PHYS 319 Lab Notes

Rio Weil, Student# 47189394

This document was typeset on February 1, 2021

Contents

1 Lab 1

1.1 Lab Objective

The objective of today's lab is to manually set a group of 4 7-segmented number displays to read the last 4 digits of my student number, i.e. 9394. Next lab we will work on programming the MSP430 to set the display for us.

1.2 Breadboard Refamiliarization

The breadboard holes are connected as seen in the diagram below, in horizontal lines at the top/bottom of the board and in vertical groups of 5. Since the red/blue lines on my board are not broken, the power labels are connected all the way through.
The diagram was taken from `https://components101.com/misc/breadboard-connections-uses-guide`.

1.3 Understanding how the chips work

Attached is a schematic of the lab breadboard display. The first part of this lab will be trying to understand how the DM9368 (7 segment decoder/driver/latch) and the 74HC139 (2 to 4 line decoder) work. The combination of these two chips allows us to control the 4 segments with 7 wires (D0-D3 data pins, Address A0/A1, and Strobe) rather than 16.

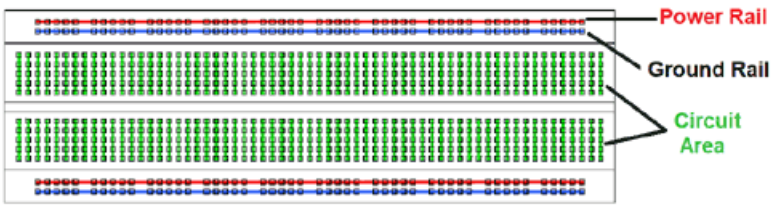

Figure 1: Connectnedness of breadboard holes.

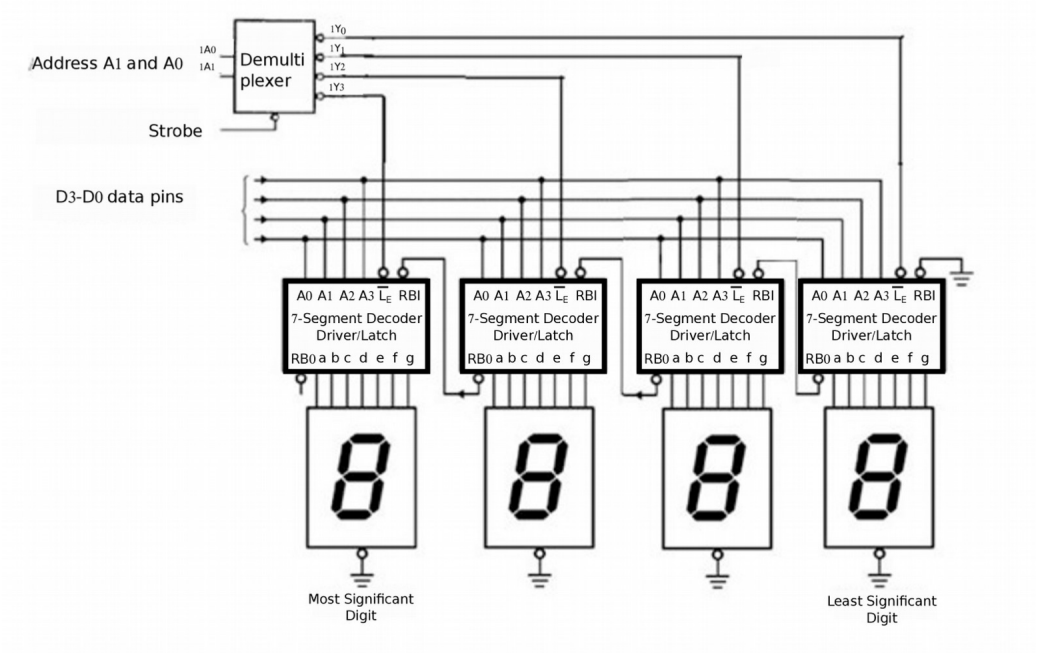

Figure 2: Schematic of Lab Breadboard Display

To start, it may be useful to define what all this jargon means. A *decoder* changes code into a circuit of signals. The DM9368 does this by taking in inputs (essentially binary "code" as we will see shortly) and turning this into a signal (the display we see on screen). What goes on inside the DM9368 might be a black box but we leave the inner workings of the microchip to the computer engineers. A *driver* controls another circuit component, which again the DM9368 can be seen to be doing as it controls the display. A *latch* is a circuit component with two states that can store stable information (in this case, the DM9368 can store the data of what digit we wish to display, depending on the high/low status of the STROBE pin). A *demultiplexer* is something that takes in a common input line into multiple output lines (which we will see the 74HC139 does in a moment, as it takes in two input lines to determine the status of 4 output lines) (and a *multiplexer* does basically the opposite).

We first being with a discussion of how the DM9368 works. There are four of these, one for each display, as shown above. The D0-D3 (data) pins (which we can control to have an input of either high or low) feed into the A0-A3 (address) pins on each of the DM9368 chips. The attached truth table from the datasheet of the DM9368 shows us what inputs correspond to which segments of the display lighting up:
Essentially, we use 4 binary inputs (4 bits) to set the display. Looking at this datasheet, it becomes clear that if we did not have a 74HC139 decoder chip, we would require 16 inputs (i.e. one for each of the A0-A3

Truth Table

BINARY STATE	INPUTS $\overline{LE}$	$\overline{RBI}$	A3	A2	A1	A0	OUTPUTS a	b	c	d	e	f	g	$\overline{RBO}$	DISPLAY
—	H	*	X	X	X	X	◄ STABLE ►							H	STABLE
0	L	L	L	L	L	L	L	L	L	L	L	L	L	L	BLANK
0	L	H	L	L	L	L	H	H	H	H	H	H	L	H	0
1	L	X	L	L	L	H	L	H	H	L	L	L	L	H	1
2	L	X	L	L	H	L	H	H	L	H	H	L	H	H	2
3	L	X	L	L	H	H	H	H	H	H	L	L	H	H	3
4	L	X	L	H	L	L	L	H	H	L	L	H	H	H	4
5	L	X	L	H	L	H	H	L	H	H	L	H	H	H	5
6	L	X	L	H	H	L	H	L	H	H	H	H	H	H	6
7	L	X	L	H	H	H	H	H	H	L	L	L	L	H	7
8	L	X	H	L	L	L	H	H	H	H	H	H	H	H	8
9	L	X	H	L	L	H	H	H	H	L	L	H	H	H	9
10	L	X	H	L	H	L	H	H	H	L	H	H	H	H	8
11	L	X	H	L	H	H	L	L	H	H	H	H	H	H	8
12	L	X	H	H	L	L	H	L	L	H	H	H	L	H	C
13	L	X	H	H	L	H	L	H	H	H	H	L	H	H	d
14	L	X	H	H	H	L	H	L	L	H	H	H	H	H	E
15	L	X	H	H	H	H	H	L	L	L	H	H	H	H	F
X	X	X	X	X	X	X	L	L	L	L	L	L	L	L**	BLANK

