ASSIGNMENT 3: PLAYING WITH THE 68HC11

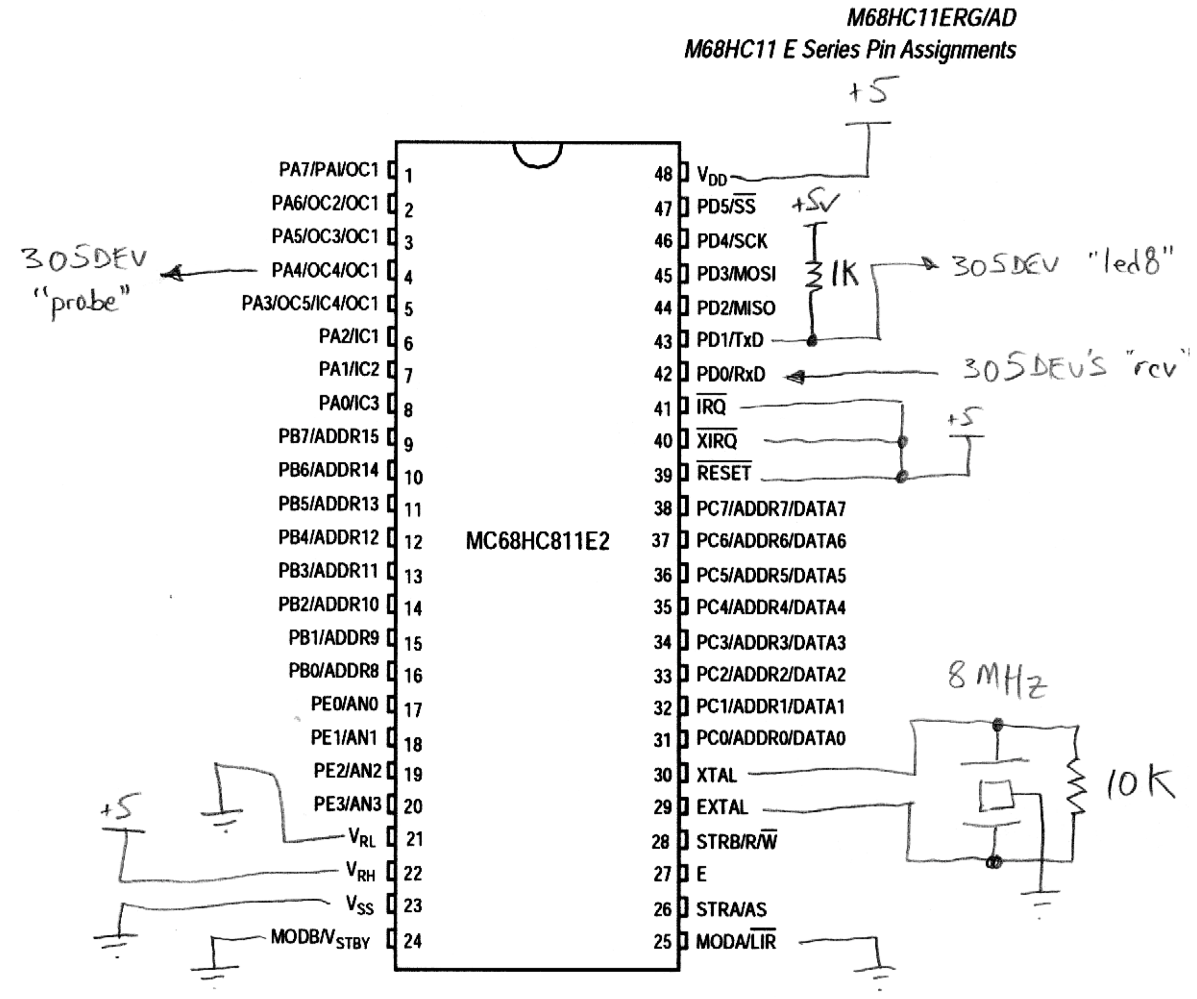
Due Wednesday, Oct 9, 2002

Introduction

In this assignment, you will (1) construct a minimal 68HC11 system on your breadboard, and (2) use the serial port of a computer to load programs into the HC11's internal 512 bytes of RAM (making use of the serial bootstrap feature of the CPU).

Initially, you will load a known-working program into the HC11 to verify that your wiring setup and serial line configuration is working. Afterward, you will modify and write new programs for the HC11 and use your setup to load them into the HC11 for testing and debugging.

