

Jacob Micros Assignment

by Jacob Brown

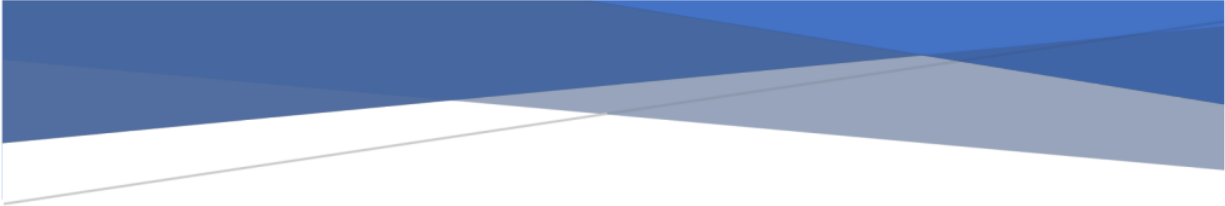
Submission date: 28-Nov-2022 02:11PM (UTC+0000)

Submission ID: 190319304

File name: 32800_Jacob_Brown_Jacob_Micros_Assignment_466098_326858693.docx (176.89K)

Word count: 6169

Character count: 29606



Microcontrollers, microprocessors, and their processes

Micros, Peripherals, and Interfaces

Abstract

In this report, we will summarize and conduct an investigation into the function and subsequent applications for microcontrollers and microprocessors along with methods of interfacing and examples of assembly code being utilized to produce a result requiring a multilayered design process. Further detailing and explaining the function of a section of assembly code will also be reported on in the later sections of this report

Jacob Brown
2112087

Contents

Abstract..... 0

Section 1 – Microprocessors and Microcontrollers 2

Section 2 - Interfacing and Interfacing Techniques 4

Assembly Language Demonstration 9

Bibliography 34

DISCLAIMER: CODE IS NOT INCLUDED IN MY WORK BUT AS IT WOULD HAVE BEEN ATTACHED AS A SOURCE, BUT TURNITIN DIDN'T ALLOW IT.

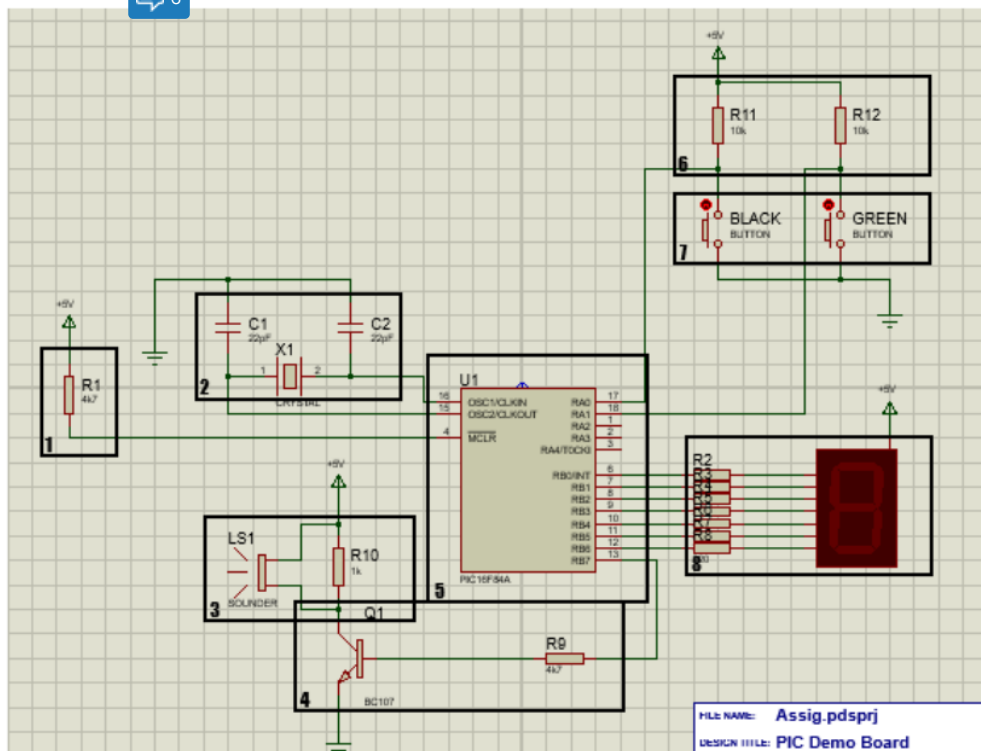
Section 1 – Microprocessors and Microcontrollers

Generally, Microprocessors can be broken down into several groups based off what their specification dictates, one of the largest defining features of the microprocessor architecture is the structure of the memory relative to the processing unit within, generally classed between two archetypes, these being Von Neuman and Harvard. Von Neuman's main feature is a unified memory bank that allows for the conducting of processing and functions wherein a single set of buses are required to transfer Data, instructions, and addresses, then to be relayed out of the CPU. The primary benefit of Von Neuman is that it requires fewer components due to the simplified structure, which is a large part in understanding why it is so commonplace in the contemporary market, while still maintaining a good rate of throughput, when compared to what components allow. As a result, all but highly specialized and intensive machines for purpose-built applications prefer the Von Neuman model. Harvard architecture is defined by the inclusion of two separate memory banks, these being purposed towards data and program respectively[1]. The complications that arise from this are largely twofold, that being the increased size and increased cost, as not only is an additional memory bank required but also all the accompanying infrastructure, primarily referring to buses, which the material costs and size costs mean that for the same cost, a Von Neuman processor is generally both smaller and more cost effective. Though Harvard will generally have a greater throughput of instructions and executions due to the simultaneous transfer that is characteristic of this style and shared load between buses also aids in the further in not only increasing throughput but also reducing in data contention and related (potential) timing. It is worth noting that as general computational logic is growing more and more intensive, Harvard style architecture is growing to be increasingly more commonplace and thus, cheaper due to the merits of industrial theory, beginning to surpass the equivalent losses to component complexity and cost when considered in an exclusive context, despite their potential falloffs in places like with standard instruction set space occupation alongside those previously mentioned[2].

Further divisions can be drawn along bitwise processing capability, depending on the amount of data that is expected to be handled by a particular system, more powerful specifications may be utilized. While generally more power intensive, 64 and 32-bit microprocessors are generally the high end of widely employed non computationally exclusive uses in the commercial sphere and can process exponentially more than the other commercially viable alternatives such as 16-bit and are generally employed for usage in systems with large data streams such as audio player equipment. While smaller less high requirement systems tend to lean towards 8-bit systems[3], which are largely employed with a multitude of embedded systems as they still retain high levels function relative to the requirements of their environment.