*The $\overline{RBI}$ will blank the display only if a binary zero is stored in the latches.
*The $\overline{RBO}$ used as an input overrides all other input conditions.
H = HIGH Voltage Level
L = LOW Voltage Level
X = Immaterial

Figure 3: Truth table of what inputs into A0-A3 for the DM9368 chip corresponds to the values that we see on the display.

address pins for each of the DM9368+displays). Note that in this truth table, L represents a low voltage (i.e. ground) and H represents a high voltage (i.e. 5V). It is now clear what we are manipulating when we change the D0-D3 data pins from our switchboard, but now we should look into what manipulating the A0 and A1 datapins do (we can see on the schematic that manipulating A0 and A1 will control the demultiplexer, which in turn controls the LE pin on each of the DM9368s).

Attached is the functional diagram for the 74HC139 decoder, as well as a table of what it does according to each input: Looking at this table, it becomes clear that depending on our inputs into the A0 and A1 pins,

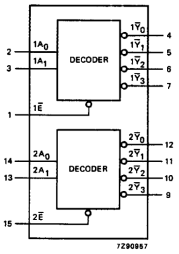

Fig.4 Functional diagram.

FUNCTION TABLE

INPUTS $n\overline{E}$	nA_0	nA_1	OUTPUTS $n\overline{Y}_0$	$n\overline{Y}_1$	$n\overline{Y}_2$	$n\overline{Y}_3$
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	H	L	H	L	H	H
L	L	H	H	H	L	H
L	H	H	H	H	H	L

Figure 4: The schematic of the 74HC139 decoder and function table of what inputs correspond to what outputs

the decoder chip returns an output of L to one of the four of Y0-Y3, which each correspond to one DM9368 chip. Looking back at the truth table of the DM9368 chip, we can see that if the LE input is set to high, then the display will be STABLE (which corresponds to an "F" on the display) and if the LE input is set to low, then we are able to control the address pins to set the display to be what we want.
Putting it together, this gives us a method on how to control each display one at a time. When we control the A0/A1 pins through the switchboard, we are specifying which one of the four DM9368 chips/displays we are setting. We can then use the D0-D3 pins to set that specified chip/display to the desired number (using the truth table as attached above). We then switch the STROBE pin from high to low, which updates the display (this is given to us in the lab manual). Repeating this 4 times for each of the displays will allow us to set the displays to our student number! From this, it is clear why this setup allows us to reduce the number of input pins we use from 16 to 7; instead of having to set 4 address pins (4 bits) for 4 DM9368 displays at once, we instead control which chip to control using the A0/A1 pins that lead into the 74HC139 decoder, then we can use the D0-D3 pins to set each DM9368 individually. These 6 pins + the strobe pin for updating the chip/displays are definitely a gain in terms of efficiency in the number of inputs required.

1.4 Finer details on input/output voltages and specifications

In this section, we discuss on finger specifications of the devices we are using, with respect to voltages (e.g. answering the question of what constitutes as high voltage?) as well as AC characteristics of the chips and display that we are using.
We first answer the question of what is meant by the quantities of V_{OHmin}, V_{IHmin}, V_{OLmax}, and V_{ILmax}[1]. For TTL (Transistor to Transistor logic) devices such as the DM9368, ideally we would have that a "high" voltage/logic state would be perfectly $5V$ and that a "low" voltage/logic state would be perfectly $0V$. However this is very difficult to implement in practice and hence actual devices accept (as input) and transmit (as output) a range of what is considered "high" versus "low" voltage. For TTLs, this range is $0V - 0.8V$ for "low" and $2V - 5V$ for the "high" in terms of reading in the input. V_{ILmax} refers to the maximum voltage at which input can be read in as "low", which in this case is $0.8V$, and V_{IHmax} refers to the minimum voltage at which input can be read in as "high", which in this case is $2V$. For TTL outputs, the range is $0 - 0.5V$ for a "low" voltage and the range is $2.7 - 5V$ for "high" voltage. V_{OLmax} refers to the maximum voltage at output which can be considered as "low" voltage, which in this case is $0.5V$, and V_{OHmin} is the minimum voltage at output which can be considered as "high", which in this case is $2.7V$.
The voltage tolerance of CMOS (Complementary Metal Oxide Semiconductor) devices such as the 74HC139 decoder are very different. The acceptable input ranges are $0V - 1.5V$ for a "low" voltage and $3.5V - 5V$ for a "high" voltage. For outputs, the ranges are very narrow with $0V - 0.05V$ as the range that could be considered a "low" output and $4.95V - 5V$ for the range that could be considered as a "high" voltage state. These ranges correspond to $V_{\text{ILmax}} = 1.5V$, $V_{\text{IHmin}} = 3.5V$, $V_{\text{OLmax}} = 0.05V$, and $V_{\text{OHmin}} = 4.95V$. These findings are summarized in the table below. In addition to the general parameters for TTL and CMOS devices (as obtained from the website in the footnote), we also include the information for the the DM9368 chip used in this experiment (the data for the 74HC139 could not be found in the datasheet).

