MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 3

Lab Guide

Spring 2016

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

The goal of this lab is to introduce more advanced code structures and assembly instructions. We will also begin looking at the transition from high-level languages to low-level assembly in more detail. From the next lab and onwards, we will increasingly use high-level code in place of low-level assembly. This lab will focus on introducing the different types of addressing modes, and how they relate to high-level constructs.

Addressing Modes Introduction

Almost every instruction executed by a microcontroller or microprocessor requires the manipulation of data. We have already seen several instructions and gained an intuitive understanding of some basic addressing modes. Let's examine these in more detail first.

Inherent Addressing (INH)

The simplest possible addressing mode is one where there is no data **explicitly** addressed by the instruction; the choice of data source (i.e., which registers to use) is said to be **inherent** to the instruction. They are labelled 'INH' in the instruction set. The following are a variety of useful instructions, some of which we have seen, and some of which are new. When reviewing the instructions that follow, recall the primary register set of the 68HC12: (i) two 8-bit accumulators, A and B, which can be combined into one 16-bit accumulator, D, and (ii) two 16-bit index/general-purpose registers, X and Y.

NOP	–	No operation (CPU waits one cycle)
INCA/INCB	–	Increment accumulator A / B
DECA/DECB	–	Decrement accumulator A / B
TBA	–	Transfer contents of B to A
CLI	–	Clear interrupt mask (enables **interrupts**)
IDIV	–	16x16 bit integer divide $\left(\frac{D}{X} \rightarrow D\right)$
EMUL	–	16x16 bit unsigned multiply $(D \times Y \rightarrow [Y:D])$
RTS	–	Return from a subroutine
RTI	–	Return from an interrupt

As you can see, there is no choice of data source available to the instructions above. For RTS, we will use it in combination with a CALL instruction to create **subroutines**. For RTI, we will use it when dealing with **interrupts** – asynchronous events that a microcontroller must execute special code to handle immediately upon generation. The remaining instructions are binary arithmetic instructions that involve only a fixed set of registers. Thus, inherent instructions can affect data in registers or memory, but there is no choice in sources for the programmer.

Immediate Addressing (IMM)

An instruction which uses immediate addressing, as the name suggests, stores a constant value in code memory immediately following the instruction op-code. It is most useful for loading simple constants, initial values, or other non-variable numbers into a register that represents a variable. Aside from the direct register manipulation instructions ('LDAA', etc.), most arithmetic and binary logic operations (ANDA, ORAA, etc.) support immediate values as an operand. Note that for the immediate values are typically limited to the bit width of the **destination** register (i.e., 8-bit immediate value for LDAA, 16-bit for LDD).

```
LDAA #12      –   Loads a constant decimal 12 into A
ANDB #$FF     –   Logical AND of B with constant hex 0xFF
ADDD #500     –   Adds a constant decimal 500 to D(= [A:B])
```

Recall earlier that the registers X and Y serve a dual purpose. Aside from giving us two general-purpose 16-bit registers, they also serve as **index** registers. We will often use X and Y as part of more complex addressing methods, for example to hold a **base address**. We can load X and Y with immediate values in the same manner as before:

```
LDX   #$10A   –   Loads a constant hex 0x010A into index X
```

We can also make use of assembler **labels** to refer to a specific memory location by name. In the following code the label 'TONE' refers to a specific memory location:

```
TONE:     FCB 0xFF, 0x0A, ... , 0x00
          ...
          LDX #TONE
```

Specifically, the label 'TONE' is a line label. The **assembler directive** 'FCB' which follows stands for **Form Constant Byte**, meaning a constant value that is stored in memory at the location where the FCB directive is written. We can use this directive to easily store **lists** of constant values in code memory. When the LDX command is executed, it loads the memory **address** (not the value itself) corresponding to the starting value of this list into X. In essence, X becomes a pointer to this constant data – we will use this to implement arrays.