In relation to the previous two statements, we can begin to explain why, not simply state that the PIC16F84A has the specification set that it does, featuring an 8-Bit microcontroller structured in a Harvard style[4], which would imply that it is intended for low power consumption and a more reliable throughput. As such, we can assume that the intended application for these particular PIC chips would be in energy conservative, high reliability systems with a capability to work with RISC architecture (As opposed to Von Neuman CISC standard) such as battery reliant systems or potentially systems that require unique or complex timing mechanisms or coprocessing methods to function effectively.

Bellow, figure 1



Above in figure 1 we have a general layout of the circuit segmented into numbered boxes for ease of explanation.

Box 1: is simply a resistor to build some resistance into the Master Clear pin on the PIC, largely employed while programming to clear the device if necessary, it includes a resistor so that the voltage emitted during coding doesn't [7] onto the VDD rail, equally, it can be required to protect against spikes or noise causing an unwanted trigger, a resistor, such as the 4k7 used here is not excessively high but might cause problems in the event of VDD rise during programming. Alternatively, this can also be an easy physical reset to use in the event that it is required [5].

Box 2: Features a common employment of a clock system, this being made of a crystal oscillator and two capacitors, which are common to have employed in the event of power malfunction that would otherwise result in the failure of the clock in its functioning as an accurate measure of consistent time units. [9]

Box 3: Contained here is a pairing of a piezo buzzer and resistor [10] here the resistor is used to reduce the sound level a little instead of changing the voltage that gets driven directly to the piezo, as these buzzers are a risk to aural health unless properly managed.

Box 4: Contained herein are a resistor and a transistor, used to direct current towards the piezo while also providing a line to base, while not letting the 4k7 resistor interfere with the operation of the piezo. [11]

Box 5: Contains the microcontroller PIC itself, just separated from the rest of the components due to its importance and ease of explanation. The combination of these [12]

Box 6 & 7: These two boxes are easier to explain together, while 6 contains resistors and 7 contains buttons, which allow for a non-excessive amount of voltage to allow the microcontroller to correctly interpret the button activation relative to each vertical pair.

Box 8: Contains the output device in this instance, the segmented display, relative resistors are included to avoid damage to the corresponding segment from the incoming (Relative) voltage and allow for the function of the segmented display as intended



Section 2 - Interfacing and Interfacing Techniques

The initial count program is a simple set of functions that increment the binary value of a value stored in the data registers, that is then incremented after a pause period of roughly equivalent to 200ms, it repeatedly displays this outputted numeric value into binary along the 7 different inputs leading into the segmented display, while not displaying the actual value on the display unit, it does display the correct binary value. As it is required to first count from 0 to F in Hexadecimal (or 0x), the first noted approach must follow along the process of assigning the binary outputs to be displayed along the relevant ports of the PIC and call them in sequence, the following is a demonstration of a higher-level pseudo representation of the process needed.

```
Main(void){  
    int[16].init(LEDVALUES) =  
[0x40,0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x10,0x08,0x03,0x46,0x21,0x06,0x0e);  
    for(int.init(i) = 0;i++;i<=17);  
        PortB_Out(LEDVALUES[i]);  
    endfor;  
    i = 0;  
}
```



In this high-level example, the logic is simple and in general a rather short program as it is just a routine count and display sequence, however, when coded in assembly it looks a lot longer, in a pseudo assembly example we would expect it to look something along these lines.

```
0000: MOVLW 00  
0001: MOVWF 0A  
0002: GOTO 004  
0003: NOP  
0004: MOVF 03,W ;Start of variable declarations  
0005: ANDLW 1F  
0006: MOVWF 03
```

```
0007: MOVLW 40
0008: MOVWF 10
0009: MOVLW 79
000A: MOVWF 11
000B: MOVLW 24
000C: MOVWF 12
000D: MOVLW 30
000E: MOVWF 13
000F: MOVLW 19
0010: MOVWF 14
0011: MOVLW 12
0012: MOVWF 15
0013: MOVLW 02
0014: MOVWF 16
0015: MOVLW 78
0016: MOVWF 17
0017: CLRF 18
0018: MOVLW 10
0019: MOVWF 19
001A: MOVLW 08
001B: MOVWF 1A
001C: MOVLW 03
001D: MOVWF 1B
001E: MOVLW 46
001F: MOVWF 1C
0020: MOVLW 21
0021: MOVWF 1D
0022: MOVLW 06
0023: MOVWF 1E
0024: MOVLW 0E
0025: MOVWF 1F ;End of variable declarations
```

```

0026: BSF  03.5      ;Output LED 1
0027: CLRF  06
0028: BCF  03.5
0029: MOVF  10,W
002A: MOVWF 06      ;End of Output for LED 1  16
002B: BSF  03.5      ;Output for LED2 and so on every 5 lines from here until GOTO in 076
002C: CLRF  06
002D: BCF  03.5  17
002E: MOVF  11,W
002F: MOVWF 06
0030: BSF  03.5
0031: CLRF  06
0032: BCF  03.5
0033: MOVF  12,W
0034: MOVWF 06
0035: BSF  03.5
0036: CLRF  06
0037: BCF  03.5
0038: MOVF  13,W
0039: MOVWF 06
003A: BSF  03.5
003B: CLRF  06
003C: BCF  03.5
003D: MOVF  14,W
003E: MOVWF 06
003F: BSF  03.5
0040: CLRF  06
0041: BCF  03.5
0042: MOVF  15,W
0043: MOVWF 06

```

0044: BSF 03.5
0045: CLRF 06
0046: BCF 03.5
0047: MOVF 16,W
0048: MOVWF 06
0049: BSF 03.5
004A: CLRF 06
004B: BCF 03.5
004C: MOVF 17,W
004D: MOVWF 06
004E: BSF 03.5
004F: CLRF 06
0050: BCF 03.5
0051: MOVF 18,W
0052: MOVWF 06
0053: BSF 03.5
0054: CLRF 06
0055: BCF 03.5
0056: MOVF 19,W
0057: MOVWF 06
0058: BSF 03.5
0059: CLRF 06
005A: BCF 03.5
005B: MOVF 1A,W
005C: MOVWF 06
005D: BSF 03.5
005E: CLRF 06
005F: BCF 03.5
0060: MOVF 1B,W
0061: MOVWF 06

```

0062: BSF  03.5
0063: CLRF 06
0064: BCF  03.5
0065: MOVF 1C,W
0066: MOVWF 06
0067: BSF  03.5
0068: CLRF 06
0069: BCF  03.5
006A: MOVF 1D,W
006B: MOVWF 06
006C: BSF  03.5
006D: CLRF 06
006E: BCF  03.5
006F: MOVF 1E,W
0070: MOVWF 06
0071: BSF  03.5
0072: CLRF 06
0073: BCF  03.5
0074: MOVF 1F,W
0075: MOVWF 06
0076: GOTO 026      ;Return to start of main

```

The above code is rather inefficient; however, it accomplishes the simple task of displaying in sequence the characters required, if  18, assembly language tends to span a greater line count than higher counterparts both in short and more complex tasks, as space efficiency in a similar manner is remarkably difficult to achieve when siting so many exact commands must call to a greater specificity down to the individual register. This can then extrapolate out to form more complex counting mechanisms such as the one later required in this brief.