	TTL (general)	CMOS (general)	DM9368
V_{ILmax}	0.8V	1.5V	0.8V
V_{IHmin}	2V	3.5V	2V
V_{OLmax}	0.5V	0.05V	0.4V
V_{OHmin}	2.7V	4.95V	2.4V

For the spec Next, we discuss the AC characteristics of the latch (i.e. the flip-flop DM9368) and how fast it can switch/how long it takes to respond to input changes. This can be found in the "Switching characterstics" part of the datasheet. We find that the propogation delay from the address pins A_n to the output/display pins $a - g$ (that each control a part of the display) are between $50 - 75$ nanoseconds and the propogation delay between the LE pin (which is the pin whose state determines whether we can modify the output of

[1]Information from `https://www.allaboutcircuits.com/textbook/digital/chpt-3/logic-signal-voltage-levels/`.

the chip or not) is between $70 - 90$ nanoseconds. Hence a good estimate for the response time for how fast the chip can respond to input changes is around ~ 70ns.

1.5 Set-up of the circuit

I began by plugging in the IC to the power supply by connecting the USB to my computer. I then hooked up the GND and the 5V wires to the pins on the display. I then placed down the CTS 206-8 to the board (straddling across the center) and connected it to the 5V power rail using resistors (the resistors are there so we can give the pins a high/logical 1 input but still be safe). The diagram of the setup is shown below: After

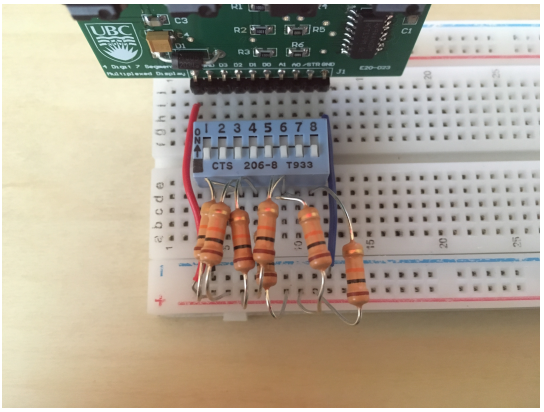

Figure 5: (Wrong) Closeup photo of the pin wiring

this process, the display showed "FFFF". Luke came and pointed out that we need to have the switches setup in such a way that they will be grounded when the input is not on. Therefore, we revised our setup so this would be the case. With Luke's suggestion, The D0-D3 pins, the A0, A1 pins, and the strobe pin are all setup in a way such that the pins get the 5V (logical 1) input if the switchbox is open, and the pins are grounded (logical 0) if the switches are closed and the circuit leads into ground. This is demonstrated conceptually in the figure below:

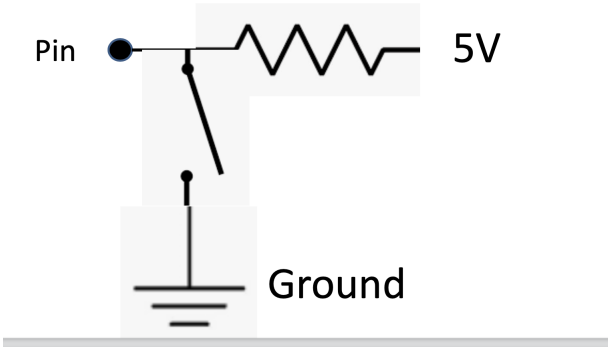

Figure 6: Demonstration of the switch setup. The pins are grounded when the switches are off, and have a HIGH (5V/1) input through a resistor when the switches are turned on.

After this revision, the circuit and wiring look as follows:

Essentially, the wiring consists of wiring from the 5V power rail to the 5V pin, ground wires from the grounding power rail to the ground pins, resistors going from the 5V power rail to the display and also to

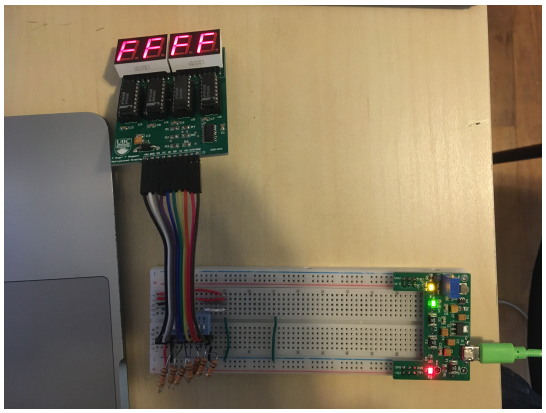

Figure 7: (Correct) picture of whole circuit

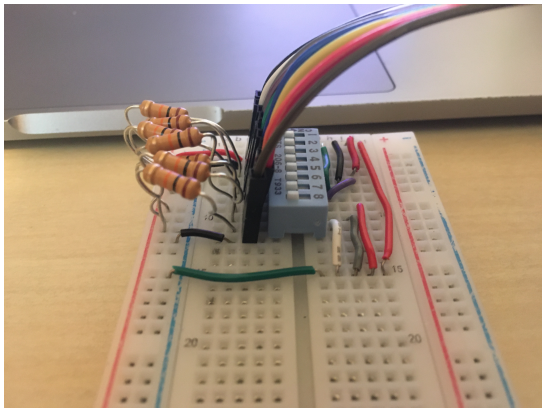

Figure 8: Closeup shot of the wiring

the switches in parallel. Then, each of the switches are hooked up to ground (from the ground) on the other side, allowing for a switch from high (5V) to low (GND). Note that after this I replaced the rainbow wires and just plugged in the display directly due to some connectivity issues with the rainbow wires (this can be seen in the final results as will be demonstrated shortly). At this point, the display still shows "FFFF", corresponding to the default state.

1.6 Setting the display manually

The 1 switch on my box is always grounded and has no effect. Switches 2-5 control the data pins D3-D0, switches 6-7 control address pins A1-A0, and switch 8 controls the STROBE. The last 4 digits of my student number are 9394, so let us walk through the process of setting the display to read these numbers. Initially, the display reads "FFFF" with all of the pins at L (grounded) and the strobe set to H (5V). In this explanation, note that "high/H/1" will correspond to having a 5v input (the switch is open) and "low/L/0" will correspond to having the pin grounded (the switch is closed).

1. To set the first digit of the display (which will be 9), we set A0 and A1 to high (as this tells the 74HC139 decoder to set Y3 output to L and the rest to H, i.e. set the LE pin for the first DM9368 chip to L (modifiable) and the rest to H (fixed)). We then set D3-H, D2-L, D1-L, D0-H (i.e. 9 in binary on the data pins). We then set strobe from high to low to update the display with 9 (and back to high for the next step).