Other Assembler Directives

The previous code would not be especially useful without additional assembler directives. We have already seen labels used to refer to code addresses as well as constant data:

```
LOOP:     LDAA #01
          ...
          JMP LOOP
```

In the above code, the JMP instruction uses the 'LOOP:' as a location in code memory to transfer program execution to. However, the question is where exactly in memory is any of the above located? For a high-level language, this would be taken care of by the compiler. Even for the 68HC12 Assembler, there are default locations. However, we can take specific control by using the 'ORG' or ORiGin directive. This tells the assembler exactly what location in code memory the next instruction or data should be stored at. For example:

```
ORG         $0180
TONE:       DB 0xFF, 0x0A, ... , 0x00
            ...
            LDX #TONE
```

The use of #TONE in this example is equivalent to writing #0x0180, as we specify (through ORG $0180) that the next instruction or data (TONE in this case) will begin at hex location $[0180]_{16}$ in memory. We will use this directive to control the placement of our data, and also subroutines and interrupts later. Note the use of DB ('define byte'), which performs the same function as FCB.

Register Addressing

Other microcontrollers (not the 68HC12) support an additional form of addressing called **register addressing** where there are **two** operands, **at least** one of which is a register or accumulator. This more general form of addressing is often found on microcontrollers with a greater number of general purpose registers. In fact, in these types of microcontrollers, an instruction can actually contain a mix of addressing modes:

```
    MOV   B, #12
    MOV   R0, #0xFF
    MOV   R5, #0xFF
    ADD   A, #0x01
    ADD   A, B
    SUBB A, #0x06
```

Notice that the instructions similar to the above do not appear in the 68HC12 instruction set for the most part. Given its relatively small set of registers, most destination registers are typically **inherent**. Additional instructions allow the movement of data between registers (such as TBA, transfer B to A, for example). Again, this only works because of the small register set.

Direct Addressing (DIR) and Extended Addressing (EXT)

One major language feature that we have been missing up to this time is the ability to access memory. We actually have made extensive use of this feature in Lab 2. For this purpose, the 68HC12 defines two addressing modes: direct and extended.

For direct addressing, the 68HC12 can load, store, and work with data in the **address range** of 0x0000 to 0x00FF. In other words, we are limited to a range of 8-bits in address space, with the 8 MSBs of each address assumed to be zero. Instructions using this mode are smaller in code memory and execute slightly faster:

LDAA $A0	–	Load A with data located at 0x00A0
LDD $00FE	–	Load D with data located at 0x00FE:0x00FF
ADDD $18	–	Add to D the data located at 0x0018:0x0019

To access the remainder of the memory space (the full 16 bits), we use extended addressing mode. Commands for this mode look identical in format, except for the fact that they accept a 16-bit value for the address:

LDAA $A1A0	–	Load A with data located at 0xA1A0
LDD 1000	–	Load D with data located at 0x03E8:0x03E9
ADDD $1800	–	Add to D the data located at 0x1800:0x1801
STAA display	–	Store A at address pointed to by the code label 'display'

Notice, however, that the address used in each of the above instructions is fixed at build time; we cannot modify it while the program is running. This is suitable for creating constant, global, or otherwise static variables (as we did in Lab 2). For **local** variables (such as those created in a subroutine call) or other **dynamic** variables, we need to use something like the **stack**. We also need some additional addressing modes to properly address more complex structures, such as arrays.

Indexed Addressing Mode (IDX)

The 68HC12 implements a family of different methods for calculating an **effective** address from a register, and then loading data from the memory pointed to by that effective address. The simplest of these modes is the sub-family of indexed addressing modes. As the name implies, the indexed mode typically uses the index registers (X and Y) in combination with either a fixed or variable **offset**. If we examine the **constant-offset** indexed instructions, one can see that the constant fixed offset can be 5, 9, or 16 bits in length. This means that we can address a range of either −16/+15, −256/+255, or −32768/+32767, depending on the size of the constant (5, 9, and 16-bits respectively). This means that indexed addressing can potentially address the entire 16-bit memory range given **any** base index in X or Y. Some examples of simple indexing are as follows:

LDAA –5, X – Take the value in X, subtract 5, and interpret as an address; load the data at this address into A

LDAB $200,Y – Load data at address (Value of Y + 0x0200) into B

STAA 9, SP – Take the value in **SP** (Stack Pointer), add 9, interpret as an address; store the data in A at this effective address

Note that the last of these examples (STAA), and its equivalent LDAA counterpart, are **crucially** important. They allow us to address memory locations **relative** to the current stack pointer value. This means we can use the stack to implement and access **temporary, dynamically allocated** local variables. These are mandatory if we declare any variables in later **functions** that we implement at the high level.

