

ELE538 Microprocessor Systems Lab 1: Using the *Monitor* Program & Introduction to Assembly Language Programming

Peter Hiscocks
Ken Clowes

Department of Electrical and Computer Engineering
Ryerson University

phiscock@ee.ryerson.ca
kclowes@ee.ryerson.ca

August 25, 2002

Contents

1 Objectives	2
2 Preparation	2
3 Requirements	2
4 Theory	3
4.1 Number Representations	3
5 Setting up	4
5.1 Using the Terminal Emulator	5
5.2 HELP command (H)	5
5.3 Memory Dump (MD)	6
5.4 Exercises	7
5.5 Memory Modify (MM)	8
5.6 Questions	9
6 Devices: Hardware Input and Output	9
6.1 Register Modify (RM)	9
7 Hand Assembly	11
7.1 Exercise	11

8 Writing an Assembly Language Program	12
8.1 Assemble the Program	14
8.2 Upload the Program	15
8.3 Check the Program	15
8.4 Run the Program	16
9 Breakpoints and Debugging	16
9.1 Crash during Breakpoint	16
10 Assignment	17
11 Additional Readings	17
12 Appendix: The <code>as11n</code> Assembler	17
13 References	18

1 Objectives

This tutorial introduces the basic lab environment used in the course. Once you have completed the lab, you will be able to:

1. Use the Buffalo Monitor for basic operations such as examining and modifying memory locations and registers.
2. Using the 'hand assembly method', create and run a tiny program at full speed (with or without breakpoints).
3. Using a text editor, create a complete assembly language program, assemble it, load it into the computer, and run it.
4. Use breakpoint and memory dump techniques to debug problems with a source code program.

2 Preparation

Read these lab notes to get a general idea of the concepts and work required.

3 Requirements

To get credit for the lab, you must:

1. Setup the required environment (i.e. directory structure, access to `ele538` commands.)
2. Be able to demonstrate knowledge of how to use the basic Buffalo Monitor commands.
3. Demonstrate an assembly language program to multiply two integers. This should include a copy of the listing file and may required demonstration of breakpoints.

4 Theory

A little bit of theory can help you understand the lab environment you will be using.

The MPP 6811 microprocessor board and the EEBOT robot are the ultimate targets of the embedded systems you will design in the course. The microprocessor is set up to run a special program, called the Buffalo Monitor, when it is turned on or the reset button is pushed. The Monitor software reads user commands, executes them and displays information back to the user.

The Monitor is designed to communicate with the user through a serial port connected to a dumb terminal. In the lab, the MPP board is connected to a serial port on the workstation which emulates a dumb terminal in a command-line window.

The Monitor Program in this development system is BUFFALO, which as a matter of interest stands for *Bit User Fast Friendly Aid to Logical Operations*¹.

Typing a command into the terminal emulator causes it to be sent to the microprocessor development system, where the command is executed by the monitor program. For example, typing in 'MD' instructs the monitor program to do a 'Memory Dump', ie, send the contents of some area of microprocessor memory.

The monitor program is a low-level (ie, very simple and primitive) method of sending programs to the development system and executing various debugging commands on the development system. As such, it is a very basic tool that is used in the early stages of getting software to work on the microprocessor development system.

The 6811 system has a memory space of 64 Kbytes. Memory locations are specified with 16-bit numbers expressed in hexadecimal. Consequently, the memory space range is 0x0000–0xFFFF.²

The complete memory map of the microcomputer development system is shown in figure 1 on page 7. Some of the memory locations are occupied by bytes of RAM, others by ROM or device registers and some memory locations are not used at all. It is important to note that not all memory locations behave in the same way.

For example, if you write 0x1A to a byte of RAM and then read that location back, you will of course read what was just written. However, if you write to ROM, the contents of the byte will not be changed. Locations mapped to device registers can behave even more strangely.

4.1 Number Representations

When you interact with the Buffalo monitor, byte contents are always input and displayed in hexadecimal format. As a programmer, however, it is often more convenient to think in terms of decimal numbers (signed or unsigned), binary representation or ASCII character codes. To use the Buffalo Monitor effectively, you should be able to translate between these different representations.

For example, suppose that a byte contains 0xB6. As a binary number, this is expressed as 0b10110110. (Note that we use the convention of numbering the individual bits of an n-bit binary number from n-1 to 0 where bit-0 is the least significant bit. In this case, we can say that bit-7 is a 1 and bit-0 is a 0.) If the number is meant to represent an unsigned quantity that is normally thought of as a decimal number, you would convert 0xB6 to the decimal number 182. If, however, it was meant to represent a signed quantity, then you would convert it to -74.