2. Set A0-L and A1-H. This sets Y2 L and the rest H, and so sets the LE pin on the second DM9368 chip to L (and the rest to H so they are fixed). We then set D3-L D2-L D1-H D0-H (3 in binary) and then set strobe from high to low to update the display.

3. Set A0-H and A1-L. This sets Y1 L and the rest H, and so sets the LE pin on the third DM9368 chip to L (and the rest to H so they are fixed). We then set D3-H, D2-L, D1-L, D0-H (9 in binary) and then set strobe from high to low to update the display.

4. Set A0-L and A1-L. This sets Y0 L and the rest H, and so sets the LE pin on the fourth DM9368 chip to L (and the rest to H so they are fixed). We then set D3-L, D2-H, D1-L, D0-L (4 in binary) and then set strobe from high to low.

And this finishes the lab :D For fun, we can also do the last 4 digits of my student number in hexadeximal.

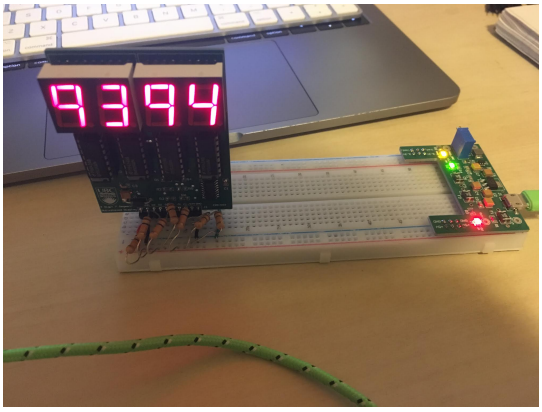

Figure 9: I feel like a proud dad. My display did it!

$47189394 \mapsto 2D00D92$, so we would like to put $0D92$. Unfortunately putting 0 on the display is not possible as it is the only input that requires us to have the RBI pin at H (rather than at L) and we have no access to this pin on our board. Inputting a binary 0 (i.e. 0000 or LLLL) into the DM9368 results in the display being turned off rather than the display being set to 0 if the RBI pin is set to L (what it is fixes as) rather than H. Since all of the RBI pins are connected + grounded, there is no hope of trying to illegally access the pin to try to set it to high "illegally", either. So unfortunately, that display will have to go blank. However, punching in the rest we obtain the (somewhat satisfactory result):

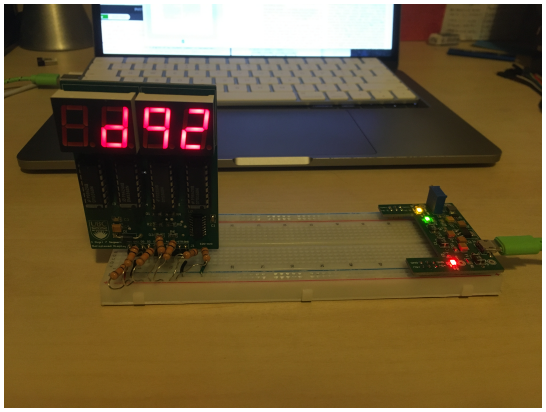

Figure 10: (Most of the) last 4 digits of my student number in hexadecimal.

2 Lab 2

2.1 Finding the code files

All of the code used in this lab can be found at `https://github.com/RioWeil/PHYS319-MSP430-assembly`.

2.2 Lab Objective

The objective today is to do what we did last lab (display our student number on the multiplexed breadboard 4-display) but now via programming a microcontroller (the MSP430).

2.3 Configuring the MSP to Work with my Mac

Before coming to lab, the assembler/flasher/drivers for communicating with the MSP430 Launchpad were installed on my computer following the Mac OSX instructions found on the course website. They can be referred to here `https://phas.ubc.ca/~kotlicki/Physics_319/tools_install_mac.pdf`. To communicate with the MSP430, I plug it into my computer (drivers now installed) and type in:
`mspdebug rf2500` into the terminal window. To use the assmebler (assuming I have a .asm assembly file written), I type in:
`naken430asm -o <name>.hex <name>.asm`
where <name> is the name of the .asm file. Then, to run it on the MSP430, I can type in:
`mspdebug rf2500`
`prog <name>.hex`
`Ctrl-D` (quite literally, hit Ctrl-D. Don't type it, future Rio.)
Where the first line opens the connection, the second line loads int he program, and the last line quits/exits.

2.4 MSP430 Specifications and Input/Outputs

Consulting the MSP430 specifications, we find that V_{OH} is given by $V_{CC} - 0.3V$ and V_{OL} is given by $V_{SS}+0.3V$[2] (where $V_{CC} = 1.8-3.6V$ during program execution but typically $3.3V$, and $V_{SS} = 0V$/grounded). Consulting the values we found for the max/min possible voltages for the TTL boards last lab (see section 1.4 from last lab), $V_{ILmax} = 0.8V$ and $V_{IHmin} = 2V$, so we can indeed see that the high/low output voltage states as from the MSP430 correspond to high/low input voltage states to the TTL (the 74HC139). However, we cannot conclude the same for the other way around; looking at the absolute maximum ratings for the inputs on the MSP430, we find that the voltage applied to V_{CC} to V_{SS} ranges from $-0.3Vto4.1V$, and the voltage applied to any pin ranges from $-0.3V$ to $V_{CC}+0.3V$. Beyond this range we run the risk of damaging the board. Looking at the low/high output ranges for the TTL, we have that the range is $0-0.5V$ for "low" and $2.7-5V$ for "high". While the low voltage state does not pose a problem, the high voltage output goes above the absolute maximum ratings for the voltage applied on any pin. The high output can be between $3.6V-5V$; above $3.6V$, we go above the maximum value for V_{CC} during program running. Above $4.1V$, we go above the absolute maximum value for V_{CC} and V_{SS}. Above. $4.4V$, we go above the absolute maximum value for any pin even assuming we set V_{CC} at the highest possible $4.1V$. In short, this would not be a good idea and we would run the risk of frying our precious board. In conclusion, we would conclude that wiring up the output of the MSP430 to the TTL board would be safe and have the correct voltage ranges (which is fortunate, as else this lab would prove impossible) but we may not wire the outputs of the TTL to the MSP430 unless we really didn't want that money back from the equipment deposit.