This is essentially just a recursive set of instructions telling the chip to set the bits stored in value 3.5  19 to store the value of the LED output needed, such as 1F representing 0x0e. then move it to working after clearing the register 06, used generally as a temporary  20 table. This just iterates through until it reaches the end of the code, at which point it returns to command 026 and resumes displaying, avoiding wasting time by reassigning registers that act as constants and wasting computational throughput.

Assembly Language Demonstration

Before proceeding with further detailing of the assembly language, it is worth noting that these assembly code iterations are written with line index referencing enabled, which is required to run these exact code items, without which it would produce compiler errors, in order to execute on default setting proteus, it would be necessary to enable this setting or to remove the line addresses that prefix the assembly lines.

Task 1

Task 1 desires us to create a hexadecimal count that goes on ad infinitum, with the count occurring every 500ms, a relatively simple task that could be done much more efficiently than the bellow C style pseudo via the use of recursive programming, in this instance.

```
#include <main.h>
```

```
int LED0 = 0x40;
```

```
int LED1 = 0x79;
```

```
int LED2 = 0x24;
```

```
int LED3 = 0x30;
```

```
int LED4 = 0x19;
```

```
int LED5 = 0x12;
```

```
int LED6 = 0x02;
```

```
int LED7 = 0x78;
```

```
int LED8 = 0x00;
```

```
int LED9 = 0x10;
```

```
int LEDA = 0x08;
```

```
int LEDB = 0x03;
```

```
int LEDC = 0x46;
```

```
int LEDD = 0x21;
```

```
int LEDE = 0x06;
```

```
int LEDF = 0x0e;
```

```
void main()
```

```
{
```

```
    while(TRUE)
```

```
    {
```

```
        output_B(LED0);
```

```

delay_ms(500);
output_B(LED1);
delay_ms(500);
output_B(LED2);
delay_ms(500);
output_B(LED3);
delay_ms(500);
output_B(LED4);
delay_ms(500);
output_B(LED5);
delay_ms(500);
output_B(LED6);
delay_ms(500);
output_B(LED7);
delay_ms(500);
output_B(LED8);
delay_ms(500);
output_B(LED9);
delay_ms(500);
output_B(LEDA);
delay_ms(500);
output_B(LEDB);
delay_ms(500);
output_B(LEDC);
delay_ms(500);
output_B(LEDD);
delay_ms(500);
output_B(LEDE);
delay_ms(500);
output_B(LEDFF);
delay_ms(500);

```

```
}  
}
```

The above would function as a manually coded system for the above-mentioned task, which would look similar to this in assembly.

```
0000: MOVLW 00  
0001: MOVWF 0A      ;Initialize register and Working file to "0A"  
0002: GOTO 018      ;Go to index 018  
0003: NOP           ;No operation breaker for continuity  
0004: MOVLW 21      ;Here until line 017 are also an initialization of sorts, with the purpose of  
                    ;acting as a delay time equivalent to an oscillator and is apparently standard practice for this pic  
                    ;model. – Required to determine the pulse required for outputs and other pulses  
0005: MOVWF 04  
0006: MOVF 00,W  
0007: BTFSC 03.2  
0008: GOTO 017  
0009: MOVLW 01  
000A: MOVWF 0D  
000B: CLRF 0C  
000C: DECFSZ 0C,F  
000D: GOTO 00C  
000E: DECFSZ 0D,F  
000F: GOTO 00B  
0010: MOVLW 4A  
0011: MOVWF 0C  
0012: DECFSZ 0C,F  
0013: GOTO 012  
0014: GOTO 015  
0015: DECFSZ 00,F  
0016: GOTO 009  
0017: RETURN        ;end of the delay period  
0018: MOVF 03,W      ;Here until 039 is the sequence for assigning registers their constant values  
                    ;for the LEDs
```

0019: ANDLW 1F
001A: MOVWF 03
001B: MOVLW 40
001C: MOVWF 10
001D: MOVLW 79
001E: MOVWF 11
001F: MOVLW 24
0020: MOVWF 12
0021: MOVLW 30
0022: MOVWF 13
0023: MOVLW 19
0024: MOVWF 14
0025: MOVLW 12
0026: MOVWF 15
0027: MOVLW 02
0028: MOVWF 16
0029: MOVLW 78
002A: MOVWF 17
002B: CLRF 18
002C: MOVLW 10
002D: MOVWF 19
002E: MOVLW 08
002F: MOVWF 1A
0030: MOVLW 03
0031: MOVWF 1B
0032: MOVLW 46
0033: MOVWF 1C
0034: MOVLW 21
0035: MOVWF 1D
0036: MOVLW 06
0037: MOVWF 1E

```

0038: MOVLW 0E
0039: MOVWF 1F      ;End of assignment
003A: BSF  03.5     ;Sets bit 03,5
003B: CLRF  06      ;Clears content of 006 register
003C: BCF  03.5     ;Clears bit 3.5, Creating a pulse
003D: MOVF  10,W
003E: MOVWF 06      ;Moves previous working register to act as an output for a prior assigned
value (As assigned in the previous several lines of assignment protocols)
003F: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
0040: MOVWF 20
0041: MOVLW FA
0042: MOVWF 21
0043: CALL  004
0044: DECFSZ 20,F
0045: GOTO  041      ;Primitive end for delay of ~500ms
0046: BSF  03.5     ;activate for Pseudo Equivalent LED1
0047: CLRF  06
0048: BCF  03.5
0049: MOVF  11,W
004A: MOVWF 06      ;End of LED1 Output Sequence
004B: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
004C: MOVWF 20
004D: MOVLW FA
004E: MOVWF 21
004F: CALL  004
0050: DECFSZ 20,F
0051: GOTO  04D      ;Primitive end for delay of ~500ms
0052: BSF  03.5     ;activate for Pseudo Equivalent LED2
0053: CLRF  06
0054: BCF  03.5
0055: MOVF  12,W