As another example, a programmer may want to see if a 16-bit variable that should be 61466 really has that value. If the number is encoded into two sequential memory locations, the Buffalo monitor allows you to see that the most significant byte is 0xF0 and the least significant byte is 0x1A. Simple conversion shows that, indeed 0xF01A (unsigned) is 61466. (Similarly, as a signed number, 0xF01A is equivalent to -4070.)

¹Some other genius at Motorola thought of this. Don't blame me.

²The prefix 0x is the *C style* method of indicating *hexadecimal* format. Sometimes the \$ is used for the same purpose. You should be familiar with both formats, since your two professors on this course have differing preferences. In the case of the BUFFALO monitor program, it **assumes** that all numerical values are in hexadecimal, and no preface is used at all.

If you have forgotten how to convert between numbers of different bases or how the twos-complement method for representing signed numbers work, this is a good time to review them.

5 Setting up

1. Enter the command:

```
which as11n
```

The response should be:

```
as11n not found
```

If the command was found, ask the lab instructor for help.

2. Type in the following commands exactly as shown:

```
touch ~/.myzshrc  
echo 'source ~/courses/ele538/.stdsh' >> ~/.myzshrc
```

3. Now log off and log back on again.
4. Once you are logged back on, enter the command `which as11n` again. This time the response should be:

```
/home/courses/ele538/bin/as11n
```

5. If you do not obtain this, carefully check your typing to try to figure out what you did wrong. Ask your instructor for help if necessary. Do **NOT** continue until you get the proper response to the `which as11n` command.
6. Using the Unix `mkdir` command, Create a directory called `ele538` and a subdirectory `lab1` within it. (i.e. the directory `/ele538/lab1` should now exist.) Change to that directory. The rest of the lab should be conducted from `~/ele538/lab1`.

It is essential that you be well organized in this course, so we recommend that you

- Create a separate subdirectory for each lab
- Number in sequence each version of any program you write, so that you always know which is the most recent. (The time and date stamp on the file will also help with this.)
- Carefully back up any work to floppy disk (or use some other method of backing it up, such as emailing it to yourself.)

What this means

It's not absolutely necessary to understand what is going on in the previous section, but it may help, so here goes.

This section is all about ensuring that the correct version of the assembler program `as11n` is in place. The assembler is a program that converts a text file into executable binary codes, and is an essential component in program development. The original assembler was `as11`. Professor Clowes fixed several shortcomings of that program and generate the new version, `as11n`.

The `which as11n` command searches your `PATH` to see if a give program `as11n` exists. The `PATH` lists directories where executables are found, so this command essentially asks if the executable `as11n` exists and will run when commanded.

The commands

```
touch ~/.myzshrc
echo 'source ~/courses/ele538/.stdsh' >> ~/.myzshrc
```

ensure that your shell is set up with the correct `PATH` variable to find the new `as11n`.

5.1 Using the Terminal Emulator

Execute the terminal emulator program on the host computer, by executing the command `'vt6811'`. Power on the microprocessor development system. You should see a sign-on message from the microprocessor system. This establishes that it is alive and sending a startup message to the host computer.

In all the following commands, the command string is typed in, followed by <carriage return>.

When you are done, <ctrl>-c will exit you from the terminal emulator.

5.2 HELP command (H)

You can now type commands to the Buffalo Monitor. When you are finished, exit the terminal emulator program with control-C.

The command `HELP`, for example, causes the microprocessor system to list the available commands, as shown below:

```
ASM [<addr>] Line asm/disasm
  [/,=] Same addr,      [^,-] Prev addr,      [+ ,CTLJ] Next addr
  [CR] Next opcode,    [CTLA,.] Quit
BF <addr1> <addr2> [<data>] Block fill memory
BR [-][<addr>] Set up bkpt table
BULK Erase EEPROM,          BULKALL Erase EEPROM and CONFIG
CALL [<addr>] Call subroutine
GO [<addr>] Execute code at addr,          PROCEED Continue execution
EEMOD [<addr> [<addr>]] Modify EEPROM range
LOAD, VERIFY [T] <host dwnld command> Load or verify S-records
MD [<addr1> [<addr2>]] Memory dump
MM [<addr>] or [<addr>]/ Memory Modify
  [/,=] Same addr,  [^,-,CTLH] Prev addr,  [+ ,CTLJ,SPACE] Next addr
  <addr>O Compute offset,          [CR] Quit
MOVE <s1> <s2> [<d>] Block move
```

```

OFFSET [-]<arg>  Offset for download
RM [P,Y,X,A,B,C,S] Register modify
STOPAT <addr> Trace until addr
T [<n>] Trace n instructions
TM Transparent mode (CTLA = exit, CTLB = send brk)
[CTLW] Wait,          [CTLX,DEL] Abort          [CR] Repeat last cmd

```

Try it.