2.5 Wiring up the MSP430 to the display

The lab manual was followed to connect the correct pins. We remove the manual switchboard that we used last day, and wire up the pins as follows:

[2] Values found pg. 21 and 24 from `https://phas.ubc.ca/~kotlicki/Physics_319/msp430g2553.pdf`

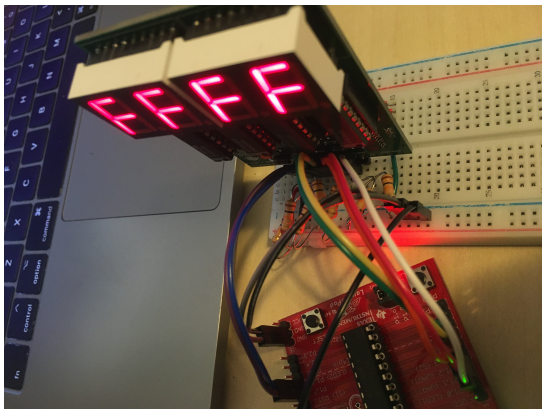

Figure 11: How the output pins from the MSP430 were connected to the breadboard and the 74HC139. Coincidentally, the initial display reads the final grade I'm going to get in this course.

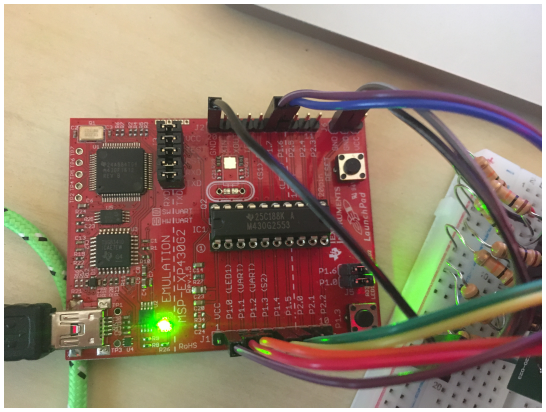

Figure 12: Close-up look at which output pins were used on the MSP. Color coded according to table below.

MSP340	Lab Breadboard w/ Display	Wire Colour
GND pins (3)	Ground Rail	Black/Grey
P1.0	Strobe	White
P1.1	A0	Red
P1.2	A1	Orange
P1.4	D0	Yellow
P1.5	D1	Green
P1.6	D2	Blue
P1.7	D3	Purple

The wiring is shown in the figures above.

2.6 Programming our display board

We are given a template of skeleton code that includes a file defining the names of addresses, bytes and words:

```
.include "msp430g2553.inc
```

Then, we have a command that tells the assembler where to put the program in memory:

```
ORG 0xC000
```

We then initialize the stack pointer/initialize the RAM for stack operation (the magic of what this means will be revealed in a later lab):
`mov #0x0400, SP`
We then disable the watchdog timer:
`mov.w #WDTPW|WDTHOLD, &WDTCTL`
Now finally moving into the realm of things we can actually understand well, we configure all P1.X pins at output pins, with the exception of P1.3: `mov.b #11110111b, &P1DIR`
Where the `mov.b` indicates we want to move the bit, the 7 numbers correspond to the each of the pins (from P1.7 to P1.0), the `b` indicates binary, and `&P1DIR` indicates the place we want to move the byte. The bits are all set to 1 (except for the P1.3 bit, which we do not use as output as we will later (in program 2) use it as an interrupt pin) to indicate that we want to configure all of these pins to output pins.
Up until this point, this code was all things that were given to us. Now, we write some assembly code (**screams**) to program the output pins. Fortunately, what is required of us here are fairly simple. To do so, we write lines of code of the form:
`mov.b #DDDD0AA1b, &P1OUT`
`mov.b #DDDD0AA0b, &P1OUT`
`mov.b #DDDD0AA1b, &P1OUT`
Where `DDDD` are the four bits that correspond P1.7-P1.4 (in order) and send information to data pins D3-D0 on the 74HC139, the P1.3 pin is always set to 0, `AA` are the two bits that correspond to P1.2-P1.1 and send information to the address pins A1-A0 on the 74HC139, and the last bit corresponds to P1.0 and controls the Strobe pin on the 74HC139. As before, `mov.b` indicates we are moving a bit to a new location and `&P1OUT` indicates we are outputting these values at the pins. Essentially, we set `AA` to specify which display we want to set (11 would be the first display, 10 would be the second, 01 would be the third, 00 would be the fourth) and `DDDD` specifies what number we want to set on the display (essentially, send the binary number that we want to show on the display). We then have the three lines of code as we first set the code with the strobe pin high, then set the strobe pin low to update the display, and then set it back to high so we can proceed to set the next display. If I wanted to represent 9 on the third display, this code would be:
`mov.b #10010011b, &P1OUT`
`mov.b #10010010b, &P1OUT`
`mov.b #10010011b, &P1OUT`
We do this process for each of the four seven segment displays. This is documented in the `setdisplay.asm` file so we will not go through the gory detail of each step here (and instead refer to the file linked with this document). At the end of the .asm file, we disable the CPU and load the reset vector with the location of the program start (after powerup/reset).
After following the procedure for compiling the program and getting the MSP430 to run it (see 2.2 for details) we get the desired result!

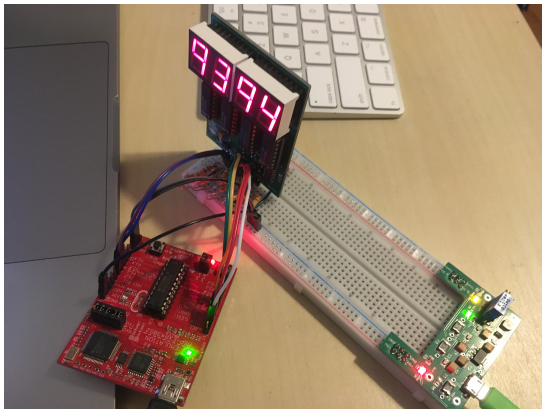

Figure 13: The displays show the last 4 digits of my student number, as desired.

```
.include "msp430g2553.inc"
    org 0xc000
start:
    mov.w #WDTPW|WDTHOLD, &WDTCTL
    mov.b #0x41, &P1DIR

    mov.w #0x01, r8

repeat:
    mov.b r8, &P1OUT
    xor.b #0x41, r8
    mov.w #60000, r9

waiter:
    dec r9
    jnz waiter
    jmp repeat
    org 0xfffe
    dw start
```