```



```

0056: MOVWF 06      ;End of LED2 Output Sequence
0057: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
0058: MOVWF 20
0059: MOVLW FA
005A: MOVWF 21
005B: CALL 004
005C: DECFSZ 20,F
005D: GOTO 059      ;Primitive end for delay of ~500ms
005E: BSF 03.5      ;activate for Pseudo Equivalent LED3
005F: CLRF 06
0060: BCF 03.5
0061: MOVF 13,W
0062: MOVWF 06      ;End of LED3 Output Sequence
0063: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
0064: MOVWF 20
0065: MOVLW FA
0066: MOVWF 21
0067: CALL 004
0068: DECFSZ 20,F
0069: GOTO 065      ;Primitive end for delay of ~500ms
006A: BSF 03.5      ;activate for Pseudo Equivalent LED4
006B: CLRF 06
006C: BCF 03.5
006D: MOVF 14,W
006E: MOVWF 06      ;End of LED4 Output Sequence
006F: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
0070: MOVWF 20
0071: MOVLW FA
0072: MOVWF 21
0073: CALL 004
0074: DECFSZ 20,F

```

```

0075: GOTO 071      ;Primitive end for delay of ~500ms
0076: BSF 03.5      ;activate for Pseudo Equivalent LED5
0077: CLRF 06
0078: BCF 03.5
0079: MOVF 15,W
007A: MOVWF 06      ;End of LED5 Output Sequence
007B: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
007C: MOVWF 20
007D: MOVLW FA
007E: MOVWF 21
007F: CALL 004
0080: DECFSZ 20,F
0081: GOTO 07D      ;Primitive end for delay of ~500ms
0082: BSF 03.5      ;activate for Pseudo Equivalent LED6
0083: CLRF 06
0084: BCF 03.5
0085: MOVF 16,W
0086: MOVWF 06      ;End of LED6 Output Sequence
0087: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
0088: MOVWF 20
0089: MOVLW FA
008A: MOVWF 21
008B: CALL 004
008C: DECFSZ 20,F
008D: GOTO 089      Primitive end for delay of ~500ms
008E: BSF 03.5      ;activate for Pseudo Equivalent LED7
008F: CLRF 06
0090: BCF 03.5
0091: MOVF 17,W
0092: MOVWF 06      ;End of LED7 Output Sequence
0093: MOVLW 02      ;Primitive start for time wasting delay of ~500ms

```

```

0094: MOVWF 20
0095: MOVLW FA
0096: MOVWF 21
0097: CALL 004
0098: DECFSZ 20,F
0099: GOTO 095 ;Primitive end for delay of ~500ms
009A: BSF 03.5 ;activate for Pseudo Equivalent LED8
009B: CLRF 06
009C: BCF 03.5
009D: MOVF 18,W
009E: MOVWF 06 ;End of LED8 Output Sequence
009F: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00A0: MOVWF 20
00A1: MOVLW FA
00A2: MOVWF 21
00A3: CALL 004
00A4: DECFSZ 20,F
00A5: GOTO 0A1 ;Primitive end for delay of ~500ms
00A6: BSF 03.5 ;activate for Pseudo Equivalent LED9
00A7: CLRF 06
00A8: BCF 03.5
00A9: MOVF 19,W
00AA: MOVWF 06 ;End of LED9 Output Sequence
00AB: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00AC: MOVWF 20
00AD: MOVLW FA
00AE: MOVWF 21
00AF: CALL 004
00B0: DECFSZ 20,F
00B1: GOTO 0AD ;Primitive end for delay of ~500ms
00B2: BSF 03.5 ;activate for Pseudo Equivalent LEDA

```

```

00B3: CLRF 06
00B4: BCF 03.5
00B5: MOVF 1A,W
00B6: MOVWF 06 ;End of LEDA Output Sequence
00B7: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00B8: MOVWF 20
00B9: MOVLW FA
00BA: MOVWF 21
00BB: CALL 004
00BC: DECFSZ 20,F
00BD: GOTO 0B9 ;Primitive end for delay of ~500ms
00BE: BSF 03.5 ;activate for Pseudo Equivalent LEDB
00BF: CLRF 06
00C0: BCF 03.5
00C1: MOVF 1B,W
00C2: MOVWF 06 ;End of LEDB Output Sequence
00C3: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00C4: MOVWF 20
00C5: MOVLW FA
00C6: MOVWF 21
00C7: CALL 004
00C8: DECFSZ 20,F
00C9: GOTO 0C5 ;Primitive end for delay of ~500ms
00CA: BSF 03.5 ;activate for Pseudo Equivalent LEDC
00CB: CLRF 06
00CC: BCF 03.5
00CD: MOVF 1C,W
00CE: MOVWF 06 ;End of LEDC Output Sequence
00CF: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00D0: MOVWF 20
00D1: MOVLW FA

```

```

00D2: MOVWF 21
00D3: CALL 004
00D4: DECFSZ 20,F
00D5: GOTO 0D1 ;Primitive end for delay of ~500ms
00D6: BSF 03.5 ;activate for Pseudo Equivalent LEDD
00D7: CLRF 06
00D8: BCF 03.5
00D9: MOVF 1D,W
00DA: MOVWF 06 ;End of LEDD Output Sequence
00DB: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00DC: MOVWF 20
00DD: MOVLW FA
00DE: MOVWF 21
00DF: CALL 004
00E0: DECFSZ 20,F
00E1: GOTO 0DD ;Primitive end for delay of ~500ms
00E2: BSF 03.5 ;activate for Pseudo Equivalent LEDE
00E3: CLRF 06
00E4: BCF 03.5
00E5: MOVF 1E,W
00E6: MOVWF 06 ;End of LEDE Output Sequence
00E7: MOVLW 02 ;Primitive start for time wasting delay of ~500ms
00E8: MOVWF 20
00E9: MOVLW FA
00EA: MOVWF 21
00EB: CALL 004
00EC: DECFSZ 20,F
00ED: GOTO 0E9 ;Primitive end for delay of ~500ms
00EE: BSF 03.5 ;activate for Pseudo Equivalent LEDF
00EF: CLRF 06
00F0: BCF 03.5

```

```

00F1: MOVF 1F,W
00F2: MOVWF 06      ;End of LEDE Output Sequence
00F3: MOVLW 02      ;Primitive start for time wasting delay of ~500ms
00F4: MOVWF 20
00F5: MOVLW FA
00F6: MOVWF 21
00F7: CALL 004
00F8: DECFSZ 20,F
00F9: GOTO 0F5      ;Primitive end for delay of ~500ms
00FA: GOTO 03A

