Tiffany Q. Liu
March 28, 2011
CSC 270
Lab #8

# Lab #8: Introduction to the 6811 Microprocessor Kit

## Introduction

The purpose of this lab was to become familiar with the 6811 microprocessor kit, by examining what is under the hood of the processor cartridge and by writing, inputting, and running programs on the 6811.

## Materials

Figure 1. 6811 Microprocessor Kit (Taken from D.Thiebaut).

## Engineering/Exploration

In order to see what the 6811 processor cartridge in the kit (Fig. 2) was actually composed of, we removed the cartridge and opened it up using a Phillips screwdriver. We were able to identify the following parts on the circuit board (Fig. 3):

- 6811 processor
- oscillator (crystal)
- memory (ROM – the operating system)
- surface mount logic gates
- buses
- transistors
- capacitor
- resistors
- static RAM (D43256C)

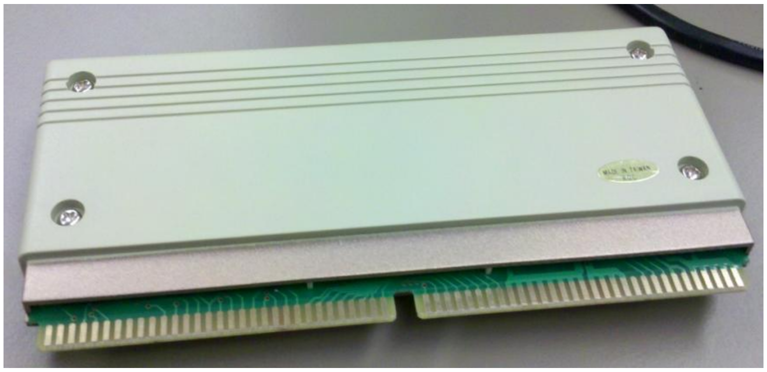

Figure 2. 6811 Processor Cartridge.


Figure 3. 6811 Circuit Board.

**Inputting First Program in the Kit**

After putting the cartridge back together, we tried inputting the following prewritten and assembled program in the kit:

```
ADDR BYTES
0000 CC 00 18
0003 BD C0 1B
0006 BD C0 27
0009 CC 00 27
000C BD C0 1B
000F BD C0 27
0012 BD C0 27
0015 BD C0 00
0018 50 72 6F 67 72 61 6D 6D 69 6E 67 20 69 73
0026 00
0027 0D
0028 45 61 73 79 20 61 6E 64 20 46 75 6E 21 21 21
0037 00
```

When entering the program, we used the following table, which contained descriptions of each key on the kit, as reference:

| Key | Name | Description |
|---|---|---|
| | RESET | Takes you out of trouble! Press this key whenever you want to return to the "main prompt" showing CPU UP. |
| | RPO | **R**eturn to **P**revious **O**peration. |
| - | Return | Press the RETURN key to exit a function and save data contained in the CPU registers. |
| + | List | Use List in break mode to list the breakpoints. |
| RPO | Help | Display a help message. Keep pressing it for more messages. |
| D | RS-232 | Press this key to transfer control from the kit's keyboard to the serial port. |
| E | Baud | Press this key to set the transfer rate to the outside serial port. |
| F | Insert | Use Insert to change the contents of the memory. |
| A | Load | Use Load to transfer a file stored in the memory module to the memory in the kit. We will not use the memory module in this course. |
| B | Save | Use Save to transfer a file from the kit memory to the memory module. We will not use the memory module in this course. |
| C | Dup | Press Dup (Duplicate) to copy a fie from one memory module to another. |
| 7 | Break | Use the Break key to set a breakpoint in memory. |
| 8 | W Reg | Use W-Reg to break a program when a register value reaches a desired value |
| 9 | W Loc | Press the W-Loc key to break a program when an address location reaches a desired value. This mode is similar to Break. |
| 4 | M Blk | Use the M-Bkl key to move a block of data in memory from another memory location. |
| 5 | I Blk | Press the I-Blk key to set a block of memory to a desired value. First enter the start address, then the end address. |
| 6 | Down | Press Down to download a Motoral Shex file into the kit's memory. |
| 1 | Exm Mem | Pressing this key in the select mode lets you examine the contents of any memory location. |
| 2 | Exm Reg | Pressing this key in the select mode allows you to modify any of the CPU registers |
| 3 | Go | Press Go to run a program in memory. The default start address is 0000H, but you can specify another address. |
| 0 | SS | Allows for single stepping the program. |
| | NMI | Press NMI to interrupt any program running, or operation being performed by the trainer. The memory is not modified. |
| | Reset | Press Reset to reinitialize the Trainer to its power-up state. All breakpoints are erased. |