Advanced Indexing

While we now have the ability to create local variables, we still cannot easily address an **array**. To accomplish this, we can use a **combination** addressing mode which combines the indexed mode above with a partially-inherent register operand. The result is very similar to the **register addressing** mentioned earlier:

LDAA B, X – Take the value in X, add the value in B, and interpret as an address; load the data at this address into A

Notice that this allows us to create an array as follows: (i) load the base address into X, (ii) load the desired offset into an accumulator, and then (iii) use the above instruction. If, for example, we want to access an array variable, such as "a[i]", we would load the address for a[0] into X or Y, the offset 'i' into A or B, and then execute an instruction like LDAA B,X above.

Since we often move through arrays and similar variables **sequentially or linearly**, the 68HC12 provides one last advanced indexing mode that is convenient for this function. Let's say that we have a 'for' loop or similar structure that loops through an array, as in 'a[i]' or similar. We can use a **pre or post-increment index mode** to automatically load a value from a base address **and** automatically increment the address. For instance:

LDAA 1, X+ – Take the value in X, interpret as an address, and load the data at this address into A. **Then**, after this is complete, increment X by one.
STAB 2, –Y – Take the value in Y, subtract two **first**, and **then** interpret as an address, storing the data in B at this new address.

Notice that we have control over both how much we increment by and whether our address is incremented **before or after** the actual memory access. This is very useful for implementing loop functions.

Branching and Flow Control

In order to write more advanced programs, we will also need to be able to control the flow of execution. We touched upon this briefly in Lab 2, as well. In general, if we want constructs like loops, if/then/else, or other decision-making code, we need a means of controlling which instruction executes next. This is achieved through branching instructions. We have already seen the standard JMP:

```
LOOP:       LDAA #01
            ...
            JMP LOOP
```

This stands for absolute JuMP, meaning upon executing, the program will continue at exactly the location in code memory specified by its operand. This requires that we store the complete address of the **jump vector** (target location) as part of the instruction. The above instruction uses **extended addressing** to store the exact, complete 16-bit address of the desired jump location (the jump vector). All other variants of the JUMP instruction support this form of addressing, meaning we can potentially jump (at any time we want) to code located anywhere in memory.

However, this can be inefficient; the JUMP class of instructions uses more code memory and they tend to be slower. While the execution time of an individual instruction is often minute, branching instructions are often executed repeatedly and rapidly (such as in a loop), thus even a small change in the time they take can have a big impact on overall program execution speed. Thus, the 68HC12 (and most other microcontrollers) offer a second form of flow control instruction called the **branch** class. We have seen these instructions in Lab 2 as well. These instructions use a special addressing mode called relative addressing.

Relative Addressing

When an instruction using relative addressing is executed, the target address is calculated at **run-time**. The instruction itself stores only an **offset** which is added to the current value of the program counter (PC). For example, consider the code on the following page.

```
ORG $0100
MAIN:       LDAA #10
            LDAB #20
            BEQ MAIN
```

The program would begin by reading and executing the first two LDAA/LDAB instructions. Each of these instructions requires two bytes of code memory (one for instruction opcode, one for immediate 8-bit valye). Thus, the BEQ instruction begins at 0x0105. The BEQ instruction itself takes two bytes of memory (one for the **opcode**, and one for the offset). That means that after being loaded and starting to execute, the program counter would be advanced to 0x0107, the address of what would be the next instruction (the one after BEQ). However, the BEQ instruction may (or may not) modify the PC. If it does, then the relative offset is added directly to the PC. We store this offset as a signed 2's complement number, meaning that when added to the PC, the offset allows us to obtain a new range for the PC of PC–128 to PC+127. For the above example, the relative offset would actually be stored (in 2's complement) as $-7_{10} = [1111\ 1001]_{2's\ compl.} = 0xF9$. We can see that if we add this value back to the PC, we get the address of MAIN (0x0100):

```
PC (value after loading JZ MAIN instruction)    = 0x0107     = 0000 0001   0000 0111
Offset to get PC = 0x0100                       = 0x    F9   =             1111 1001
```

We apply **sign extension** to the offset, meaning we simply duplicate the MSB and then perform the addition:

```
PC (value after loading JZ MAIN instruction) =  0x0104 =     0000 0001    0000 0111
Offset to get PC = 0x0100                    =  0xFFF9 =  +  1111 1111    1111 1001
Result = 0x0100                                        =     0000 0001    0000 0000
```

Note that we ignore the carry out from the addition. The result is the correct jump vector of 0x0100. Luckily, most assemblers carry out the translation from labels (such as 'MAIN:' in the above code) to offsets for us. However, you must be aware of the limited amount of code that you can 'branch over' in this way – no more than ~128 bytes of code in either direction. More modern microcontrollers have full bit-width offsets.