```

Task 2

This task asks for the button press to active a twp-part conditional statement, this is possible in pseudo logic with either a case or two while statements, depending on how the coder prefers, in this instance the code has been written in the style of the latter option.

```
#include <main.h>
```

```
int LED1 = 0x79;
```

```
int LED2 = 0x24;
```

```
unsigned int holding = 0;
```

```
unsigned int holding2 = 0;
```

```
void main()
```

```
{
```

```
    while(TRUE)
```

```
    {
```

```
        holding = input(PIN_A0);
```

```
        holding2 = input(PIN_A1);
```

```
        if(holding == 1){
```

```
            while(holding == 1){
```

```
                output_B(LED1);
```

```

        holding = input(PIN_A0);
    }
    output_B(0xFF);
}
if(holding2 == 1){
    while(holding2 == 1){
        output_B(LED2);
        holding2 = input(PIN_A1);
    }
    output_B(0xFF);
}
}
}

```

The above removes the method in Task 1 of assigning all of the LED segments and their respective numbers and instead only defines the two units requested to be displayed, simply transcribing this over from C to ASM would look like the following.

```

0000: MOVLW 00
0001: MOVWF 0A      ;Initialize register and Working file to "0A"
0002: GOTO 004      ;Go to index 018
0003: NOP           ;No operation breaker for continuity
0004: MOVF 03,W     ;Here until 00C creates a couple of variable assignments, the equivalent to
LED1, LED2, Holding and Holding2
0005: ANDLW 1F
0006: MOVWF 03
0007: MOVLW 79
0008: MOVWF 10
0009: MOVLW 24
000A: MOVWF 11
000B: CLRF 12
000C: CLRF 13      ;end of Declarations
000D: BSF 03.5      ;Here until 012 performs the equivalent of an input register by setting bits
equivalent to the input expected from the button if not clear to skip

```

```

000E: BSF  05.0
000F: BCF  03.5
0010: CLRF 12
0011: BTFSC 05.0
0012: INCF 12,F      ;End of first input check, seen by incrementing the flag register
0013: BSF  03.5      ;Second flag input check, see Lines 00D to 012
0014: BSF  05.1
0015: BCF  03.5
0016: CLRF 13
0017: BTFSC 05.1
0018: INCF 13,F      ;End of first input check, seen by incrementing the flag register
0019: DECFSZ 12,W    ;Check flag variable in 12 if zero then skip
001A: GOTO  02E      ; Avoid the loop if above is false with GOTO pass-over
001B: DECFSZ 12,W    ;Similar logic as above, except within a loop for this line and bellow
001C: GOTO  029
001D: BSF  03.5      ;Output sequence start, see previous task
001E: CLRF 06
001F: BCF  03.5
0020: MOVF 10,W
0021: MOVWF 06      ;end of output sequence
0022: BSF  03.5      ;Check input as in 00D
0023: BSF  05.0
0024: BCF  03.5
0025: CLRF 12
0026: BTFSC 05.0
0027: INCF 12,F
0028: GOTO  01B      ;Loop close for inner nested while
0029: BSF  03.5      ;Output equivalent to empty segment
002A: CLRF 06
002B: MOVLW FF
002C: BCF  03.5

```

```

002D: MOVWF 06
002E: DECFSZ 13,W    ;If check for second flag, pass loop if false
002F: GOTO 043
0030: DECFSZ 13,W    ;While version of the same if logic as above
0031: GOTO 03E
0032: BSF 03.5        ;Output sequence, see previous task
0033: CLRF 06
0034: BCF 03.5
0035: MOVF 11,W
0036: MOVWF 06        ;End of Output Sequence
0037: BSF 03.5        ;Start of input Sequence
0038: BSF 05.1
0039: BCF 03.5
003A: CLRF 13
003B: BTFSC 05.1
003C: INCF 13,F       ;End of input Sequence
003D: GOTO 030        ;Loop Close
003E: BSF 03.5        ;Output empty
003F: CLRF 06
0040: MOVLW FF
0041: BCF 03.5
0042: MOVWF 06        ;End of Output
0043: GOTO 00D        Return to Main loop

```

The fewer delays, outputs and variable assignments greatly reduces the bloat of the system and, through the refinements previously mentioned, applying similar checks to avoid manual recursion would save a great amount of artificial complexity and instruction space as a whole.

Task 3

The same logic that is applied to the process for task 2 can essentially be reused for this particular task, as it only requires that two outputs are used and “latched” in which case, the logical trip clear can just be removed, leaving the data in a trigger state (in this case – low), as demonstrated by the pseudo code bellow.

```
#include <main.h>
```

```

int LEDNULL = 0xFF;
int LED8 = 0x00;

unsigned int holding = 0;
unsigned int holding2 = 0;

void main()
{

    while(TRUE)
    {
        holding = input(PIN_A0);
        holding2 = input(PIN_A1);
        if(holding == 1){
            while(holding == 1){
                output_B(LEDNULL);
                holding = input(PIN_A0);
            }

        }
        if(holding2 == 1){
            while(holding2 == 1){
                output_B(LED8);
                holding2 = input(PIN_A1);
            }

        }

    }

}

```

This can be further ported to assembly as similar to the following assembly language used in proteus.

```
0000: MOVLW 00
0001: MOVWF 0A      ;Initialize register and Working file to "0A"
0002: GOTO 004      ;Go to index 004
0003: NOP           ;No operation breaker for continuity
0004: MOVF 03,W     ;Start of assignments of variables to registers, LEDNULL, LED8 and Flags
0005: ANDLW 1F
0006: MOVWF 03
0007: MOVLW FF
0008: MOVWF 10
0009: CLRF 11
000A: CLRF 12
000B: CLRF 13      ;End of assignments for the variables
000C: BSF 03.5      ;Check of inputs from button
000D: BSF 05.0
000E: BCF 03.5
000F: CLRF 12
0010: BTFSC 05.0
0011: INCF 12,F     ;End of inputs check
0012: BSF 03.5      ;Check of inputs from second button
0013: BSF 05.1
0014: BCF 03.5
0015: CLRF 13
0016: BTFSC 05.1
0017: INCF 13,F     ;End of checks for inputs for second button
0018: DECFSZ 12,W   ;If testing against first input flag, skip if false
0019: GOTO 028
001A: DECFSZ 12,W   ;While for first input flag, pass if false
001B: GOTO 028
001C: BSF 03.5      ;Output Sequence for empty display
```

```

001D: CLRF 06
001E: BCF 03.5
001F: MOVF 10,W
0020: MOVWF 06 ;End of Output Sequence for empty display
0021: BSF 03.5 ;Input check for first flag
0022: BSF 05.0
0023: BCF 03.5
0024: CLRF 12
0025: BTFSC 05.0
0026: INCF 12,F ;End of input flag Check
0027: GOTO 01A ;Loop Close
0028: DECFSZ 13,W ;Similar logic as the If+While loop above, except with the second flag
0029: GOTO 038
002A: DECFSZ 13,W
002B: GOTO 038 ;End of if+While loop skip and checks
002C: BSF 03.5 ;Output Sequence for full display
002D: CLRF 06
002E: BCF 03.5
002F: MOVF 11,W
0030: MOVWF 06 ;End of output sequence for full display
0031: BSF 03.5 ;Start of input check for second Flag
0032: BSF 05.1
0033: BCF 03.5
0034: CLRF 13
0035: BTFSC 05.1
0036: INCF 13,F ;End of input check for Second flag
0037: GOTO 02A ;Loop Close
0038: GOTO 00C ;Main Close