Notes

1. The commands are case-insensitive (so HELP and help are equivalent.)
2. The commands may be abbreviated by using only enough letters to make the command unambiguous. (For example, the only command starting with the letter G is GO; it may be entered as the single letter G.)
3. Hitting the return key invokes the previous command. To repeat a command as rapidly as possible, enter it once and then hold down the return key, which will auto-repeat.
4. The command line editing facilities are essentially non-existent. If you make a typing mistake, hit <ctrl>-x or <ctrl>- and to get back to the monitor prompt, where you can start over again.

5.3 Memory Dump (MD)

One of the most commonly used commands allows the user to examine the contents of memory. The MD command has the following syntax: MD addr1 addr2

The two addresses (expressed as four-digit hexadecimal numbers) give the range of memory locations to look at. Each memory location is read and displayed as a two-digit hex number. Each line of the memory dump displays the contents of 16 sequential bytes of memory. The initial 4-digit number is the address of the first byte displayed.

Example

The command md e060 e072 gives the following output:

```

E060 A3 08 20 F0 32 39 48 45 4C 4C 4F 20 57 4F 52 4C      29HELLO WORL
E070 44 20 56 39 36 00 48 43 31 31 20 4D 50 50 20 28 D V96 HC11 MPP (

```

Notes

1. In this case, the first byte displayed 0xA3 is located at address 0xE060. This is equivalent to the unsigned decimal number 163 or, if interpreted in twos-complement format, to the signed decimal number -93.
2. The byte at location 0xE072 is 0x56. This is equivalent to decimal 86 (in both signed and unsigned formats).
3. Note that following the 16 hex numbers are an additional 16 columns of single characters (many of which are the space character). For example, the contents of 0xE070 is 0x44 which corresponds to the ASCII code for the character D.

The complete memory map of the microcomputer development system is shown in figure 1.

Notice the EPROM, RAM and EEPROM areas. The monitor program begins at memory location \$E000 and ends at location \$FFFF. (The '\$' sign indicates hexadecimal notation.)

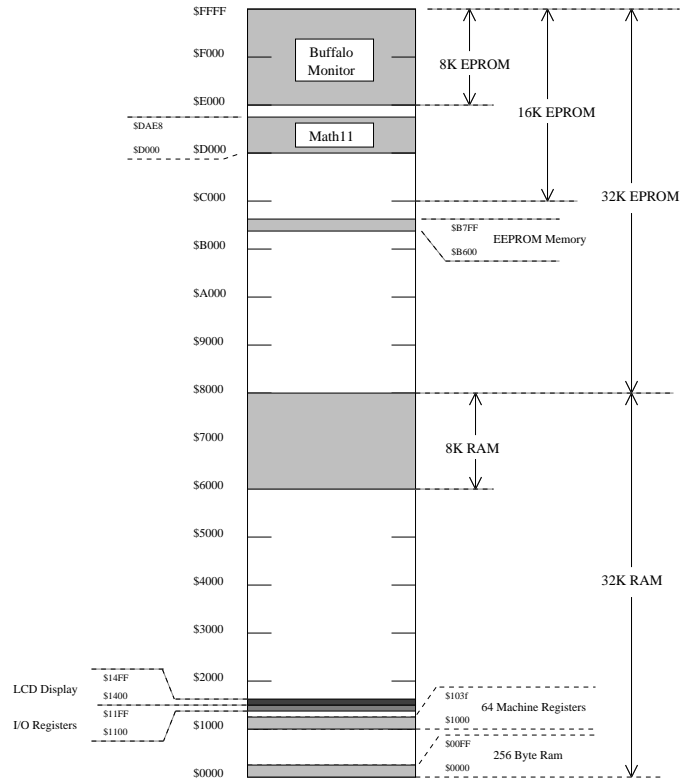


Figure 1: Memory Map

5.4 Exercises

1. Do a memory dump from 0xE650 to 0xE67f.
2. What are the contents (in hex, binary, decimal both signed and unsigned and ASCII) of memory location E653?

5.5 Memory Modify (MM)

The MM command allows modifying the contents of one or more memory locations. The syntax for the command is:

MM address

For example, typing the command:

```
>mm 6000 (then hit the enter key)
```

produces the result:

```
6000 FF
```

Once you get the result, there are three choices:

- Enter a 2-digit hex number. Notice that two digits are always required for the data. For example the value \$E would be entered as 0E.
The memory location is modified. You can then hit the space bar to view the next memory location or hit the ENTER key to terminate the command.
- Hit the space bar. The current byte is not modified and the next byte is displayed.
- Hit ENTER. This terminates the MM command. If it is the first key you type, then the MM displays the contents of a single memory location and does not modify it.