Conditional Branching

The 68HC12 and other microcontrollers implement a full set of instructions which will branch if certain conditions are met:

BLE	– Branch if last result was less than or equal to zero
BLO	– Branch if last result was 'lower' (i.e., **carry out in CCR** was set, unsigned)
BLS	– Branch if lower or the same

BLT	– Branch if less than
BMI	– Branch is minus
BNE	– Branch if result is not equal to zero
BPL	– Branch if plus
BRA	– Branch always
BRCLR	– Checks operand versus a **bit mask**, branches if all masked bits clear
BRSET	– Checks operand versus a **bit mask**, branches if all masked bits set
BSR	– Branch to a **subroutine**
BVC	– Branch if overflow bit clear
BVS	– Branch if overflow bit set

Notice that the branch instructions implement significant functionality; some (such as BRCLR/BRSET) again combine multiple functions. Lab 2 would be made shorter through the use of BRSET, for example. The majority of the above branches are triggered from conditions tracked by the CCR (again from Lab 2). Thus, we can branch based on the result of most previous instructions.

The 68HC12 also implements **compare** instructions (CMPA/CMPB) that allow us to easily set up condition checking for **if/else** and similar structures **without** converting every condition into an addition or subtraction. For example, we could translate code as follows: "(A > 9)" →CMPA #9. This can be followed by an appropriate branch instruction to implement the if/else branching we saw in Lab 2.

Lab Procedure – Part 1: Program Flow

There are two primary goals to this lab. First, we will introduce the addressing modes by writing a program that reads an array of constant values. In the second part, we will build this into a more complex program (using our branch instructions) which plays a series of tones from a simple circuit that we build.

Step 1. To begin, if you have not done so already, connect the 68HC12 board's USB and power cable and open CodeWarrior.

Step 2. To begin, we will be writing a simple program that implements the following high-level C-like program:

```
int tone[10] = { 0,219,0,219,0,219,0,246,0,219};
int length[10] = { 20,10,10,10,10,10,10,10,10,10};
void main()
{
   for (int i=0; i<10; i++)
   {
      _PortB = tone[i];
      _PortB = length[i];
   }
}
```

For now, all this program does is sequentially read the values from the arrays tone[ ] and length[ ], and output the result to Port B. You can load the Lab3_1.mcp project for a skeleton code listing. The skeleton will take care of setting up Port B, as well as creating the array of constant values. You must write code to implement main( ) in the highlighted area. Some suggestions and hints:

1. You may notice that the constant arrays (tone and length) are actually stored in memory **after** your program. If your program grows too long, it will intersect with this stored data. In such cases, the assembler will generate an error message. To resolve this issue, simply increase the $4050 in the last ORG directive until it is slightly past the end of your code.

2. You will have to debug / **step through** your program using F10 / Step Over while testing to view **both values** that are displayed on Port B, as only one can be written at a time.

3. Examine your new addressing modes to determine the best way to access the data stored at TONE and LENGTH, which represent the arrays tone[ ] and length[ ]. You will also need a variable (accumulator) to store the counter 'i,' and a branch instruction or two to implement the loop. To finish your program, an endless loop is already included.

Once your code is complete, verify that it works by **stepping** through and checking that the array values correctly appear on Port B. Record your finished program as **Question 1** in the deliverable.

Step 3. Modify your code to implement the following program. We are now going to check the tone[ ] value. If it is zero, this indicates that no sound will be output by our final program, so we will use a delay routine. Otherwise, we will output a timed tone.

```
int tone[10] = { 0,219,0,219,0,219,0,246,0,219};
int length[10] = { 20,10,10,10,10,10,10,10,10,10};
void main()
{
   for (int i=0; i<10; i++)
   {
      _PortB = tone[i];
      if (tone [i] == 0)
      {
          _PortB = 0xFF;
          // Tone output code will go here later
      }
      else
      {
          _ PortB = 0x00;
          // No output delay will go here later
      }
   }
}
```

Step through and test the above code to ensure that your if/else statement executes correctly. Record your code for the if/else statement as **Question 2** in the deliverable.

