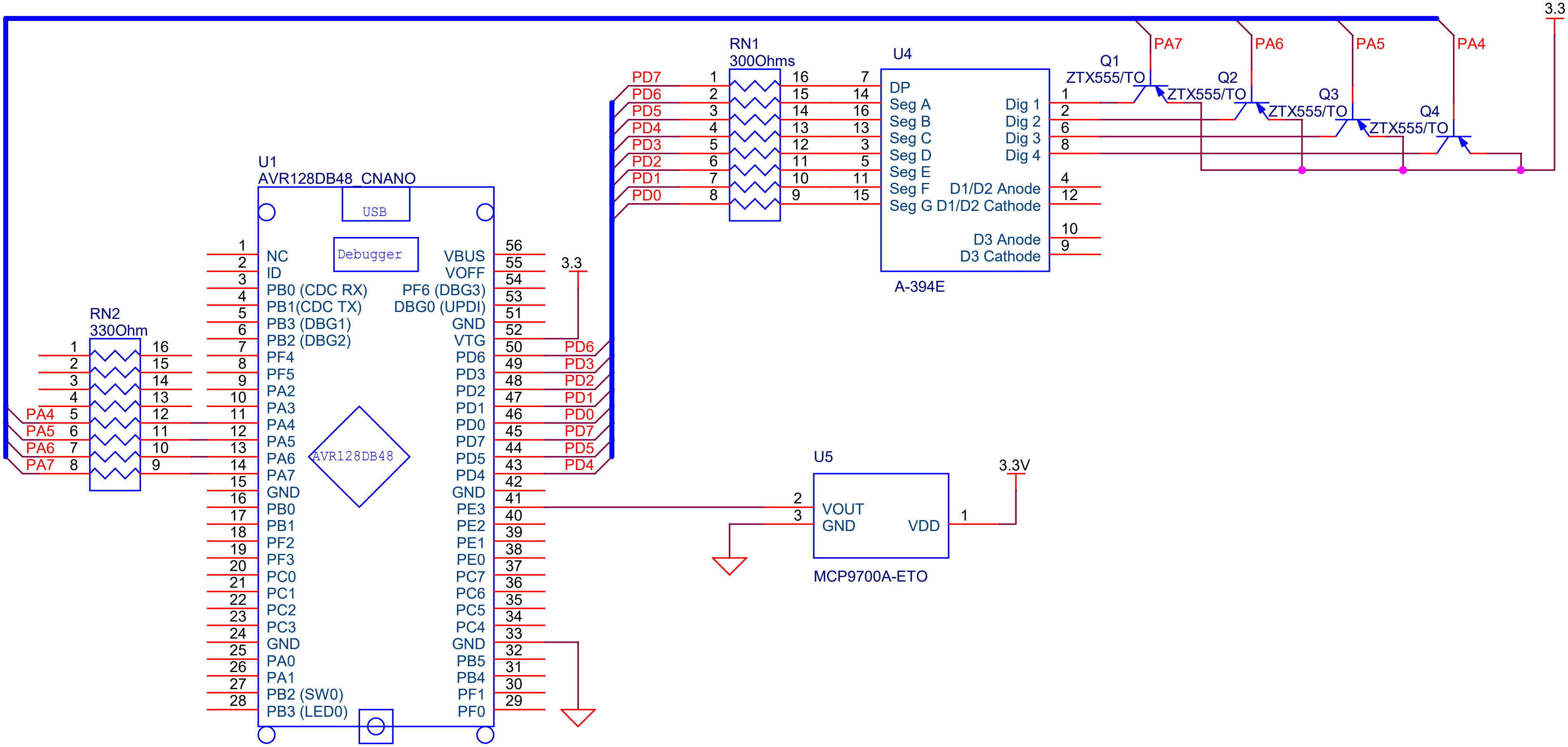
The smallest temp change we can detect is from two consecutive hexs. Thus, plugging this into the equation that we have, we can compute the smallest temperature change to be .61C.