Example: If you type in mm 6000, you will see:

```
>mm 6000  
6000 FF
```

(where the '>' symbol is the monitor prompt.) If you then type 11 and hit the enter key, you will see:

```
>mm 6000  
6000 FF 11
```

A memory dump then confirms that the location has been modified as shown below:

```
>md 6000 6000  
6000 11 FF FF FF FF FF FF BF 04 65 81 20 05 30 08 00 e 0
```

Multiple bytes may be modified in one mm operation, as demonstrated here:

```
>mm 6000  
6000 FF FF FF FF 34 FF 56 FF 77  
  
>md 6000 6000  
6000 FF FF FF 34 56 77 FF FF 00 00 00 00 00 00 00 00 4Vw
```

5.6 Questions

1. Determine the ASCII character codes for the first four (4) letters of your userid (i.e. the user name you type in response to the "Login:" prompt on the workstations.) Modify memory locations 0x7000-0x7003 so that they contain the 4 ASCII codes and ensure that the next memory location (0x7004) contains 0x00. Show the sequence of commands required and the resulting memory dump.
2. Memory location 0xE000 is in ROM (Read-Only memory). What happens if you try to change it to something else? Explain.
3. The 68HC11 contains 512 bytes of EEPROM memory starting at location \$B600. EEPROM is memory cells that can be individually changed, and will contain their contents even when the power is cycled. However, the write cycle is quite slow (by computer standards). A normal write to RAM occurs in one machine cycle, a few microseconds. A write to EEPROM requires several milliseconds to complete. The MM command must take this into account when writing to EEPROM. Later versions of the monitor program MM contained these special instructions for writing to EEPROM.
4. Check whether memory locations in the EEPROM area can be modified by the MM command.

6 Devices: Hardware Input and Output

The Memory Dump and Memory Modify commands can be very useful in debugging hardware that is attached to the microprocessor. Often, the hardware signal lines of an external device are 'mapped' into the memory address space.

Hardware Inputs

For an input device (signals from an external device into the microprocessor), this means that each logic signal appears as one logic bit in some location of memory. Using the MD command, one can read that location to determine whether a particular electrical signal is being read as a logic 0 or 1. The MD command produces an output in the form of a hexadecimal byte, so it's necessary to convert the data to binary format to interpret it properly.

For example, if a particular hardware data input location read as \$AA, then it's binary value is 10101010. This tells us that bit zero is read as zero volts, bit 1 as +5 volts, and so on.

Hardware Outputs

When hardware output lines are mapped into the address space at a particular location, the logic level on these particular lines may be changed from zero volts to 5 volts by changing the contents of some memory location from logic zero to logic 1. For example, in the lab robot, the starboard motor enable is bit (something) of location (something). A logic 0 turns the motor off, a logic 1 turns it on.

6.1 Register Modify (RM)

The 68HC11 microprocessor contains a number of 'machine registers'. A diagram of these registers, called the 'programming model', is shown in figure 2. (See the 'HC11 Reference Manual', Section 6).

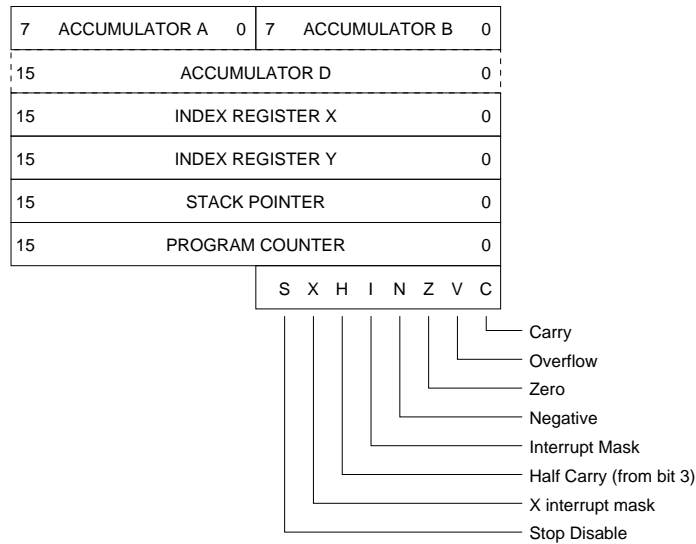


Figure 2: 68HC11 Programming Model

This diagram represents the registers that are internal to the central processing unit of the microprocessor. It ignores for example, input and output registers that control external hardware.

