COCO SIM11

GETTING TO KNOW THE 68HC11 SIMULATOR

The 68HC11 simulator is a DOS-based program that is installed on every PC in the lab. This program simulates the 6811 processor, and is useful as a first step in getting to know the machine before using the real hardware. With this, you can enter a program in assembly language, assemble the program using the assembler, and then run the program. The simulator is a bit more helpful compared to the real hardware. This is sort of like using LogicWorks to debug a circuit first before actually building the hardware.

Specifically, here are some aspects of the simulator.

It displays the states of all registers, flags, and memory locations as the program is executing.

Using the command line, we can go in and change any values as well.

It allows us to assemble or disassemble (the reverse of assembling!) one line at a time.

It alerts us when our program is accessing uninitialized memory locations, so we can catch our mistakes.

It allows us to have "breakpoints". With these, we can ask the simulator to pause at specific points in a program so we can go in and look at the registers etc.

It allows us to perform symbolic debugging. We can define symbols, and recall them. For example, we can type commands like the following:

`SYMBOL START $3400`	(defines a symbolic value)
`WHEREIS START`	(recalls a symbol's value)
`BR done`	(place a breakpoint at `done`)
`GO start`	(start executing from `start`)
`STEPTIL label3`	(single step through the program until `label3`)

It allows us to simulate input/output ports of the 6811 by using disk files.

We can also use the simulator's command line to change any registers or ports.

It works with the assembler. So, you can type in a program as a text file, assemble it using `asm`, and then run the program on the simulator. Chances are, you will catch some mistakes with the assembler,

then a few more with the simulator. If the program works on the simulator, you can move on to the real hardware. This route is usually more productive compared to direct experimentation with the hardware.

To access this software, first do a `Find sim11.exe`. Note it's location. Then bring up a DOS window on your computer and set the working directory to the one that contains `sim11.exe`.

Then, at the DOS prompt type `SIM11` to start up the simulator.

Entering Data and Code

When the simulator loads it will beep and present you with its debugging screen. The words "**PC not initialized**" may appear in the code window; that's normal. Take a moment to survey the screen. Locate the windows labeled "**CPU**", "**CC REG**", "**CODE F2**", "**MEMORY F3**", and "**DEBUG F1**". You will use these windows in the following exercises. Typing **F2** activates the code window, **F3** activates the memory window, and so on.

Pressing the "**F10**" key brings up the simulators' online help. The help window can be dismissed by typing **ESC**.

The DEBUG Window

At the simulator prompt in the **DEBUG** window, type `MM 6000 CC 24 68` and press return. You have just entered values **$CC, $24,** and **$68** into memory locations **$6000, $6001,** and **$6002.** Verify this using the "**Memory Display**" command `MD 6000`, and viewing the memory window.

The **default** number representation for the simulator is **hexadecimal**. You don't use a $ sign since the simulator assumes all numbers are in hexadecimal notation. **Recall,** the assembler's default is decimal.

The above three bytes can also be interpreted as the machine instruction `LDD #$2468` (Load accumulator D with hex constant 2468). It is put in the simulator memory starting at location **$C000**.

To verify this, set the simulator's program counter to **$C000** by typing `PC 6000` at the simulator prompt. The simulator will then disassemble from memory starting at this location and display the results in the code window in the upper right corner of the screen.

The instruction that the program counter points to is the one to be executed next; it is always shown highlighted on the second line of the code window. You should see the instruction `LDD #$2468` shown on this line. There may be other instructions appearing after this one, but we are not concerned with them since they are due to uninitialized values in the simulator memory. The simulator will attempt to disassemble anything it comes across, whether it is really code or not.

Let's execute the command that we typed in. First take note of the register window in the upper left corner of the simulator screen. Each of the 68HC11's data registers will be shown, along with their

current values. Notice what value is in the D accumulator. If you see X's, it means that this register is uninitialized.

Now at the simulator prompt, press `T`, then return. This command tells the simulator to trace through a single step. In other words, it executes the single instruction pointed to by the program counter and then stops. Any register or memory changes can then be viewed on-screen.