Hardware Setup: HC11 Wiring

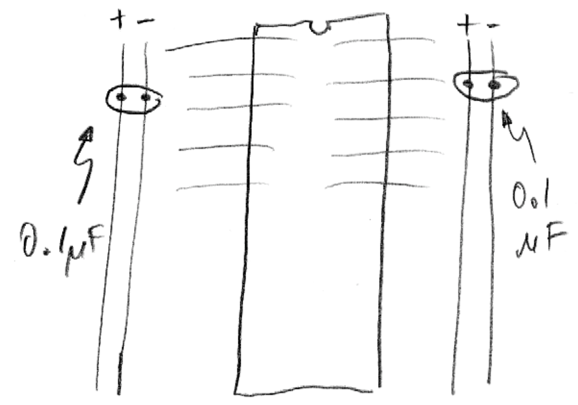


The drawing above shows how the 68HC11 chip should be wired. Here are a few notes of interest:

- Power is applied via Vdd (pin 48); ground is applied at Vss (pin 23).
- The pins ~RESET, ~XIRQ, and ~IRQ must all be tied high.
- The analog input references should be connected: Vrl to ground, and Vrh to +5v.
- The MODA (pin 25) and MODB (pin 24) lines must be tied to ground. This put the HC11 into the serial bootstrap mode when it "wakes up."
- There is a 1K pullup resistor on the HC11's TxD output. This is necessary because when the HC11 starts up in bootstrap, the pin can only assert ground, and must be pulled high to produce a logic one ("wired OR mode").
- The ceramic resonator is connected to the XTAL and EXTAL lines. The outer two pins of the resonator go to XTAL and EXTAL (either way—there's no polarity). The center pin of the resonator goes to ground.
- You must add a 10K resistor in parallel with the outer two pins of the resonator to get the oscillation frequency to stabilize.

Hardware Setup: Power Wiring

use 0.1μF capacitors ("104")
between power + ground on busses:



When wiring up the power busses, make sure to insert one 0.1μF capacitor (they're marked "104") into each power-ground bus. The diagram above illustrates this. The purpose of these capacitors is to smooth out fluctuations in the power supply, making sure the HC11 operates properly.

Software Setup

Loading the boot program into the HC11 consists of:

1. Turn on the HC11 with MODA and MODB held low. Your circuit (if wired properly) does this. The chip automatically powers on in serial boot mode.
2. Open a serial connection with parameters 1200 baud, no parity, 8 data bits, 1 stop bit (1200-N-8-1).
3. Write a 255 out the serial port. When received by the HC11, this is used to detect the baud rate.
4. Write 512 bytes of data out the serial port. These data are loaded into the HC11's internal RAM, and then execution automatically jumps to address 0.

You are free to use whatever software you like to run the serial port as described, but for your convenience, you may wish to use the Java routines provided. In this case, you need to either (a) install the Java Development Kit (JDK) on your computer, or (b) use the JDK already installed on the computers in the Olsen 314 machine room.

Instructions for installing the JDK or using the existing one are provided in a separate on-line document. Please see `http://www.cs.uml.edu/~fredm/courses/91.305/javasetup.shtml` for instructions.

HC11Boot.java

At end of this handout is the initial test program `HC11Test.java`. This program is also available from the course web site, and is linked from the `javasetup.shtml` page.

All this program does is load a tiny, three instruction program into the HC11:

```
      ldaa 0x10
      staa 0x1000
loop: jmp loop
```

Here is what this HC11 program does. First, it loads the value 0x10 into register A. Then, it stores this value to address 0x1000. Address 0x1000 is a special address—it is the location of the PORTA data register. Data that are written to this address appear as ones and zero on the PORTA pins. The effect of writing 0x10 to this address is that bit 4 of PORTA gets set to a one. Note from the HC11 datasheet that its default (power-on) value is zero. Thus, when the program runs, Port A, bit 4 (which happens to be pin 4 of the HC11) will change from zero to a one. Hooray!

The third line of the program creates an infinite loop, jumping to itself endlessly. The CPU must be doing something; it never just stops. So if you want your program to terminate, you should keep it busy by having it loop in a known fashion. Otherwise it will just plow along, executing whatever code it happens to find in memory.

Serial.java

The HC11Boot.java program uses a class that is defined in the file Serial.java. This provides the Serial object that implements rudimentary communication through a serial port.

PROBLEM 3–1: BOOT YOUR 68HC11.

You don't have to write any code here, you just have to put all the pieces just described together and get it working for yourself.

Build up the HC11 circuit as described. Connect the HC11's Port A4 output (pin 4) to the dev board probe display.

Get Java running on a PC.

Compile the `HC11Test.java` and `Serial.java` files.

Run `HC11Test.java`, and witness the Port A4 output (pin 4 of the HC11) go from zero to one, as evidenced by the probe indicator going from green to red.

Congratulations, your 68HC11 is alive!

If you got the PA4 line to turn red, you're ready to proceed. If not, you'll need to debug.

First make sure you don't have any wiring errors.