6. Write the basic equation for the computation you must perform to scale the ADC binary output that represents the measured voltage to a binary representation of the measured temperature. Are there any limitations in the order in which the operations in the equation must be performed. If so, explain the limitations and why they exist.

((RES*2500)/4096)-500

Yes, the limitation is that our divide function will cause us to lose data on the decimal side. Thus, even though in the real world, we can do multiplication and division in any order, in this case, we should do multiplication first, on order to not lose any data.



Stony Brook University: ESE 280 - Embedded Microcontroller Systems Design I

Title
Kyle Han: Lab 11 Task 3

Size	Document Number	Rev
A	ESE280L11T3KH	1

Date: Tuesday, November 30, 2021 Sheet 1 of 1

```
;
; temp_meas.asm
;
; Created: 11/30/2021 5:02:21 AM
; Author : Kyle Han
;


.equ PERIOD = 97 //((4MHz * (1/160))/256)-1


.dseg
bcd_entries: .byte 4
led_display: .byte 4
digit_num: .byte 1


.cseg
reset:
    rjmp start
.org TCA0_OVF_vect
    rjmp multiplex_display
.org ADC0_RESRDY_vect
    rjmp ADC0_response

start:

//This section deals with port configurations
    //Set all of PORTD to outputs (For the 7seg)
    ldi r16, 0xFF
    sts PORTD_DIR, r16

    //Set PA7-PA4 as outputs (For the multiplexer)
    ldi r16, 0xF0
    sts PORTA_DIR, r16

    //Configures all of PORTE as inputs
    ldi r16, 0x00
    sts PORTE_DIR, r16

//This section deals with timer TCA0
//Sets up TCA0 timer for the interrupt. 40Hz
    ldi r16, TCA_SINGLE_WGMODE_NORMAL_gc    ;WGMODE normal
    sts TCA0_SINGLE_CTRLB, r16

    ldi r16, TCA_SINGLE_OVF_bm      ;enable overflow interrupt
    sts TCA0_SINGLE_INTCTRL, r16

    ;load period low byte then high byte
    ldi r16, LOW(PERIOD)        ;set the period
    sts TCA0_SINGLE_PER, r16
    ldi r16, HIGH(PERIOD)
```

```
    sts TCA0_SINGLE_PER + 1, r16


//This section deals with Analog-to-Digital-Converter ADC0
    //Set the controls as a 12 bit resolution, and enable the ADC
    ldi r16, ADC_RESSEL_12BIT_gc | ADC_ENABLE_bm
    sts ADC0_CTRLA, r16

    //Config the voltage ref to 2.5V
    ldi r16, VREF_REFSEL_2V500_gc
    sts VREF_ADC0REF, r16

    //Sets the prescaler to 64 bits
    ldi r16, ADC_PRESC_DIV64_gc
    sts ADC0_CTRLC, r16

    //Configs PE3 as the positive analog input to the ADC
    //AIN11 = PE3
    ldi r16, ADC_MUXPOS_AIN11_gc
    sts ADC0_MUXPOS, r16

    //Configures the interrupt for the ADC
    ldi r16, ADC_RESRDY_bm
    sts ADC0_INTCTRL, r16

    //POST the display. Once posted, then start the timer and continue as normal.