Figure 14: Screenshot of Program 1

2.7 Analyzing Assembly programs - Program 1 (Loops without loops)

Now we move to the portion of the lab where we analyze and modify assembly programs. The first program (aptly titled "Program1.asm") is a program that flashes the green and red LEDs on the MSP430-EXP430G2 board, one at a time, at around a frequency of 4 switches per second. We are interesting in increasing/decreasing the frequency of this flashing. To begin, it is helpful to note that P1.0 corresponds to the red LED and P1.6 to the green LED. The program file begins by importing the necessary names with `.include "msp430g2553.inc"`, and then telling the assembler where to put the program in memory with `org 0xc000`. Additionally, we note that at the end of the file, the commands `org 0xfffe` and `dw start` specifies that if we start or hit reset on the microcontroller, we look into that location of memory, and run the code in the `start` block.

Now we actually start running some code. The `start` block comes first so we run this first (sequentially). `mov.w #WDTPW|WDTHOLD, &WDTCTL` disables the watchdog counter, and then `mov.b #0x41, &P1DIR` configures P1.6 and P1.0 to outputs (these pins correspond to LEDs on the Launchpad). `mov.w #0x01, r8` then moves a 16-bit word #0x01 into register 8.

Now, we move on to the `repeat` block. First, `mov.b r8, &P1OUT` moves the byte in register 8 to the output; this causes one of the lights to turn on (the first time through the program, it will be the red LED corresponding to P1.0). Then, `xor.b #0x41, r8` XORs the value in register 8 with #0x41, causing the value in r8 to be changed to 01000000. Then, `mov.w #60000, r9` sends the decimal number 60000 to register 9.

Now, we go to the `waiter` block. First, `dec r9` decrements the value in register 9 by one. `jnz waiter` does one of two things; if the zero bit has not been set, we go back to the top of the waiter block and then start the decrement process again. If the zero bit is set to 1 (i.e. the value in r9 has just been decremented to zero) then we go to the next line, which is `jmp repeat`. This makes the code jump back to the repeat block. There, the light colour is switched, and then the whole process repeats. This is further elaborated on in the commented code. The general flow of the program is also sketched out in the chart below:

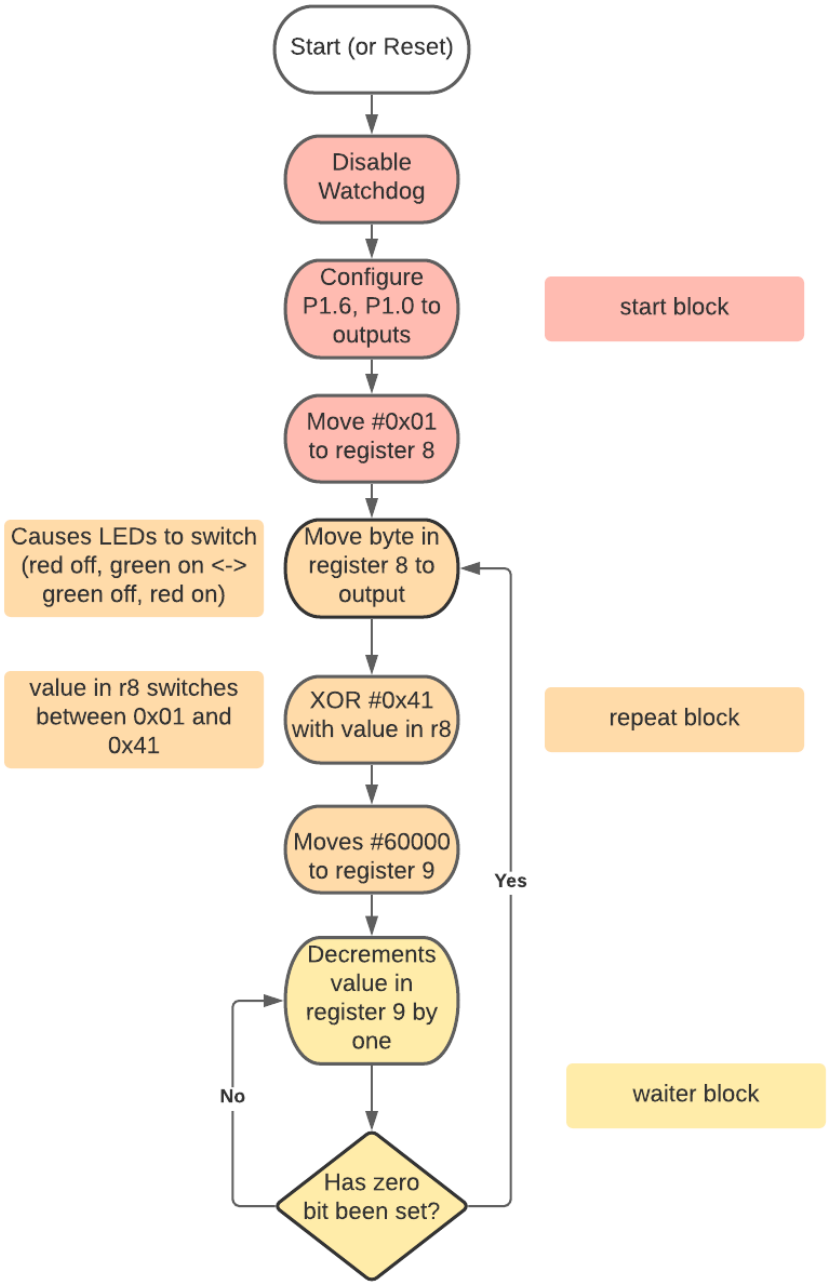

Figure 15: Flow chart for program 1. We omit the steps where we tell the assembler where to put the program in memory, where to look in memory in the case of a reset, and the import statement as those are not main components of what the program does.

```
#include "msp430g2553.inc"
    org 0x0C000
RESET:
    mov.w #0x400, sp
    mov.w #WDTPW|WDTHOLD,&WDTCTL
    mov.b #11110111b, &P1DIR
    mov.b #01001001b, &P1OUT
    mov.b #00001000b, &P1REN
    mov.b #00001000b, &P1IE
    mov.w #0x0049, R7
    mov.b R7, &P1OUT


    EINT
    bis.w #CPUOFF,SR




PUSH:
    xor.w #0000000001000001b, R7
    mov.b R7,&P1OUT
    bic.b #00001000b, &P1IFG

    reti

    org 0xffe4
    dw PUSH
    org 0xfffe
    dw RESET
```