If this doesn't result in happiness, try to find either or both of (a) a known working HC11 setup, so you can test your development computer configuration, and (b) a known working computer setup, so you can test your board better.

If that fails, come to office hours/lab!

PROBLEM 3–2: BEEPING.

The following small HC11 program, `beep.s`, creates an oscillation on the Port A4 pin:

```
;;; beep.s
;;; toggles Port A4 with delay loops between each toggle,
;;; creating an oscillation in the audible range.

        ldx #0x1000             ; point at register base
loop:   bset 0,x,#0x10          ; set bit 4 in PORTA register
declp1: deca                    ; decrement reg A
        bne declp1              ;   and loop till it hits zero.
        bclr 0,x,#0x10          ; clear bit 4 of PORTA
declp2: deca                    ; decrement reg A
        bne declp2              ;   till it hits zero.
        bra loop                ; now recycle at the beginning.
```

The program works by setting and clearing bit 4 in the Port A register, making use of the bit set and bit clear instructions (`bset` and `bclr`). In between each set and clear, there is a delay that results from decrementing the A register till it hits zero.

Using the `as6811` assembler (available from the Resources: Software area of the course web site), assemble the program. Type at the prompt:

```
as6811 -l beep.s
```

so that the assembler produces a listing file. From the listing file (named `beep.lst`), look at the resultant object code. For instance, the first code line of the file assembles to:

```
0000 CE 10 00             5         ldx #0x1000        ; point at register base
```

The first column of numbers, "0000," is the location that this code assembled to. The second set of numbers "CE 10 00," is the object code for this instruction. The next number, "5," is the line number of this statement in the source file.

Examine the rest of the file so that you can find the object code.

Now, create your own copy of the `HC11Test.java` file. Rename it to `BeepTest.java` or similar.

Open the file in your favorite editor, and replace the Java class name `HC11Test` with your new filename (e.g., "`BeepTest`"). You should also replace the class name in the print statement in the `setup` method, though of course this doesn't affect functionality.

Now, replace the object code loaded into the `buf` array with the object code from your assembled `beep.lst` file. The first few lines should now look like:

```
buf[i++] = (byte)0xce;
buf[i++] = (byte)0x10;
buf[i++] = (byte)0x00;
```

Continue in this fashion, copying the whole of the HC11 beep program into the Java source file.

Now, compile the Java file, turn on your HC11 board, and run the Java program to download the HC11 code.

When the Java program has finished, the code should be running on your HC11. If you plug the output from the Port A 4 line (pin 4 of the HC11) into the 305DEV probe line, you should see both the red and green LEDs on at the same time (actually, they're flashing back and forth, but too fast for you to see).

Now, plug the PA4 line into the piezo input. You should hear a tone!

Problem 3–2a. Your HC11 has an 8 MHz oscillator. This is divided by 4 to result in a 2 MHz instruction clock (the E clock). In the 68HC11 Reference Manual there is a mapping from instructions to cycles; for example, the LDX #0x1000 instruction takes 3 cycles, or 1.5 microseconds, to execute.

What is the frequency of the tone that the program generates? **Your answer should be a frequency in cycles per second (Hz), along with a justification for how you got it.**

Problem 3–2b. Re-write the program to generate precisely 1000 Hz. Assemble it, copy the assembled object code into a new instance of the loader program, and run it to make sure it works. Hint: the NOP instruction does nothing but consume one byte of code space and 2 instruction cycles. **Turn in a printout of your working .lst file, plus a copy of the .java file you used to download it to the HC11.**

Problem 3–2c. Which tone is louder—the original tone, or the 1000 Hz tone? Note: this is somewhat subjective, but take a stab at it. **Turn in the answer by frequency, e.g., "xxxx Hz seems louder."**

PROBLEM 3–3: SERIAL COMMUNICATIONS.

In this exercise, we will see how the HC11 can send data back to the PC using the serial port. You will write an HC11 program that converts binary data to a hexadecimal number.

Demonstration

First, a demonstration of how to send data from the HC11 to the PC. The following program `serialxmit.s` will cause the HC11 to repeatedly transmit the ASCII character set beginning with code 0x21 (the exclamation point, '!') and ending with code 0x5a (the capital letter 'Z'):

```
;;; serialxmit.s
;;; counts from 0x21 to 0x5a on 68HC11 serial port and then repeats

base=   0x1000
scsr=   0x102e                  ; serial comms status reg
scdr=   0x102f                  ; serial comms data reg

        ldx #base               ; pointer to register base
loop:   ldaa #0x21              ; start with ASCII 0x21 to transmit
xmit:   staa <scdr,x            ; transmit reg a
        brclr <scsr,x,#0x40,.   ; loop here until byte transmitted
        inca                    ; inc char to be transmitted
        cmpa #0x5B              ; see if it's beyond ASCII 0x5a
        bne xmit                ; no, send the next one
        bra loop                ; reset char to 0x21
```

Let's look at the program in some detail.