    rcall post_display

//Starting line
    //set clock and start timer
    ldi r16, TCA_SINGLE_CLKSEL_DIV256_gc | TCA_SINGLE_ENABLE_bm
    sts TCA0_SINGLE_CTRLA, r16

    //Start ADC0's conversion
    ldi r16, ADC_STCONV_bm
    sts ADC0_COMMAND, r16
    sei

loop:
    nop
    rjmp loop


;***************************************************************************
;*
;* "ADC0_response" - Read the result of ADC0
;*
;* Description: Read in the value, and update bcd_entries/led_display accordingly
;*
;* Author: Kyle Han
;* Version: 0.1
;* Last updated: 11/29/2021
```

```
;* Target: AVR128DB48
;* Number of words:
;* Number of cycles:
;* Low registers modified: r16-r18
;* High registers modified:
;*
;* Parameters: None
;*
;* Returns: None
;*
;* Notes: Requires digit_num, bcd_entries, and led_display to be declared in .dseg
;*
;***************************************************************************
ADC0_response:
    //Saving things to the stack
    push r16
    push r17
    push r18
    push r19
    push r20
    push r21
    push r22

    //Past this point, we know that a conversion is ready for us to read in.
    lds r16, ADC0_RESL
    lds r17, ADC0_RESH

    //We will now compute the formula to transform the result into our degrees celsius
    //Eqn: T(0.1C) = ((RES x 2500)/4096)-500

    //Load s the multipleier (2500)(1001 1100 0100) into r18 and r19
    ldi r18, 0b11000100
    ldi r19, 0b00001001

    rcall mpy16u

    //Result now stored in r18-r21. Divide by 4096 (2^12) (Anotherwards, ignore the 3 least signifigant hexs)
    //Ignore 18, and the latter half of 19
    //Result of this is stored (High) r20, (low) r19
        ldi r22, 4 //Do this shift 4 times
Div4096:
    clc
    ror r20
    ror r19
    dec r22
    brne Div4096
    //This shift has been done 4 times. We've now divided by 4096

    //We need to check if the number is smaller than 500 first.
```

```
    //Larger: Res-500
    //Smaller: 500-Res, set T bit
    //if it is larger, we continue as normal
    //Output is stored in r17 (HIGH), r16 (LOW)
    ldi r16, LOW(500)
    ldi r17, HIGH(500)
    cp r19, r16
    cpc r20, r17
    brlo negative
    sub r19, r16
    sbc r20, r17
    mov r16, r19
    mov r17, r20
    rjmp continueUpdate

negative:
    set
    sub r16, r19
    sbc r17, r20


continueUpdate:
    rcall bin2BCD16




//Will copy everything needed into bcd_entries, and create the corresponding values in led_display
    //Entry 0
    lds r18, tBCD0
    andi r18, 0x0F
    sts bcd_entries+3, r18
    rcall hex_to_7seg
    sts led_display+3, r18

    //Entry 1
    lds r18, tBCD0
    andi r18, 0xF0
    //We need a shift right to get this back to bcd (Since that only requires 4 bits)
    lsr r18
    lsr r18
    lsr r18
    lsr r18
    sts bcd_entries+2, r18
    rcall hex_to_7seg
    sts led_display+2, r18

    //Entry 2
    lds r18, tBCD1
    andi r18, 0x0F
```

```
    sts bcd_entries+1, r18
    rcall hex_to_7seg
    sts led_display+1, r18

    //Entry 3 is special. If the T bit is set, then we know we must display a - (Negative).
    //Otherwise, display whatever should be displayed.
    //Entry 3
    brtc positiveEntry3

negativeEntry3:
    ldi r18, 0xFE //For the negative sign
    sts led_display+0, r18
    clt
    rjmp retADC


positiveEntry3:
    lds r18, tBCD1
    andi r18, 0xF0
    lsr r18
    lsr r18
    lsr r18
    lsr r18
    sts bcd_entries+0, r18
    rcall hex_to_7seg
    sts led_display+0, r18

//Returns from the ADC subroutine. Saves things and make sure the registers are left unmodified.
retADC:
    pop r22
    pop r21
    pop r20
    pop r19
    pop r18
    pop r17
    pop r16

    //Restart ADC0's conversion
    ldi r16, ADC_STCONV_bm
    sts ADC0_COMMAND, r16
    reti

;***************************************************************************
;*
;* "hex_to_7seg" - Hexadecimal to Seven Segment Conversion
;*
;* Description: Converts a right justified hexadecimal digit to the seven
;* segment pattern required to display it. Pattern is right justified a
;* through g. Pattern uses 0s to turn segments on ON.