The registers have various functions:

- | | |
|-------------------------------|--|
| Accumulator A (8 bits): | Math operations and general purpose storage |
| Accumulator B (8 bits): | Math operations and general purpose storage |
| Accumulator D: | Operations that take place on both A and B are treated as taking place on the 16 bit D register. regarded as a 16 bit accumulator. |
| Index X (16 bits): | Pointer operations |
| Index Y (16 bits): | Pointer operations |
| Stack pointer SP (16 bits): | Pointer into the stack region |
| Status Register (8 bits?): | Various device flags |
| Program Counter PC (16 bits): | Pointer to the address of the next instruction to execute |

For our purposes at this point, the registers of interest are the ACCUMULATOR A and the PROGRAM COUNTER.

When the monitor program starts, it copies the machine registers to a storage area in RAM. Using the monitor Register Modify instruction RM, the programmer may modify the values of these various stored register contents. When the programmer executes the GO instruction, the monitor program rewrites these new register values into the machine registers and starts program execution at the memory address specified by the contents of the Program Counter register. (You can also start a program at some location \$1234 by entering the command GO 1234).

- Execute the RM instruction and examine the contents of each machine register.

Now we have enough information to enter and run a small computer program, using a method called 'Hand Assembly'.

7 Hand Assembly

Using a method called 'hand assembly', it is possible to use the Memory Modify and Register Modify commands to install a program on the development system. We'll do this with a program to increment (increase by 1) the contents of the accumulator register. In symbolic form the program is:

```
INCA ; Increment the A accumulator
INCA ; ditto
SWI ; break to the monitor program.
```

Each INCA instruction increases the contents of the accumulator A by 1. If we initialize ACCA to 00, it should contain 02 when the program breaks to the monitor.

SWI stands for SoftWare Interrupt. For now, it's only required to understand that an SWI command causes the machine to terminate the current program and switch to the monitor program. The details of how this occurs require some understanding of the stack mechanism and interrupts, and is in the Technical Manual.

7.1 Exercise

Manually convert each instruction into its corresponding operation code, and then enter the computer program into the microcomputer. In detail:

- Use the MC68HC11 Reference Manual (aka 'The Pink Book') to look up the 'operation code' for the INCA instruction and the SWI instruction. These instructions have a 'one byte' op code, which keeps things simple. Other instructions could require up to 4 bytes in total.
- Decide on a suitable starting address (the 'Origin') for the program. The start of RAM, location \$6000, is a suitable choice.
- Use the MM command to enter the sequence of three bytes, starting at the chosen origin.
- Use the MD command to verify that the instructions have been stored correctly.

Next, you need to set up the machine registers to run the program:

- Use the RM command to set the ACCA to 00.
- Use the RM command to set the PC to the value of the Origin address
- Don't change any of the other register values. If you make a mistake, press <ctrl>X to abort the current command, and then start over. The monitor is too primitive to have much in the way of command line editing.
- Use the RM command once again to examine the register contents and ensure that they are correct.

Now run the program by executing the Go (G) command.

The computer should break to the monitor with the program counter pointing at the next byte that would be executed. The accumulator ACCA should contain the expected result. Check that this is the case.

This 'Hand Assembly' process is far too tedious and error-prone to use for entering anything more than a few lines of program code. It is however useful for modifying an existing program during debugging, a process known as 'patching' the code.

8 Writing an Assembly Language Program

An assembly language program is the symbolic description of a program that is to be translated into machine language and loaded onto the microprocessor. For a high level language, such as C, one instruction usually corresponds to many machine instructions. For assembly language, each instruction in the source code corresponds to one machine instruction. The resultant translation into machine language is referred to as ‘the program binary’.

The process is as follows:

- Using the editor of your choice on the host machine (Sun workstation or PC), write the assembly language source program and save it as a text file, which we will call `firstprog.asm`.
- Convert the program to machine language by processing it with the 68HC11 assembler `as11n`. This generates a ‘listing file’ `firstprog.lst`, which shows the source code and its corresponding machine translation. It also shows where in the memory map the machine codes are stored.
- The assembler can optionally include in the listing file a ‘symbol table’ that shows the location of each symbolic address in memory.
- The assembler also generates an encoded form of the program binary that can be uploaded to the microprocessor, `firstprog.s19`. The format of the encoding is known as ‘S-Records’. The S Records file includes address information that specifies where the binary should reside in microprocessor RAM, and checksums that can be used to ensure that the uploading occurred correctly.

The AS11 assembler used in this exercise generates ‘absolute code’: the binary is assembled to run at its final location. This is relatively simple to use and understand, but it does not support the inclusion of library routines.