Figure 16: Screenshot of Program 2

Now to modify the program! To make it flash twice as fast is simple; we simply change the value we write to register 9 in the repeat block from #60000 to #30000; this makes the decrement cycle/waiting time half as long as we have to decrement half as less. Making it flash slower is not as simple; we cannot just double the value we write to register 9 as we have a 16-bit integer limit, and the maximal value we can write there is $2^16 = 65536$. To get around this, we simply use another register and make another waiter code block (I called this waiter2, I know very original). We store #60000 in register 10 just like we do for register 9, and put in the second waiter block before the first so the program has to decrement through the value in register 10 and register 9 before looping back; this results in light switching that is twice as slow.
The original program and the half/double speed versions can be found in the github repository (all heavily commented).

2.8 Analyzing Assembly programs - Program 2 (Interrupts)

The next program has the function that it starts by turning both LEDs on, and then on presses of the P1.3 button, turns both off (and then on further presses, turns the two on, then off etc.). Let's dissect how this code works. The program file begins by importing the necessary names with `.include "msp430g2553.inc"`, and then telling the assembler where to put the program in memory with `org 0xc000`. Additionally, we note

that at the end of the file, the commands `org 0xfffe` and `dw RESET` specifies that if we start the program, or hit reset on the microcontroller, we look into that location of memory, and run the code in the `RESET` block. Additionally, the commands `org 0xffe4` and `dw PUSH` specify that if we interrupt the program, to look into that location of memory, and run the code in the `PUSH` block.
Getting into the bulk of the code, we start with the `RESET` block. First, the `mov.w #0x400, sp` initializes the stack pointer, and `mov.w #WDTPW|WDTHOLD, &WDTCTL` disables the watchdog. `mov.b #11110111b, &P1DIR` configures P1.3 to be an input and the rest of P1.0-1.7 to be outputs. `mov.b #01001001b, &P1OUT` Then sets P1.6, P1.3, and P1.0 to be high and the rest to low to output, which turns both LEDs on. `mov.b #00001000b, &P1REN` then enables the pulldown resistor on P1.3, and `mov.b #00001000b, &P1IE` enables input to P1.3 as an interrupt (helpful, because there is a button we can press to send an input to P1.3 on the Launchpad board!) Then, `mov.w #0x0049, R7` writes #0x0049 (0000000001001001 in binary) to register 7. `mov.b R7, &P1OUT` then sets P1.6, P1.3, P1.0 outputs to high. `EINT` enables a global interrupt (allows interrupts to occur) and `bis.w #CPUOFF,SR` sets bits in the status register that tells the CPU to halt. The proram is then stopped until the reset button is hit (in which case we start with the reset block of code once more) or the P1.3 button is hit and an interrupt occurs.
In this second case, the code in the `PUSH` block is run. `xor.w #0000000001000001b, R7` XORS 0000000001000001b with 0000000001001001b in R7, changing it to 0000000000001000b. Then, `mov.b R7,&P1OUT` moves this new value in R7 to the output, meaning both LEDs turn off as bot h P1.0 and P1.6 are set to low. `bic.b #00001000b, &P1IFG` clears the bit that holds the interrupt flag so the processor does not try to deal with it again. `reti` then indicates a return from interrupt. A future press of the reset button will start us back at `RESET`, and another press of the P1.3 button will send us back to the `PUSH` code, wherein R7 gets XORd again, now resulting in both LEDs turning on when the byte is moved to the output. On future presses of the switch, the LEDs will therefore turn on and off. These flow of the program is represented visually in the flow chart below.