The program begins by installing the value 0x1000 into the X register (the label `base` was previously assigned as 0x1000). This allows indexed-by-X instructions to be used to talk to the serial control registers.

Next, the value 0x21 is loaded into the A register. This will be the first byte that is transmitted.

At the next line, beginning with the label `xmit`, the value in the A register is written to the `scdr` (serial communications data register). This will cause the value in A to be written out the serial port. Note that the indexed-by-X addressing mode is used. This requires a one-byte value which is added to the value in X to form the extended address. The less-than-symbol < is an operator in the assembler which takes the low byte of a label's value. Hence, the low byte of the `scdr`, or 0x2f, is taken and added to the 0x1000 in X to point to the `scdr` register.

To avoid overrunning the register, it is necessary to check a "busy" bit in the `scsr` (the serial communications status register). The next instruction does this. This is the "branch-if-bit-clear" instruction, which uses an indexed-by-X addressing mode. The `<scsr` when added to X points to the status register. The `#0x40` operand indicates we are interested in testing the 2nd highest bit in this register (which, if you refer to the HC11 documention, is the Transmit Complete flag, with a 1 indicating it is complete). Finally, the dot at the end of the instruction is the address to which to jump. In the assembler syntax, the dot has a special meaning, which is: the address of the beginning of this instruction. Thus, the `brclr` will literally jump to itself if it bit is clear, which means that the serial transmit is still underway. Hence, we will be stuck here in this instruction until the transmit is done.

The next instructions increment the A register and loop in the appropriate way. After being incremented, the A register is compared with the value 0x5b (one higher than the last value we want transmitted). The compare value performs a subtraction (A – 0x5b, in this case) and then throws the result away. But the compare does set the condition code bits, and next we test if the result was "not equal to zero," using the `bne` instruction. If it wasn't zero, we want to transmit this next value of A, so we jump to the `xmit` label. If it *was* zero, then A must have been 0x5b, and we don't want to transmit that, so we jump to the `loop` label, which resets A to 0x21 before transmitting again.

The file `SerialXmitBoot.java` contains this program entered into the boot code. You can compile and run this file to get the program going.

Now, you need to run a terminal emulator on your PC in order to see what the code is doing. You may either (1) use a standard terminal emulator on your PC (e.g., HyperTerminal), with serial settings 1200-N-8-1 and no flow control, or (2) compile and run the supplied `SimpleTerm.java` code, which provides a rudimentary terminal emulator with the correct settings.

Regardless of the terminal emulator method, when you have it all working, you should see the following on your screen:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRST
UVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./01234
56789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-.
/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

etc.

Does it work? Make sure you have the HC11's serial transmit line (pin 43 of the chip) plugged into the "xmt" port of your dev board, not LED8!

Problem 3–3. Using the techniques presented in `serialxmit.s`, write an HC11 program that reads the value of the Port C register, converts it into an ASCII representation of its hexadecimal value, writes those two ASCII characters out the serial port, and then repeats. Also, between each "printing" of the Port C value, output the binary values 0xA followed by 0xD out the serial port. (These are the linefeed and carriage return characters, respectively, and will cause the terminal emulator to display your hex values each on their own line, rather than sprawled across the screen.)

Some notes on this exercise:

- Connect the eight Port C pins to your switch bank (pushbuttons and slide switches), so you can readily generate different values on Port C.
- The Port C register defaults as inputs in the HC11's bootstrap mode, so you can simply read the PORTC register (no other set-up is required) to read the pins.
- There is a sample ASCII table on page 58 of the Motorola M68HC11ERG document.
- Converting the Port C value to hexadecimal is relatively straightforward. You convert the high "nybble" (upper four bits) into the correct ASCII digit from 0 to 9 or A to F, send it out, and then convert the lower nybble and set it out.
- Masking operations (e.g., "ANDA #0xF0") and bit shift operations (e.g., "LSRA") should be helpful.
- Don't read the Port C register twice, first transforming the high nybble and then reading it again and transforming the low nybble! This could introduce errors if the Port C value changed between processing of the high nybble and the low nybble. Read it once and cache it between these two operations.
- You may wish to solve this problem incrementally rather than writing the whole entire code and then trying to get it to work! E.g., get the high-nybble conversion working and printing, then do the low nybble.

Turn in: (1) your assembled listing file (.lst) of your final code, and (2) a print-out of a screen-snapshot of the results, showing you got it working. The screensnap should show the pixels on your PC's display, not a cut-n-paste of the characters being printed to the terminal emulator.