```

It is worth noting that it is not necessary in this instance to call the register with the value for LEDNULL and LED8 as it could just be called to display the direct value instead of a location that suits that respective value as sorted previously, however, to keep consistency with higher level

programming practices, this particular coding set features and elected decision to keep the aforementioned practice standard.

Task 4

This task incorporates a couple of the smaller tasks previous explored by requiring a recursive approach for the button checks, seeing as the task doesn't state as to if the program has to increment from the currently displayed value or simply count down or up from the starting point of 0 or F respectively, as such, I have elected to alter it from the shown number, using a common register or value to store the displayed item. The logic for such a task would look similar to the following in a high level pseudo code:

```
#include <main.h>

unsigned int ledNum[] =
{0x40,0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x10,0x08,0x03,0x46,0x21,0x06,0x0e};

int count = 0;
unsigned int holding = 0;
unsigned int holding2 = 0;
void countingUp(void);
void countingDown(void);

void main()
{
    output_b(ledNum[0]);
    while(TRUE)
    {
        holding = input(PIN_A0);
        holding2 = input(PIN_A1);
        if(holding == 1){
            while(holding == 1){
                countingUp();
            }
        }
    }
}
```

```

    if(holding2 == 1){
        while(holding2 == 1){
            countingDown();
        }

    }

}

}

}

void countingUp(void){
    output_B(ledNum[count]);
    holding = input(PIN_A0);
    count++;
    delay_ms(500);
    if(count > 15){count=0;}
}

void countingDown(void){
    output_B(ledNum[count]);
    holding = input(PIN_A0);
    count=count-1;
    delay_ms(500);
    if(count < 0){count=15;}
}

```

This one poses some difficulty in terms of complexity for the assembly code equivalent, as we still would need to have a rather long set of assignments at the start of the code, which would serve to bloat, but is ultimately unavoidable if it is desirous to maintain a standard of assignment regarding constant values. We can start to investigate the conversion of this code by following recursive principles, unlike in the approach of task 1, as such it would be simpler to create two sections similar to the functions defined in the pseudo code, as to avoid unnecessary manual repetition. This would look something like the following:

```

0000: MOVLW 00
0001: MOVWF 0A      ;Initialize register and Working file to "0A"

```

```

0002: GOTO 052      ;Go to index 052
0003: NOP           ;No operation breaker for continuity
0004: MOVLW 24      ;Similar to previous tasks, triggers the oscillator for pulses to be used in
outputs and so on, continues to line 017
0005: MOVWF 04
0006: MOVF 00,W
0007: BTFSC 03.2
0008: GOTO 017
0009: MOVLW 01
000A: MOVWF 0D
000B: CLRF 0C
000C: DECFSZ 0C,F
000D: GOTO 00C
000E: DECFSZ 0D,F
000F: GOTO 00B
0010: MOVLW 4A
0011: MOVWF 0C
0012: DECFSZ 0C,F
0013: GOTO 012
0014: GOTO 015
0015: DECFSZ 00,F
0016: GOTO 009
0017: RETURN        ;end of oscillator initialization
0058: MOVLW 40      ;Start of variable assignment for LED Values
0059: MOVWF 10
005A: MOVLW 79
005B: MOVWF 11
005C: MOVLW 24
005D: MOVWF 12
005E: MOVLW 30
005F: MOVWF 13

```

```

0060: MOVLW 19
0061: MOVWF 14
0062: MOVLW 12
0063: MOVWF 15
0064: MOVLW 02
0065: MOVWF 16
0066: MOVLW 78
0067: MOVWF 17
0068: CLRF 18
0069: MOVLW 10
006A: MOVWF 19
006B: MOVLW 08
006C: MOVWF 1A
006D: MOVLW 03
006E: MOVWF 1B
006F: MOVLW 46
0070: MOVWF 1C
0071: MOVLW 21
0072: MOVWF 1D
0073: MOVLW 06
0074: MOVWF 1E
0075: MOVLW 0E
0076: MOVWF 1F ;End of variable Declarations
0052: MOVF 03,W ;Start of main, initializing a few more flag variables to the main
0053: ANDLW 1F
0054: MOVWF 03
0055: CLRF 20
0056: CLRF 21
0057: CLRF 22 ;End of function initialization and flag declaration
0077: BSF 03.5 ;Start of output sequence for LEDs starting at 0 and waiting
0078: CLRF 06

```

```

0079: BCF 03.5
007A: MOVF 10,W
007B: MOVWF 06 ;End of output Sequence for LEDs
007C: BSF 03.5 ;Flag check against button inputs
007D: BSF 05.0
007E: BCF 03.5
007F: CLRF 21
0080: BTFSC 05.0
0081: INCF 21,F ;End of flag input check with increment if true
0082: BSF 03.5 ;Check for second flag input
0083: BSF 05.1
0084: BCF 03.5
0085: CLRF 22
0086: BTFSC 05.1
0087: INCF 22,F ;End of second flag check with increment if true
0088: DECFSZ 21,W ;If check for first flag
0089: GOTO 08E
008A: DECFSZ 21,W ;While logic following flag
008B: GOTO 08E
008C: GOTO 018 ;Counting up function call with added flag check
008D: GOTO 08A
008E: DECFSZ 22,W ;If for second flag
008F: GOTO 094
0090: DECFSZ 22,W ;While linked to second flag
0091: GOTO 094
0092: GOTO 038 ;Counting down function call with added second flag check
0093: GOTO 090
0094: GOTO 07C ;Loop end
0095: SLEEP ;Shouldn't be triggered but sleep because apparently it is standard to
include
0018: MOVLW 10 ;Start of display for Relevant LED sequence