;*
```

```
;* Author:          Ken Short
;* Version:         0.1
;* Last updated:        101221
;* Target:          AVR128DB48
;* Number of words:
;* Number of cycles:
;* Low registers modified:
;* High registers modified:
;*
;* Parameters: r18: hex digit to be converted
;* Returns: r18: seven segment pattern. 0 turns segment ON
;*
;* Notes:
;*
;***************************************************************************
hex_to_7seg:
    push ZH
    push ZL
    ldi ZH, HIGH(hextable * 2)  ;set Z to point to start of table
    ldi ZL, LOW(hextable * 2)
    ldi r16, $00                ;add offset to Z pointer
    andi r18, 0x0F              ;mask for low nibble
    add ZL, r18
    adc ZH, r16
    lpm r18, Z                  ;load byte from table pointed to by Z
    pop ZL
    pop ZH
    ret

    ;Table of segment values to display digits 0 - F
    ;!!! seven values must be added
hextable: .db $01, $4F, $12, $06, $4C, $24, $20, $0F, $00, $04, $08, $60, $31, $42, $30, $38


;***************************************************************************
;*
;* Multiplex_display
;*
;* Description: reads everything in the memory "array" and converts it to a 7seg interpetatoin.
;*
;*
;*
;* Author:          Kyle Han
;* Version:         0.1
;* Last updated:        11032021
;* Target:          AVR128DB48
;* Number of words:
;* Number of cycles:
;* Low registers modified:
;* High registers modified:
```

```
;*
;* Parameters: A pointer called array, and pointer X available
;* Returns: Everything inside pointer X
;*
;* Notes:
;*
;***************************************************************************


multiplex_display:

//Turns off the whole display by outputting 1s to PORTA. Then, check with digit_num to see which display should be lit
turn_off:

    //First, we must push all the registers we are using to the stack. This is so that the original values are restored later on
    push r16
    push r17
    push r18
    in r16, CPU_SREG
    push r16
    push XL
    push XH

    //Handles the interrupt and resets the timer
    ldi r16, TCA_SINGLE_OVF_bm  ;clear OVF flag
    sts TCA0_SINGLE_INTFLAGS, r16

    ldi r16, 0xFF
    sts PORTA_OUT, r16
    lds r17, digit_num
    inc r17
    //If we are at position 4, return to position 0
    cpi r17, 0x04
    brsh overflow

output:
    sts digit_num, r17
    ldi XH, HIGH(led_display)
    ldi XL, LOW(led_display)
    add XL, r17
    ld r18, X
    //rcall hex_to_7seg
    sts PORTD_OUT, r18

//Will check digit_num to decide which display on the 7seg is ebing outputted
checking_dig:
    cpi r17, 0x00
    breq dig0
    cpi r17, 0x01
    breq dig1
```

```
    cpi r17, 0x02
    breq dig2
    cpi r17, 0x03
    breq dig3

    //r18 stores the value of the digit to be displayed
dig0:
    ldi r18, 0x70   //PA7
    sts PORTA_OUT, r18
    rjmp restore

dig1:
    ldi r18, 0xB0   //PA6
    sts PORTA_OUT, r18
    rjmp restore

dig2:
    ldi r18, 0xD0   //PA5
    sts PORTA_OUT, r18
    rjmp restore

dig3:
    ldi r18, 0xE0 //PA4
    sts PORTA_OUT, r18
    rjmp restore

//Reads off the stack the values that we preserved inside turn_off
restore:
    pop XH
    pop XL
    pop r16
    out CPU_SREG, r16
    pop r18
    pop r17
    pop r16

    reti

//Sets digit_num back to 0, since we only have 4 digits
overflow:
    ldi r17, 0x00
    sts digit_num, r17
    ldi XH, HIGH(led_display)
    ldi XL, LOW(led_display)
    rjmp output


;***************************************************************************
;*
;* "post_display" - Power On Self Test
;*
;* Description: Will POST the 7segment display hooked up to port D, multiplexed
```

```
  by PA7-PA4.