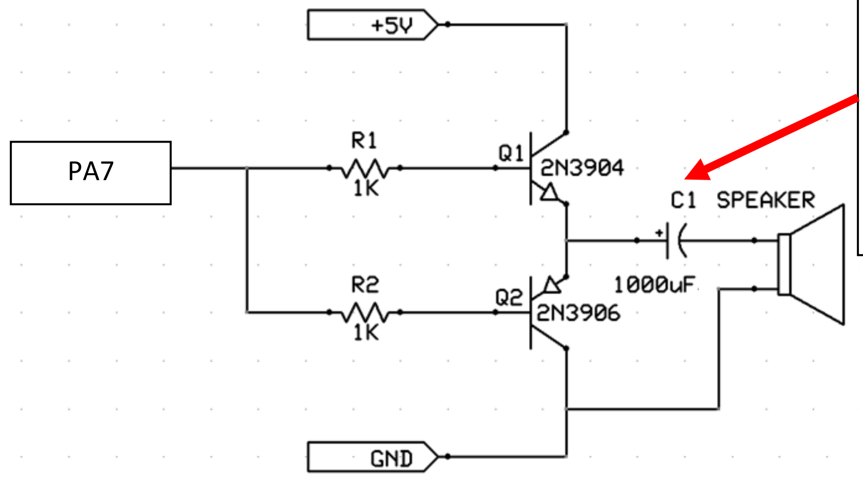
Lab – Part 2: Tone Generation

The second half of the lab requires some basic circuit building. We will be using a form of amplification called **Class D** amplification. While typically used for either very efficient or very high-power outputs, we will be implementing only a simplified version. The basic concept behind Class D amplification is the use of PWM, or pulse-width modulation.

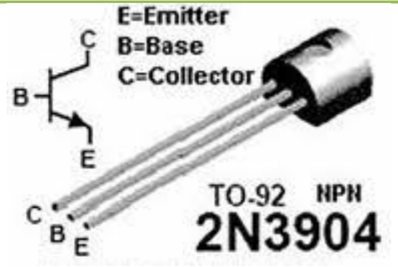
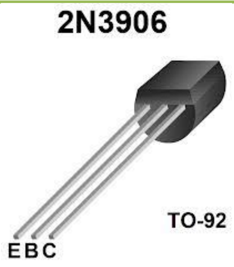
In order to produce sound, we will make use of our knowledge of the frequency domain from MIE346. Consider a square wave; it has a particular period of repetition, T. If we take $1/T$, we get the fundamental frequency of this square-wave. However, from MIE346, one can recall that a square wave actually contains an infinite sum of increasing frequencies, starting with its fundamental. If we want only a single tone (i.e., one sine wave), all we need to do is create a square wave of the desired frequency and filter out all of the frequencies higher than the fundamental. To do so in a Class D amplifier, we would normally use a combination of L/C filters. However, since we are producing only relatively simple tones, we will actually make use of the speaker's inductance to automatically filter the higher frequencies. The resulting circuit is very simple to construct, but allows us to create tones without full D/A conversion and signal processing.

As a note, the focus of this lab is not so much about the circuit, but if you are interested in Class D audio (of which this is a very rudimentary example) or have any questions about the process, please feel free to ask the instructor. We will revisit input and output in later labs, where we will learn how to properly drive BJTs, MOSFETs, and other devices correctly using logic outputs. We will also learn about other forms of Digital to Analog (and Analog to Digital) conversion.

Step 4. For now, we will continue with building the circuit. To begin, identify and make sure that you have all the necessary parts.

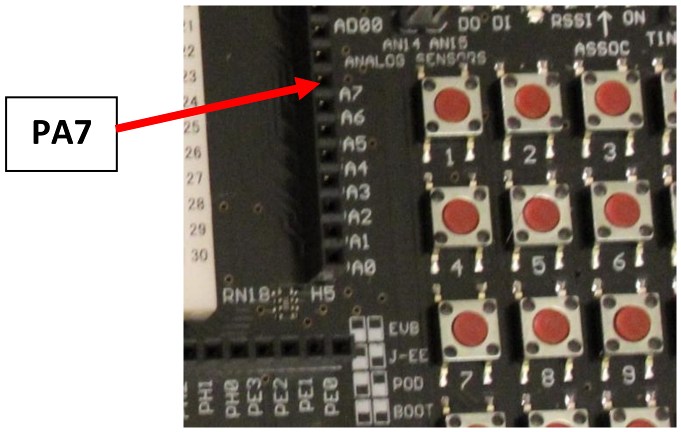


NOTE THAT THIS CAPACITOR IS POLARIZED; if you are unfamiliar with how to connect it correctly, ask a TA! If connected backwards, it can explode and/or damage your board.