Here is `firstprog.asm`, the example of an assembly language program:

```
*****
*   Demonstration Program
*
* This program illustrates how to use the assembler.
* It adds together two 8 bit numbers and leaves the result
* in the 'SUM' location.
* Peter Hiscocks, August 2001
*****
ORG $6000                ; Program location
FIRSTNUM                FCB 01                ; First Number
SECNUM                  FCB 02                ; Second Number
SUM                     RMB 1                 ; Result of addition
ORG $6010
                        LDAA FIRSTNUM        ; Get the first number into ACCA
                        ADDA SECNUM          ; Add to it the second number
                        STAA SUM             ; and store the sum
                        SWI
```

The features of this program are as follows:

- A line beginning with an ‘*’ is ignored and may be used for commentary. The comments are extremely important. Assembly language is difficult to understand at the best of times, so header comments are vital.
- Comments may also be added to individual text lines, after a semicolon.
- ORG is an ‘assembly directive’. It does not translate directly into machine code, but advises the assembler. In this case, it specifies where the program is to be located. We have two ORG statements. The first one defines the start of ‘working storage’ in RAM, where variables will be stored. The second ORG defines where the ‘program text’ starts.
- Notice that hexadecimal values must be designated with a leading \$ sign. Without the dollars sign, the value is interpreted as a decimal value. This is an easy mistake to make, but it’s obvious if you check that the program loaded correctly into the memory area you expected.
- FIRSTNUM, SECNUM and SUM are ‘symbolic addresses’. They will be translated into specific addresses in the memory space. An assembly language program must adhere to a relatively strict format. In the case of this assembler, it is required that **only symbolic addresses begin in the first column, all other text (such as directives) must be indented at least one space or tab**. Notice that the symbolic names reflect their function in this program. This is an important documentary clue for those unlucky souls (including yourself, perhaps) that have to read and understand this program at a later date. Symbolic names can be up to 13 characters long and can contain the understroke character.
- The FCB (Force Constant Byte) reserves one byte in memory and initializes it to some value. Notice how symbolic names, FIRSTNUM and SECNUM, are attached to this reserved space.
- The RMB (Reserve Memory Byte) reserves one or more bytes in memory. They are not initialized to any particular value. Notice how a symbolic name, SUM, is attached to this reserved space.

Two other important directives are FCC and FDB:

FCC Force Constant Characters . This directive is used for defining a string message in memory. For example, the following stanza could be used to define a null-terminated string with symbolic starting address ‘HELLO’:

```
HELLO          FCC 'HELLO WORLD V96'
                FCB $00
```

The characters between single quotes will be translated into their ASCII values and stored, one per byte, in memory.

FDB Force Double Byte . This directive reserves and initializes two bytes of memory. It’s often used to store a 16 bit address in memory. For example, the following stanza will initialize the location VECTOR to the 16 bit address HELLO:

```
VECTOR        FDB HELLO
```

8.1 Assemble the Program

Using the editor of your choice (I prefer the Joe editor, but anything will work), type in the program as given above. When it is complete, save it as `firstprog.asm`.

Invoke the assembler program to process this file with the command line

```
as11n firstprog.asm
```

Notice that `as11n` is `as<eleven>n`, not `as<elel>n`. (A more detailed reference on `as11n` is in section 12 on page 17).

If there are no errors (caused by mistakes in typing, for example) the assembler will generate file `firstprog.lst` and possibly also the file `firstprog.s19`.

(If there are errors caused by mistakes in typing for example, error messages will appear on screen.)

Startup another window and load `firstprog.lst` into an editor in that window, so you can view it easily. The listing file `firstprog.lst` will look something like this:

```
original program by Motorola.
a few modifications by Randy Sargent (rsargent@media.mit.edu)
0001 *****
0002 *      Demonstration Program
0003 *
0004 * This program illustrates how to use the assembler.
0005 * It adds together two 8 bit numbers and leavesthe result
0006 * in the SUM location.
0007 * Peter Hiscocks, August 2001
0008 *****
0009 6000          ORG $6000          ; Program location
0010 6000 01       FIRSTNUM          FCB 01          ; First Number
0011 6001 02     SECNUM            FCB 02          ; Second Number
0012 6002          SUM              RMB 1          ; Result of addition
0013 6010          ORG $6010
0014 6010 b6 60 00          LDAA FIRSTNUM      ; Get the first number into ACCA
0015 6013 bb 60 01          ADDA SECNUM      ; Add to it the second number
0016 6016 b7 60 02          STAA SUM         ; and store the sum
0017 6019 3f              SWI              ; Break to the monitor
```

This is an extremely important file and contains essential information for debugging the computer program.

- In each line, the first number is the line number of the instruction.
- The second number absolute address where the instruction or data will be stored. Using this listing file, we know what computer memory locations are used by the program. We can use the memory dump instruction to check these locations to ensure that the correct instructions are at those locations, and to read out the program data.
- The symbolic instructions are converted into op code and operand. Notice how the instruction `LDAA FIRSTNUM` is converted into a one-byte op code `b6` and two-byte operand `60 00` that corresponds to the absolute address of `FIRSTNUM`. Refer to the `LDAA` instruction in the 68HC11 Reference Manual and confirm that the op code is correct for an absolute address.