Question 1. What value is in the D accumulator now? $_____

Question 2. What did the instruction that we entered do? _____

Question 3. What values are in the A and B accumulators? A $_____ B $_____

Question 4. How are they related to the value in the D accumulator?_____

Single Line Assembler

It was easy to enter and execute a single instruction, but to enter an entire program with the memory modify (**`MM`**) command would prove to be a major inconvenience. The simulator has other more convenient ways to put programs into memory. The simulator contains a single line assembler that you can invoke with the `ASM` command. Type

```
ASM 6000 <return>
```

Notice that the `ASM` instruction changes the prompt to ">". You can type in one instruction per line, for as many lines as you want. Just press `return` on a blank line to terminate the assembler mode.

```
LDAA #5 <return> <return>.
```

Typing `<return>` enters a blank line and makes the simulator exit the `ASM` mode. Now type

```
PC 6000
```

to reset the program counter to **$6000**. Note that the instruction is shown in the code window at the location **$6000**. Now type

```
ASM <return> LDX #$8 <return> <return>.
```

Question 5. Into what memory location(s) was this instruction placed?_____

Using the single line assembler, you can enter small programs into the simulator. The cross-assembler program we will use later allows you to assign a value to a label, but you cannot define labels within the single line assembler. For example, the simulator will interpret the command

```
ASM <return> BRA 6012 <return>
```

But the simulator will not interpret the command

`ASM <return> BRA LOOP <return>`

(because `LOOP` is not defined).

Using the single line assembler, enter the following program into simulator RAM at address **$6000**.

```
        LDAA #$0
        LDX #$4
LOOP    ADDA #$1
        DEX
        CPX #$1
        BNE LOOP
        SWI
```

How do we handle the instruction `BNE LOOP`? It turns out that the simulator computes the branch offsets for us, so we only need to type in the **address** of the `ADDA #$1` instruction *in place of* `LOOP` in the `BNE LOOP` instruction.

Set the program counter to **$C000**, set the **Condition Code Register** (CCR) to `$50 (%01010000)` with the command `CC $50`, and then trace through the program step by step to see what it is doing. Note how the registers change after certain commands. (Record a change if any register changes on any execution of the loop.)

Question 6. Which register(s) change after the `ADDA` command is executed? _____

Question 7. Which register(s) change after the `DEX` command is executed? _____

Question 8. Which register(s) change after the `CPX #$1` command is executed? _____

You'll notice that typing `T` to trace through this loop can be an annoyance; and this is just a small loop. If you are waiting for something to happen at the end of this loop, would you want to sit around typing `T <return>` for each instruction? Especially when you might be writing loops that execute several thousands of instructions? There must be an easier way. There is! Sett the program counter to **$C000** again, and this time type `T 10 <return>` at the simulator prompt.

Question 9. What is happening now? _____

You'll notice that this is slightly faster than before. The screen updating after every instruction prevents the simulator from running at top speed. If you are waiting for the end of the loop, you are probably not concerned with how the registers are changing inside the loop.

Question 10. Was the number "`10`" that you just typed into the simulator a **decimal** `10` or a **hexadecimal** `10`? _____

Setting Breakpoints

Reset the program counter to **$6000**. Look at the address of the `DEX` instruction.

Type `BR <address> <return>` at the simulator prompt, where `<address>` is the address of the `DEX` instruction. You have just set a breakpoint, and it will be shown in the middle left portion of the screen. Now type `G` at the simulator prompt. `G` stands for **Go**, and it tells the simulator to begin executing instructions starting at the program counter (or the address if `G <address>` is given). You will return to the simulator prompt immediately.

Question 11. Did any registers or memory locations change? (Yes/No)_____

Question 12. Did the loop execute? (Yes/No)_____

Question 13. What address is in the Program Counter? $_____

Question 14. What is the purpose of setting a breakpoint? _____

Reset the program counter to **$6000** one last time and clear the previous breakpoint by typing `BR <return>`. Now type `GOTIL <address>` where `<address>` is the one used in the previous example.

Question 15. Briefly describe what happens.

You should now have a basic understanding of the 68HC11 simulator. A table of all of the commands available is available from within the simulator by pressing **F10**. Try this key, and explore the available commands. Ask your instructor for any clarifications that you may need. Remember that the help window can be dismissed by typing `ESC`. You can then go to any window on the screen by typing a suitable function key.

Question 16. What function key activates the memory display window? _____

Now, try few other commands to see what they do.