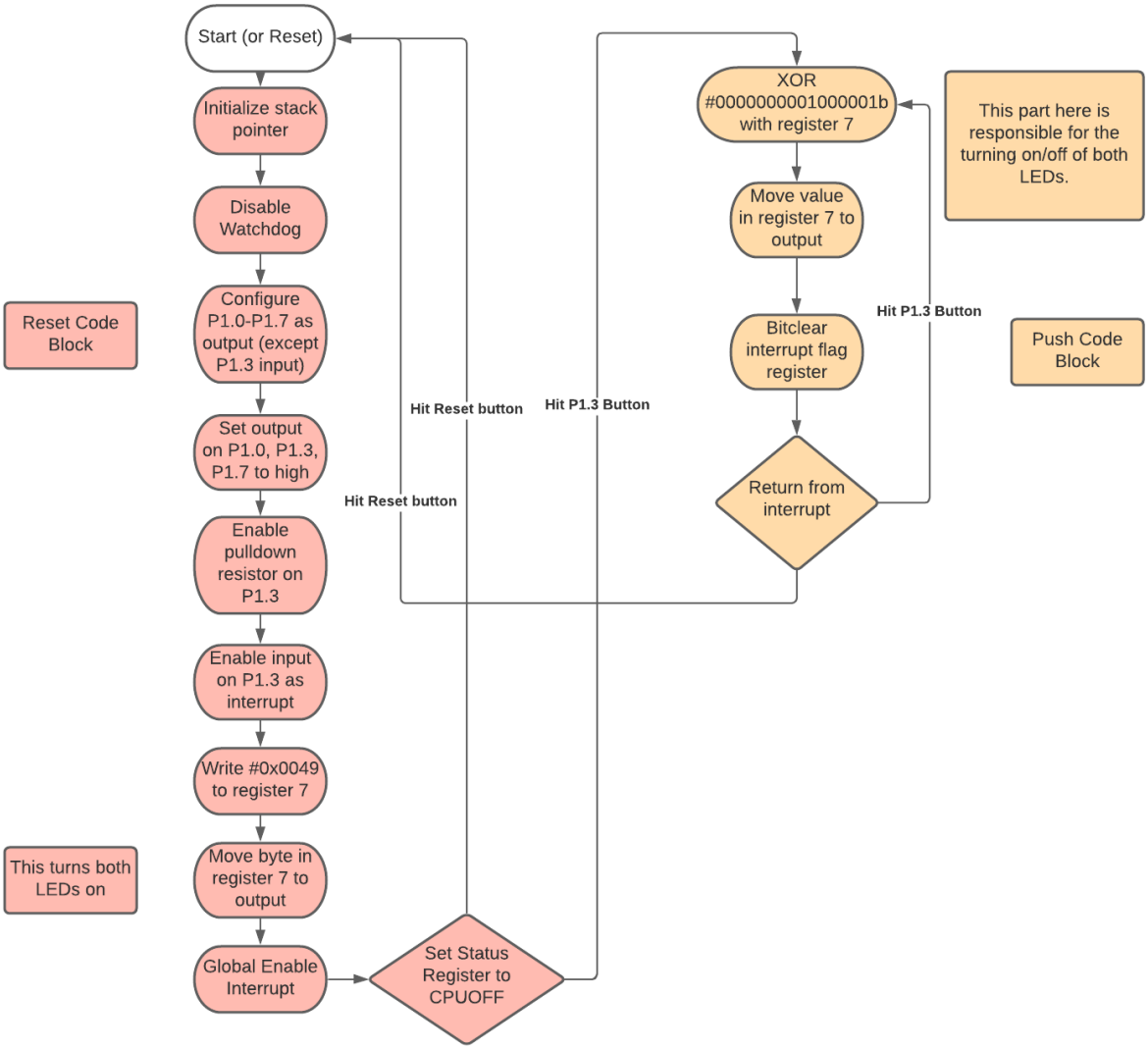

Figure 17: Flow chart for program 2.

Now that we understand the original program, we again get into modifying it. We want to change things such that on pushes of the button, the LEDs will cycle through:

(1) Both off

(2) Red on/green off

(3) Green on/red off

(4) Both on

(5) Start at (1) again.

In order to do this, I added:
`mov.w #0000000000000001b, R8`
`mov.w #0000000001000000b, R9` At the end of the `RESET` code block (before `EINT`) and replaced `xor.w #0000000001000001b, R7` at the beginning of the `PUSH` code block with:
`xor.w R9, R8`
`xor.w R8, R7` this has the desired result, and we didn't need to write any more functions. How does it work? Well, the value in register 9 is fixed as 01000000 (I'll only write the latter byte as that's what's important), and every time the button is pressed, the value in register 8 get's XORd with this. Hence, the value in register 8 constantly is switching between 00000001 and 01000001. Additionally, each cycle, we have that the value in R7 (which is what gets written to the output) gets XORd with what is in register 8. This allows the values in register 7 to cycle through the four necessary states. Since this might be a bit unclear in the way I've worded it, let's just walk through what one 4 button cycle looks like. In the following description, I again only write the last 8 bits of the words in the registers (the rest are all zeros and are insiginificant for the program):

1. Initially, R7 is set to 01001001, R8 to 00000001 in the RESET block. Both LEDs are on at this point (P1.0, P1.6 both high).
2. We hit the button the first time. First, 00000001 in R8 gets XORd with 01000000 in R9. The resulting value in R8 is 01000001.
3. 01001001 in R7 is XORd with 0100001 in R8. This results in 00001000 in R7, which gets moved to the output. Both LEDs are off (P1.0 and P1.6 both low).
4. The button is hit a second time. 01000001 in R8 is XORd with 01000000 in R9. The resulting value in R8 is 00000001 in R8.
5. 00001000 in R7 is XORd with 00000001 in R8. This results in 00001001 in R7, which is moved to the output. Just the red LED turns on (P1.0 high, P1.6 low).
6. The button is hit a third time. 00000001 in R8 gets XORd with 01000000 in R9. The resulting value in R8 is 0100001.
7. 00001001 in R7 gets XORd with 01000001 in R8. This results in 01001000 in R7, which is moved to the output. Just the green LED turns on (P1.0 low, P1.6 high).
8. 01000001 in R8 gets XORd with 01000000 in R9. The resulting value in R8 is 00000001.
9. 01001000 in R7 gets XORd with 00000001 in R8. This results in 01001001 in R7, which gets moved to the output. Both LEDs turn on (P1.0, P1.6 both high).
10. The states of all registers in steps 1/9 are identical and the cycle hence repeats upon further presses of the button

The original program and the modified version to include the cycle of actions when the P1.3 interrupt button is pressed can be found in the github repository.

2.9 Bonus - Button press counter

As a challenge, I tried to write a program that would count the number of times the interrupt P1.3 button was pressed and display this on the 7-segment displays. Essentially, every time the button was pressed the values in registers would be incremented by some amount, and would update the displays (with a coded "if statement" that would handle cases where the digits in a certain place surpassed 9). A lengthy discussion will be omitted here as the file is heavily commented and well-explained in terms of how the code runs (also, I should probably start on my 5 other assignments). A video of me using the program and the code file `counter.asm` can be found in the github repository.

2.10 Bonus - Breaking(?) something

In the process of trying to design the above counter program, I somehow managed to get the displays to show 0 (even though this shouldn't be hypothetically possible/accessible as per the discussion in lab 1). This is especially strange because nothing I programmed should have hypothetically caused this behavior; I only was trying to make the display read 4-blank-blank-blank. This adds nothing to the discussion, just thought it was kind of funny.

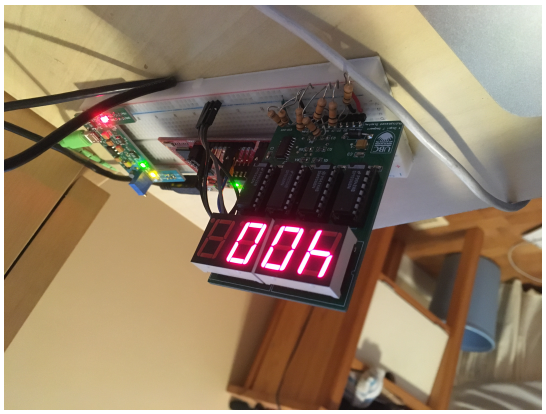

Figure 18: Me somehow getting the display to show 0 even though it shouldn't be able to. Either I'm an electronics wizard, or these ICs are ****ing with me.