To start, we pressed F followed by 0000 to insert starting at address 0000. Then we followed the assembled program and entered it byte-by-byte (2 digit values). Using the – and + keys allows the user to go up and down in memory one address at a time to review what was entered. If a mistake was made, although it is possible to change it while in insert mode, when a value gets re-entered, the addresses that come after the one that just got changed will be overwritten. Instead, press NMI, press 1 to enter examine memory mode and change the values there. Again, using the – and + keys will allow the user to go up and down in memory one address at a time. Once we finished examining our memory, we pressed NMI again. At this point, we were ready to make the processor start running the program. To do this, we pressed 3 followed by 0000 to tell the processor to go to run program in memory starting at address 0000. When the program ran, two strings appeared in the LCD display:

Programming is
Easy and Fun!!!

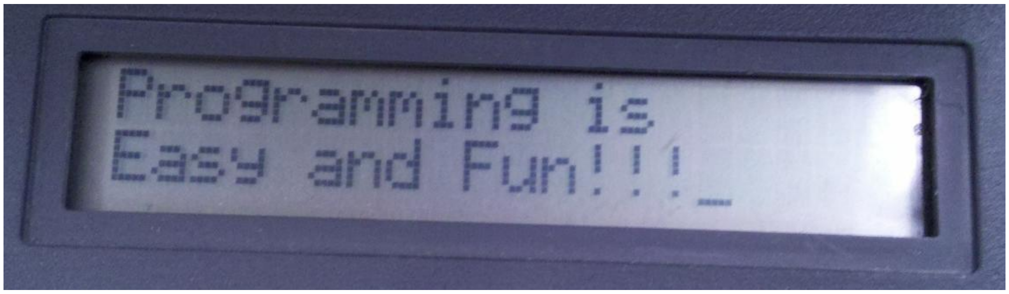


Figure 4. LCD Display Screen Showing Output of Entered Program.

These strings appeared one line at a time (the first string appeared first at the bottom of the LCD display screen, then the first string moved up to the top of the LCD display screen when the second string took the first string's former position). After a few seconds, the strings disappear.

**Writing Own Program**

Next we wrote our own program to assemble by hand and enter into the kit. The assembly code was written as follows:

```
       ORG  0000    ; specifies starting address 0

a      FCB   2       ; 2 is stored at 0000
b      FCB   3       ; 3 is stored at 0001
result FCB   0       ; 0 is stored as placeholder at 0002

       ORG   0010    ; specifies starting address 10

       LDAA  00      ; get Mem[0000] in ACCA (direct addressing)
       LDAB  01      ; get Mem[0001] in ACCB
       ABA           ; ACCA <- ACCA + ACCB
       STAA  02      ; Mem[0002] <- ACCA
```

Using the 68HC11 Instruction Set, we were able to assemble the code into the corresponding instruction bytes (made up of the opcode and operand). When we did this, we got the following:

```
ADDR BYTES
0000 02
0001 01
0002 00
...
0010 96 00
0012 D6 01
0014 1B
0015 97 02
```

We entered the above assembled program into the kit as in the previous section.

**Running Written Program**

After we entered our program into the kit, we ran our program. To do this we first initialized ACCA and ACCB to 0 by pressing 2 to enter examine register mode and using – and + to get to registers ACCA and ACCB. When at the correct registers, we set the values to 00. Then we pressed NMI. Instead of launching the program using the Go key as before, we single-stepped our program, starting at address 0010. If we were to use Go, our program would have crashed since the values stored in memory at addresses 0003 through 00ff contain code that does not belong to our program. To single-step, we pressed 0 followed by 0010. The display showed the next instruction that the kit is about to execute:

```
0010 96 00
LDAA 00
```

Then we pressed 0 continuously to single step until the kit was about to execute the ABA instruction, ie. the following appeared on the screen:

```
0014 1B
ABA
```

We then took a look at the registers by pressing 2 and using the + and – to navigate the through the list of registers. When we looked at registers ACCA and ACCB, they had changed from 0 to 2 and 3 respectively, which is what our program was written to do. We then went back to single-stepping through our program by pressing RPO to return to previous operation and then 0 to keep single-stepping. When the screen displayed SUBB #1B, we knew that our program had reached its end since the display was showing the instruction of the next address, which contained random code. We then pressed 2 to examine registers ACCA and ACCB again. ACCA now contained 5, which is the sum of its old value plus the value in ACCB, which was still 3. After pressing RPO, we then pressed 1 to check if ACCA was stored correctly in memory. Using the – and + keys, we found that [0000] = 2, [0001] = 3, and [0002] = 5, which suggested that the value in ACCA was stored correctly in memory.