;*Individually turns on each segment for a brief moment, totalling 1 second
;* Author: Kyle Han
;* Version: 0.1
;* Last updated: 11/29/2021
;* Target: AVR128DB48
;* Number of words:
;* Number of cycles:
;* Low registers modified: r16-r20
;* High registers modified:
;*
;* Parameters: None
;*
;* Returns: None
;*
;* Notes: Requires digit_num to be declared in .dseg
;*
;***************************************************************************
post_display:
    //A value of 0 turns on our LED displays
    ldi r16, 0x00
    out VPORTD_OUT, r16
    sts digit_num, r16 ;Store 0x00 to the current digit to be displayed
repeatPost:
    in r17, VPORTA_OUT
    ori r17, 0xF0
    out VPORTA_OUT, r17
    ldi r19, 0b00010000 ;This 1 will turn on a specific digit.
leftShiftPOST:
    cpi r16, 0
    breq postDispOn
    lsl r19
    dec r16
    rjmp leftShiftPOST
postDispOn:
    in r17, VPORTA_OUT
    eor r17, r19
    out VPORTA_OUT, r17
    ldi r20, 10
    rcall oneSecDelay
    //Compares digit_num to 0x04. If greater than or equal to, we've finished 
      posting
    lds r16, digit_num
    inc r16
    sts digit_num, r16
    cpi r16, 0x04
    brlo repeatPost

    in r17, VPORTA_OUT
    ori r17, 0xF0
    out VPORTA_OUT, r17
    ret
```

```
;***************************************************************************
;*
;* "oneSecDelay" - Delay the microcontroller by 1 second by occupying CPU time
;*
;* Description: Loops through and occupies CPU time to delay by 1 second
;*
;* Author: Kyle Han
;* Version: 1.0
;* Last updated: 11/29/2021
;* Target: AVR128DB48
;* Number of words:
;* Number of cycles:
;* Low registers modified:r17
;* High registers modified:
;*
;* Parameters: r20 * .1ms
;*
;* Returns: Nothing
;*
;* Notes:
;*
;***************************************************************************
oneSecDelay:
outer_loop:
    ldi r17, 133
inner_loop:
    dec r17
    brne inner_loop
    dec r20
    brne outer_loop
    ret

;***************************************************************************
;*
;* "mpy16u" - 16x16 Bit Unsigned Multiplication
;*
;* This subroutine multiplies the two 16-bit register variables
;* mp16uH:mp16uL and mc16uH:mc16uL.
;* The result is placed in m16u3:m16u2:m16u1:m16u0.
;*
;* Number of words  :14 + return
;* Number of cycles :153 + return
;* Low registers used   :None
;* High registers used  :7 (mp16uL,mp16uH,mc16uL/m16u0,mc16uH/m16u1,m16u2,
;*                          m16u3,mcnt16u)
;*
;***************************************************************************

;***** Subroutine Register Variables

.def    mc16uL  =r16        ;multiplicand low byte
```

```
.def    mc16uH  =r17        ;multiplicand high byte
.def    mp16uL  =r18        ;multiplier low byte
.def    mp16uH  =r19        ;multiplier high byte
.def    m16u0   =r18        ;result byte 0 (LSB)
.def    m16u1   =r19        ;result byte 1
.def    m16u2   =r20        ;result byte 2
.def    m16u3   =r21        ;result byte 3 (MSB)
.def    mcnt16u =r22        ;loop counter

;***** Code

mpy16u: clr m16u3       ;clear 2 highest bytes of result
    clr m16u2
    ldi mcnt16u,16  ;init loop counter
    lsr mp16uH
    ror mp16uL

m16u_1: brcc    noad8       ;if bit 0 of multiplier set
    add m16u2,mc16uL    ;add multiplicand Low to byte 2 of res
    adc m16u3,mc16uH    ;add multiplicand high to byte 3 of res
noad8:  ror m16u3       ;shift right result byte 3
    ror m16u2       ;rotate right result byte 2
    ror m16u1       ;rotate result byte 1 and multiplier High
    ror m16u0       ;rotate result byte 0 and multiplier Low
    dec mcnt16u     ;decrement loop counter
    brne    m16u_1      ;if not done, loop more
    ret


;***************************************************************************
;*
;* "bin2BCD16" - 16-bit Binary to BCD conversion
;*
;* This subroutine converts a 16-bit number (fbinH:fbinL) to a 5-digit
;* packed BCD number represented by 3 bytes (tBCD2:tBCD1:tBCD0).