1 $K\Omega$ resistor x2	 E=Emitter B=Base C=Collector C B E TO-92 NPN 2N3904 C B E 2N4401 or 2N3904 BJT (NPN)	 2N4403 or 2N3906 BJT (PNP)
1000μF capacitor **(polarized)**	**8Ω, 0.5W speaker**	Wire

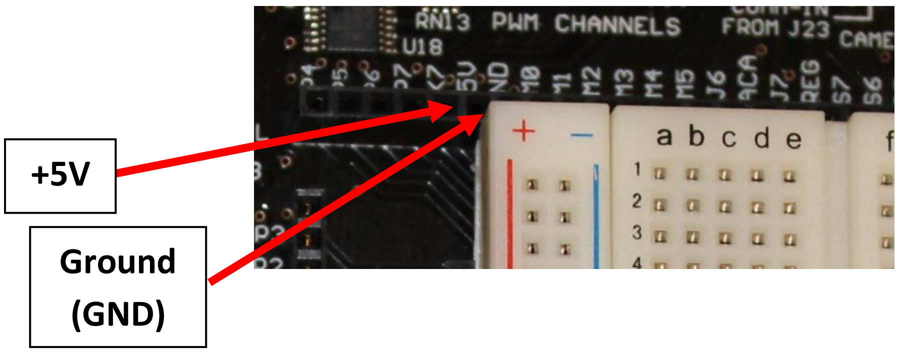
Begin constructing the circuit. Use the proto-board built onto the 68HC12 EVB, and **do not** use the external power supplies. We will instead be drawing power from the EVB directly. **Do not make connections to the 68HC12 board while it is powered on, and do not power up the circuit without having the TA check your wiring first!**

The connection to Port A (PA7 on the diagram) uses the connector located near the keypad:



You can actually connect this input to any of PA0…PA7; the entirety of Port A is switched in the same way by the code that follows.

The connection to the +5V and Ground power leads is located at the top of the protoboard. They are clearly labelled '+5V' and 'GND':



Lastly, load the project Lab3_3.mcp, download the code, and when ready run it. You should hear a **steady** tone from the speaker. Use the oscilloscope at your station to answer **Question 3 and Question 4.** You should connect the scope input probe to PA7 to view the output square wave.

Step 5. We are now ready to try creating a chain of tones to produce music. Close the previous project and load Lab3_4_music.mcp. Be sure to leave the circuit from before connected to the board.

To complete the lab, run the program and answer **Question 5 and Question 6** on the hand-in. It is also suggested that you try modifying the program to get a better feel for what each variable does. For example, try changing the delay scaling to see how it affects the sounds produced. This code should demonstrate to you why we will later need **dedicated** hardware peripherals for true PWM – the code as written monopolizes the CPU to perform relatively simple audio output, and yet is still very sensitive to timing issues.

***** End of Laboratory 3 *****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 3

Deliverable

Spring 2016

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

1. Record your array reading program here:

2. Record your code for the if/else statement here:

3. What frequency of square wave is created at the output? Is the duty cycle of the square wave 50% (i.e., is the amount of time spent 'off' equal to the amount spent 'on')? If not, why might this be the case (refer specifically to the code)?

4. Change the code from Question 3 to produce a symmetrical square wave (or as close to one as you can). Record your code changes here. Hint: Should be a short change.

5. Change the 'Inner Frequency Scaling' value from #14 to a different, yet still small, value (not zero). What effect does this have on the frequencies **and length** of each tone? Why does this happen?

6. Re-write the program (beginning at ‘start:’ and ending at ‘endprog:’ as high-level pseudo-code, high-level C-like code, or a flowchart (if you prefer). Try to capture the functionality more than the changes to individual registers, etc. It would be best to convert any loops or flow control elements to the appropriate high-level structures, such as FOR-loops, DO/WHILE, IF/ELSE, etc.

Fun (Easy): Identify the song. **(Hard):** Change it to any other 8-bit music (I recommend a Megaman boss theme). If you do, please take a video!

This lab hand-in is due at the start of the subsequent lab section.