**Endless Loop**

We then added a jump instruction at the end of our program to create an endless loop such that the jump would always take us back to the beginning of our program:

```
       ORG  0000    ; specifies starting address 0

a      FCB   2       ; 2 is stored at 0000
b      FCB   3       ; 3 is stored at 0001
result FCB   0       ; 0 is stored as placeholder at 0002

       ORG   0010    ; specifies starting address 10

       LDAA  00      ; get Mem[0000] in ACCA (direct addressing)
       LDAB  01      ; get Mem[0001] in ACCB
       ABA           ; ACCA <- ACCA + ACCB
       STAA  02      ; Mem[0002] <- ACCA
       JMP   0010    ; jump to beginning of program
```

```
ADDR BYTES
0000 02
0001 01
0002 00
...
0010 96 00
0012 D6 01
0014 1B
0015 97 02
0017 7E 00 10
```

When we single-stepped our program as before, when we got to the point where the screen displayed that the kit was about to execute the jump instruction, we pressed 0 again and the screen displayed that it was going to execute the load to register A instruction at address 0010, which is the beginning of the program, verifying that the jump took us back to the start of the program.

**Five Fibonacci's**

For this part of the lab, we wrote a program that computes the first 5 fibonacci's using the following algorithm:

```
1. fib[1] = 1
2. fib[2] = 1
3. fib[3] = fib[2] + fib[1]
4. fib[4] = fib[3] + fib[2]
5. fib[5] = fib[4] + fib[3]
```

We wrote the following and entered it into the kit:

```
; Compute the first 5 fibonacci values and store them in an array.

                          ;--- data section ---
                          ORG   0000
    0000 01       fib     FCB   1,1,0,0,0  ; create an array of 5 bytes
    0001 01
    0002 00
    0003 00
    0004 00
                  ;--- code section ---
                          ORG   0010
    0010 CE 00 00 START:  LDX   #0000      ; IX = 0, addr 1st byte of array
    0013 A6 00            LDAA  0,X        ; ACCA = mem[0]
    0015 AB 01            ADDA  1,X        ; ACCA = ACCA + mem[1]
    0017 97 02            STAA  02         ; mem[2] = ACCA
    0019 AB 01            ADDA  1,X        ; ACCA = ACCA + mem[1]
    001B 97 03            STAA  03         ; mem[3] = ACCA
    001D AB 02            ADDA  2,X        ; ACCA = ACCA + mem[2]
    001F 97 04            STAA  04         ; mem[4] = ACCA
```

Note that our code only works with arrays starting at memory address 0000. After we entered our program into the kit, we single stepped it starting at 0010 as before. After we executed the last line of our program, we pressed 1 to examine the contents of memory. When we did this we found: [0000] = 1, [0001] = 1, [0002] = 2, [0003] = 3, and [0004] = 5, which verified that our program correctly computed the first five Fibonacci values.

**Preparation for Next Lab**

In preparation for the next lab, we wrote a small program as an endless loop that reads a byte from memory address 0, increments it by 1, and stores the result back at that address:

```
; An endless loop that reads a byte from memory address 0, increments it by
; 1, and stores the result back at that address.

                          ;--- data section ---
                          ORG   0000
     0000 00       cnt    FCB   0  ; start cnt at 0

                   ;--- code section ---
                          ORG   0010
     0010 96 00    START: LDAA  cnt        ; ACCA <- cnt
     0012 4C       LOOP:  INCA             ; ACCA <- ACCA + 1
     0013 97 00           STAA  cnt        ; cnt <- ACCA
     0015 7E 00 12        JMP   LOOP       ; jump to 0012
```

This program will have a loop that takes a total of 8 cycles: 2 cycles for INCA, 3 cycles for STAA cnt, and 3 cycles for JMP LOOP.