;* MSD of the 5-digit number is placed in the lowermost nibble of tBCD2.
;*
;* Number of words  :25
;* Number of cycles :751/768 (Min/Max)
;* Low registers used   :3 (tBCD0,tBCD1,tBCD2)
;* High registers used  :4(fbinL,fbinH,cnt16a,tmp16a)
;* Pointers used    :Z
;*
;***************************************************************************

;***** Subroutine Register Variables

.dseg
tBCD0: .byte 1  // BCD digits 1:0
tBCD1: .byte 1  // BCD digits 3:2
tBCD2: .byte 1  // BCD digits 4

```

```
.cseg
.def    tBCD0_reg = r13     ;BCD value digits 1 and 0
.def    tBCD1_reg = r14     ;BCD value digits 3 and 2
.def    tBCD2_reg = r15     ;BCD value digit 4

.def    fbinL = r16     ;binary value Low byte
.def    fbinH = r17     ;binary value High byte

.def    cnt16a  =r18        ;loop counter
.def    tmp16a  =r19        ;temporary value

;***** Code

bin2BCD16:
    push fbinL
    push fbinH
    push cnt16a
    push tmp16a


    ldi cnt16a, 16  ;Init loop counter
    ldi r20, 0x00
    sts tBCD0, r20 ;clear result (3 bytes)
    sts tBCD1, r20
    sts tBCD2, r20
bBCDx_1:
    // load values from memory
    lds tBCD0_reg, tBCD0
    lds tBCD1_reg, tBCD1
    lds tBCD2_reg, tBCD2

    lsl fbinL       ;shift input value
    rol fbinH       ;through all bytes
    rol tBCD0_reg       ;
    rol tBCD1_reg
    rol tBCD2_reg

    sts tBCD0, tBCD0_reg
    sts tBCD1, tBCD1_reg
    sts tBCD2, tBCD2_reg

    dec cnt16a      ;decrement loop counter
    brne bBCDx_2        ;if counter not zero

    pop tmp16a
    pop cnt16a
    pop fbinH
    pop fbinL
ret         ; return
    bBCDx_2:
    // Z Points tBCD2 + 1, MSB of BCD result + 1
    ldi ZL, LOW(tBCD2 + 1)
```

```
    ldi ZH, HIGH(tBCD2 + 1)
    bBCDx_3:
        ld tmp16a, -Z       ;get (Z) with pre-decrement
        subi tmp16a, -$03   ;add 0x03

        sbrc tmp16a, 3      ;if bit 3 not clear
        st Z, tmp16a        ;store back

        ld tmp16a, Z    ;get (Z)
        subi tmp16a, -$30   ;add 0x30

        sbrc tmp16a, 7  ;if bit 7 not clear
        st Z, tmp16a    ;   store back

        cpi ZL, LOW(tBCD0)  ;done all three?
    brne bBCDx_3
        cpi ZH, HIGH(tBCD0) ;done all three?
    brne bBCDx_3
rjmp bBCDx_1
```