The `SWI` instruction converts into a single-byte operand.

- If there is an error (which is the usual case when first assembling a program), the listing file will announce it. It usually takes several tries to get an error free assembly of a new program. However, notice that an absence of errors in the assembly code is no guarantee of correctness. A syntactically correct program can represent something that is logical nonsense.

When the source code `.asm` file is finally correct, the assembly operation will produce the listing file **and** an S19 file called `firstprog.s19`. The S19 file is an encoding of the assembled program that can be loaded into the microprocessor.

Now use an editor to examine the S record file. It will look something like this:

```
S1056000010297
S10D6010B66000BB6001B760023FF8
S9030000FC
```

Each line of the S Record contains a memory address where the information should be stored, the information itself, and a checksum. The final line of the file informs the loader that the file is completed. For our purposes, it is enough at this point to know that the S19 file exists and it is loaded into the microprocessor.

8.2 Upload the Program

There are various methods of loading the program into the 68HC11 development system, depending on the computer and the tools available.

On a Sun Workstation, the program VT6811 (written by Jason Naughton for use at Ryerson) will load an S record file and connect to the microprocessor monitor program. The command is:

```
vt6811 firstprog.s19
```

When the upload completes, the monitor will generate a 'ready' message³.

Important Note

: Only one vt6811 terminal session can be open at a time. You can have multiple windows running other programs open on the workstation, but you can't have any more than one vt6811 session operational because they fight over access to the serial port.

If you are having trouble getting vt6811 to work, be careful to ensure that you have not started other invocations of the program on your workstation, possibly in other windows.

8.3 Check the Program

Once the program is loaded, you should check that it is actually present on the development system. Use the MD 6000 command to dump development RAM. Compare the contents of memory with the information shown on the `firstprog.lst` file. They should match.

³Alternatively, if you are running the Linux operating system, you could use a terminal emulator such as Seyon. You would first establish contact with the monitor program, so that when you press reset on the 68HC11 development system, you see the signon message in the terminal emulator window. Next, execute the monitor 'load' command, L T (load from the terminal). Then select the 'upload text' option in the Seyon terminal emulator, and select the 'firstprog.s19' file. When the upload completes, the monitor will generate a 'ready' message.

On a PC system running Windows, you can use the Hyperterminal Accessory to communicate with the 68HC11 system. Execute the monitor 'L T' command to prepare the development system to load an S record file, and then execute the 'upload text' command in Hyperterminal.

8.4 Run the Program

Now you can execute the program with the command

```
G 6010
```

The program should break to the monitor at the correct location and display the contents of the machine registers. Check that the memory location SUM contains the correct result.

Use the Memory Modify command to change the contents of FIRSTNUM to 0A and SECNUM to FF. Rerun the program and verify the result.

9 Breakpoints and Debugging

In general, computer programs do not work correctly the first time they are run, and they must be debugged. The monitor Breakpoint instruction is very useful for this.

When you insert a breakpoint in a program, the program will exit to the monitor at that point. Then you can examine memory and the machine registers to determine what went wrong.

If everything seems correct at that point, then you can cancel that breakpoint and insert a new breakpoint farther along in the program.

For example, the command

```
BR 6013
```

would insert a breakpoint at address 6013, and the computer would switch to the monitor when the program counter reached that address, ie, just before the machine executes the instruction at address 6013. Notice that the breakpoint must correspond to the address of the instruction op-code. It can't refer to an address part way through an instruction.

It is possible to insert several breakpoints at the same time, but to keep things clear it's usually best to work with one at a time. However, there is one situation where multiple breakpoints are useful. If one is not entirely sure which branch a computer program is taking, you can put breakpoints in both branches and see which breakpoint is activated.

The command

```
BR -
```

removes all breakpoints.

Insert a program breakpoint so that the program breaks just after the addition of the two numbers. The sum should be in the A accumulator at that point.

In order to know the address at which to insert the breakpoint, you will need a copy of the assembled program.

Rerun the program and verify that the program does break correctly and that the number in the A accumulator is correct.

9.1 Crash during Breakpoint

The breakpoint mechanism works by replacing the instruction op code at the breakpoint with an SWI instruction. For this reason, the program under test must be in read-write memory (RAM). For example, a program in EPROM cannot be breakpointed.

Now, if there is a breakpoint set in a program under test, then an op code has been replaced with an SWI instruction. The monitor program has a record of this replacement so that it can restore the op-code when the breakpoint is removed using the BR - command. However, suppose there is a breakpoint set in the program and the program crashes. You then have to press the RESET button to get the monitor program back. The RESET wipes out any record of the original breakpoint, and the SWI instruction is still installed somewhere in the program text. Consequently, the BR - command will not work to remove the original breakpoint.