```

```

0019: ADDWF 20,W
001A: MOVWF 04
001B: MOVF 00,W
001C: MOVWF 23
001D: BSF 03.5
001E: CLRF 06
001F: BCF 03.5
0020: MOVF 23,W
0021: MOVWF 06 ;End of display for relevant LED sequence
0022: BSF 03.5 ;Input Check for first flag
0023: BSF 05.0
0024: BCF 03.5
0025: CLRF 21
0026: BTFSC 05.0
0027: INCF 21,F ;End of input Check for first flag
0028: INCF 20,F ;Increment Counting variable
0029: MOVLW 02 ;Time waste delay
002A: MOVWF 23
002B: MOVLW FA
002C: MOVWF 24
002D: CALL 004
002E: DECFSZ 23,F
002F: GOTO 02B ;End of time waste delay
0030: MOVF 20,W ;If check for if count exceeded displayable value
0031: SUBLW 0F
0032: BTFSC 03.0
0033: GOTO 037
0034: MOVF 20,F
0035: BTFSS 03.2
0036: MOVLW 00 ;End of if Check
0037: GOTO 08D (RETURN) ;End of function

```

```

0038: MOVLW 10      ;Start of function for decrementing, start of relevant display sequence for
count (Number) of LED array values

0039: ADDWF 20,W

003A: MOVWF 04

003B: MOVF 00,W

003C: MOVWF 23

003D: BSF 03.5

003E: CLRF 06

003F: BCF 03.5

0040: MOVF 23,W

0041: MOVWF 06      ;End of Output Sequence

0042: BSF 03.5      ;Stat of input check

0043: BSF 05.0

0044: BCF 03.5
0045: CLRF 21
0046: BTFSC 05.0

0047: INCF 21,F      ;End of input check with possible increment of flag

0048: MOVLW 01      ;Decrement method for count

0049: SUBWF 20,F      ;Decrement method for count

004A: MOVLW 02      ;Time waste delay method

004B: MOVWF 23

004C: MOVLW FA

004D: MOVWF 24

004E: CALL 004

004F: DECFSZ 23,F

0050: GOTO 04C      ;Time waste delay end

0051: MOVF 20,F      ;If Process

0052: BTFSS 03.2

0053: GOTO 056

0054: MOVLW 0F

0055: MOVWF 20

```

0056: GOTO 093 / RETURN ;end of if process and relevant return

Here we can see all of the hallmarks of the C style pseudo transferred over into assembly, following the same logical process we can reach a similar function for the segmented display that initially shows a zero and then counts up and down from a constant character, there are some inflexibilities in the code but, for the vast majority of cases, this was unimpactful on the safe compiling and running of the task.

Task 5

Task five employs a new sort of task to be attempted, this time using the piezo buzzer to output a continuous tone, the specification required that a tone be emitted when one of the buttons is pressed, It was understood that this implied that it would activate when either of two buttons were pressed, for the duration of their press. The C-Pseudo that follows is the logical approach that would have to be taken.

```
#include <main.h>
```

```
unsigned int ledNum[] =  
{0x40,0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x10,0x08,0x03,0x46,0x21,0x06,0x0e};
```

```
int count = 0;
```

```
unsigned int holding = 0;
```

```
unsigned int holding2 = 0;
```

```
void countingUp(void);
```

```
void countingDown(void);
```

```
void main()
```

```
{
```

```
    output_b(ledNum[0]);
```

```
    while(TRUE)
```

```
    {
```

```
        holding = input(PIN_A0);
```

```
        holding2 = input(PIN_A1);
```

```
        if(holding == 1 || holding2 == 1){
```

```
            while(holding == 1 || holding2 == 1){
```

```
                holding = input(PIN_A0);
```

```
                holding2 = input(PIN_A1);
```

```
                output_high(PIN_B7);
```

```

    }
    output_low(PIN_B7);
  }
}
}

```

If it were to be implemented in a similar manner to the pseudo, it would look like the following.

```

0000: MOVLW 00
0001: MOVWF 0A ;Initialize register and Working file to "0A"
0002: GOTO 004 ;Go to index 052
0003: NOP ;No operation breaker for continuity
000A: MOVLW 40 ;Start of LED value delarations
000B: MOVWF 10
000C: MOVLW 79
000D: MOVWF 11
000E: MOVLW 24
000F: MOVWF 12
0010: MOVLW 30
0011: MOVWF 13
0012: MOVLW 19
0013: MOVWF 14
0014: MOVLW 12
0015: MOVWF 15
0016: MOVLW 02
0017: MOVWF 16
0018: MOVLW 78
0019: MOVWF 17
001A: CLRF 18
001B: MOVLW 10
001C: MOVWF 19
001D: MOVLW 08
001E: MOVWF 1A

```

```

001F: MOVLW 03
0020: MOVWF 1B
0021: MOVLW 46
0022: MOVWF 1C
0023: MOVLW 21
0024: MOVWF 1D
0025: MOVLW 06
0026: MOVWF 1E
0027: MOVLW 0E
0028: MOVWF 1F ;End of LED declarations (Mostly unused Relic code)
0004: MOVF 03,W ;Other variable declarations
0005: ANDLW 1F
0006: MOVWF 03
0007: CLRF 20
0008: CLRF 21
0009: CLRF 22 ;End of other declarations(Also Relic Code, but my assembly references lines
so I don't want to break the logic
0029: BSF 03.5 ;Output 0 to the display
002A: CLRF 06
002B: BCF 03.5
002C: MOVF 10,W
002D: MOVWF 06 ;End of output sequence
002E: BSF 03.5 ;Check for button press with flag 1
002F: BSF 05.0
0030: BCF 03.5
0031: CLRF 21
0032: BTFSC 05.0

```

```

0033: INCF 21,F      ;End of check with increment if true
0034: BSF  03.5      ;Check for button press with flag 2
0035: BSF  05.1
0036: BCF  03.5
0037: CLRF 22
0038: BTFSC 05.1     ;End of check with increment if true
0039: INCF 22,F      ;If with an or conditional for either value true
003A: DECFSZ 21,W
003B: GOTO 03D
003C: GOTO 03F
003D: DECFSZ 22,W
003E: GOTO 05B      ;End of if check
003F: DECFSZ 21,W    ;While with either being true
0040: GOTO 042
0041: GOTO 044
0042: DECFSZ 22,W
0043: GOTO 056      ;End of While Check
0044: BSF  03.5      ;Check inputs for flag refresh flag 1
0045: BSF  05.0
0046: BCF  03.5
0047: CLRF 21
0048: BTFSC 05.0
0049: INCF 21,F      ;Check inputs for flag 1 end
004A: BSF  03.5      ;Check inputs for flag 2 refresh
004B: BSF  05.1
004C: BCF  03.5