Faced with this situation, you could manually patch the code by replacing the breakpoint SWI instruction with the correct op-code information, which can be determined from the listing file. Alternatively, you would simply reload the program.

The moral is this: if you have any doubt at all about the program being corrupted after a crash, reload it.

10 Assignment

This assignment is due in Week 2 and must be demonstrated in the opening minutes of the lab session in Week 2.

Using the previous program as a model and referring to the 68HC11 Reference Manual, write, assemble and test a program to multiply two 8 bit numbers together. You may use the MUL instruction to do the multiplication operation. The input values should be held in two 8 bit locations named MULTIPLICAND and MULTIPLIER. The result should be held in a 16 bit location named PRODUCT.

The program should be written as an assembly language source code. It should have a descriptive header and include comments in the code. For example, your header should indicate that this is an *unsigned* multiply operation.

Demonstrate this program to your lab supervisor, using the first digit and last digit of your student number as inputs to the program. The lab supervisor will select two other input digits to use in your program.

You should be prepared show and explain the contents of the listing file, and to demonstrate the use of break-points as applied to the operation of this program.

11 Additional Readings

The fastest route to getting an understanding of assembly language is to read code that other people have written. One place to look is in the course directory `\library` subdirectory. This directory contains useful example routines which you can use in your own programs without penalty. As well, the Web page of Jim Koch, the Senior Tech Support Engineer for Electrical and Computer Engineering, has a link to the source code of the Buffalo monitor. This is a very large assembly language program, but there are many useful ideas in the code.

Reverse engineering other people's code is hard work at first, but once you get a basic understanding of the most common instructions and groups of instructions, it will go much more easily.

12 Appendix: The `as11n` Assembler

The assembler `as11n` is actually a shell script that acts as a pre-processor for a second program, `as11`, that does the bulk of the work. The shell script was written by Ken Clowes and basically makes it easier to use the assembler.

- Normal use to assembly file `foobar.asm`:

```
as11n foobar.asm
```

- If the assembly is without error, the script will generate the files `foobar.lst` and `foobar.s19`. The file `foobar.lst` contains an assembly listing and symbol table, both of which are useful for debugging purposes. The file `foobar.s19` is uploadable to the microprocessor development system.
- If there are errors, the existing file `foobar.s19` is renamed to `foobar.s19.OLD` so that there is no longer a working `.s19` file to confuse things. (If you want to go back to the previous `.s19`, use the Unix `mv` command to rename `foobar.s19.OLD` to `foobar.s19`).
- The listing `.lst` always contains the listing of the assembly together with any errors where they occur in the code.
- The command line

```
as11n -h
```

generates a help message. Appart from the `.asm` file name, this is the only possible parameter to `as11n`.

- The `as11n` script is contained in the course directory `/home/courses/ele538/bin`. You have to ensure that this directory is in your `PATH` variable to execute `as11n` easily. If you are interested in how the shell script works, it is a readable file.

13 References

bf *eebot* Technical Description

Peter Hiscocks, 2002

A complete technical description of the *eebot* mobile robot used in this course.

M68HC11 Reference Manual

Motorola Document M68HC11RM/AD REV 3, 1991

The authoratative source of information about the 68HC11 microprocessor.

Not an easy read.

68HC11 Microcontroller, Construction and Technical Manual

Peter Hiscocks, 2001

Technical information on the MPP Board, 68HC11 Microprocessor Development System

Information on programming and interfacing the MPP Board used at Ryerson and elsewhere.

M68HC11: An Introduction, Software and Hardware Interfacing

Han-Way Huang

Delmar Thompson, 2001

A basic text on the 68HC11 microprocessor.

Production Note

This file was created on a Linux system using the `joe` editor and typeset using the \LaTeX typesetting program. Diagrams were created with the `xfig` program. Previewing was by `xdvi` and conversion to postscript format by `dvi2ps`. Previewing of the postscript format used `gs` and conversion to `.pdf` format using the script `ps2pdf`. The contributions of Ken Clowes were provided in html format, manually converted to \LaTeX format and inserted into the text. The html rendition is derived from the XML file `ele538/labs/lab1/lab1.xml`. It was translated to html with the XML translation specification in `./article2html.xml` using a standard `xslt` command written in Java. The source file was written by Ken Clowes (`kclowes@ee.ryerson.ca`). The `dtd` and `xsl` files were written by `kclowes`. Everything was written using `xemacs` and maintained using `make` and the `bash` shell under WindowsME, Solaris and Linux operating systems.