```

```

004D: CLRF 22
004E: BTFSC 05.1
004F: INCF 22,F ;End of check refresh for flag 2
0050: MOVLW 00 ;Output high for the Piezo Buzzer
0051: BSF 03.5
0052: MOVWF 06
0053: BCF 03.5
0054: BSF 06.7
0055: GOTO 03F ;End of Output Sequence, with loop close
0056: MOVLW 00 ;Output low in the Peizo when not in the while loop
0057: BSF 03.5
0058: MOVWF 06
0059: BCF 03.5
005A: BCF 06.7 ;End of output segment
005B: GOTO 02E ;Return to start of Main

```

Bibliography

- [1] Gettysburg College, "1.3: Von Neumann and Harvard Architectures," *Libretexts Engineering*, Mar. 22, 2021.
- [2] Administrator, "RISC vs CISC," *Electronics Hub*, Sep. 30, 2015.
- [3] D. Kerr-Munslow, "Advantages and Pitfalls of Moving from an 8 bit System to 32 bit Architectures.," Aug. 2019.
- [4] Multiple Authors, "PIC16F84A Data Sheet," 2001.
- [5] 8 Bit Microcontrollers, "Master Clear Pin External Reset," *Microchip Technology Incorporated*, 2021.

[1]–[5]

Jacob Micros Assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

22/100

PAGE 1



Comment 1

Who is we? You should be writing in the third person voice

PAGE 2

PAGE 3



Comment 2

I wasn't expecting it to be.



Comment 3

addresses would have their own bus. So these systems would have a single data bus connecting the controller to a shared memory space, IO, any other peripherals. The address bus would connect to all of these systems, driven by the controller dictating the location of the read/write operation. The control bus would connect to all these systems too, also driven by the controller, this would dictate the kind of operation i.e. read or write, and synchronise data transport.



Comment 4

Throttling is not really a issue either. The issue I think you may be trying to get to is the fact that because there is only a single bus carrying both data and instructions, this has less effective bandwidth because it has to carry both



Comment 5

data collision is never a thing in systems like this. The peripherals only "speak when they are spoken to". The address and control busses prevent this from every happening.

**Comment 6**

Captions should:

1. include a description of the thing they belong to,
2. be beneath the thing they belong to.

**Comment 7**

Word choice error?

**Comment 8**

This resistor is simply there to pull up the MCLR pin to the supply - this has the effect of stopping the PIC from resetting. The reason we use a resistor instead of just connecting the pin directly to supply is to limit the current, protecting the PIC from excess current flow.

**Comment 9**

No, they are load capacitors for the crystal oscillator.

**Comment 10**

No, as the sounder is a piezo device - these are mainly capacitive and so need a resistor to discharge them. removing this resistor would make the sound quieter.

**Comment 11**

the transistor switches current to the piezo sounder, while the resistor limits the current available to the transistor from the PIC - protecting the IO pin of the PIC.

**Comment 12**

incomplete sentence...

**Comment 13**

the resistors are current limiting resistors - protecting both the PIC and the LEDs in the 7-segment display

**Comment 14**

You have not explained where these values came from or what they represent. Why these specific numbers?



Comment 15

This is not what you have coded in assembly below. This algorithm uses nice tight code and stores the values in a continuous block of memory, recalling them one at a time based on a for loop variable. Your assembly code produces the same effect but uses a different mechanism to do so. It manually pushes hard coded values to the LEDs. These are not the same.

PAGE 6

PAGE 7



Comment 16

Why haven't you used register and variable names? This makes your code far more difficult to read than it needs to be - for both you and me!



Comment 17

I'm not sure why you are repeatedly switching memory banks and setting PortB to outputs every time you want to update its pins. The registers remember their previous settings (hence the name memory!) This could have been achieved in a far neater way by using the INDR and FSR registers.

PAGE 8

PAGE 9



Comment 18

I feel you may have purposely made this code inefficient as there are a large number of redundant lines of code. It looks as though you have coded this in C and copied the output of the compiler.



Comment 19

"value 3.5" is actually one of the bank select bits that allows you to switch between banks to set the TRIS register (which controls the direction of the pins - input or output)



Comment 20

register 6 corresponds to Port B's pins, not somewhere you would store a variable temporarily.

PAGE 10

PAGE 11

PAGE 12



Comment 21

That is not what clearing bit 3.5 does.



Comment 22

These three lines of code are the type of thing a cheap non-optimised compiler would insert into the assembly when "Output_B(LED_x)" is called. This code actually changes the bank select bits then ensures Port B is an output and resets the bank select bits to their original

value before actually writing the value to the PortB register. No human would write code like this - they would know the pin is configured as an output and remains an output throughout the entire program, and hence they wouldn't repeatedly write to the TRISB register (Bank 01, file 06).

PAGE 34



Comment 23

Why are your line numbers out of order?



Comment 24

These inflexibilities are exactly the thing you were supposed to avoid with tight, reusable code. This is neither tight or reusable, nor does it follow exactly the "C style" pseudo code you presented.

PAGE 35



Comment 25

Why are you writing to the PCLATH register? (Program counter latch high byte)



Comment 26

Where are lines 0004 - 0009?

PAGE 36



Comment 27

Why is line 0004 directly after line 0028?

PAGE 37

PAGE 38

GRADING FORM: SMU GENERIC_SMOSEY

JACOB BROWN

22

ENGAGEMENT WITH LITERATURE

including reading, referencing, academic conventions and academic honesty



You have clearly not done much reading around the circuits and systems and have mis-stated the purpose of most of the components. - Please see my comments in your text

7

KNOWLEDGE & UNDERSTANDING

Knowledge of the basic underlying concepts and principles of a subject.



You have not sufficiently explained the board, its components, or its operation.

6

COGNITIVE & INTELLECTUAL

Evaluate underlying concepts and principles of a subject and interpret qualitative and quantitative data in order to develop lines of argument and make sound judgements.



You have not written the assembly included in this report. I strongly believe that you have used a compiler to generate it for you. This is dishonest and is only cheating yourself.

0

PRACTICAL APPLICATION SKILLS

Different approaches to solving problems in particular contexts.



While the pseudo code you have provided would solve the problems, the assembly implementations of these presented do not match algorithmically.

0

TRANSFERABLE SKILLS

Communicate the results of their study/work accurately and reliably, and with structured and coherent arguments; the qualities needed for employment requiring the exercise of some personal responsibility within a structured and managed environment.



A fairly well written report although including the code as screen shots of the proteus editor would have been preferable to the way you have included your code.

9