Functions and bit manipulations

N. Natarajan

Department of Electrical and Computer Engineering
University of Michigan-Dearborn
4901 Evergreen Road
Dearborn-48128

nnarasim@umich.edu

1 Objective

To become familiar with bit level operations and writing functions. This lab also illustrates the use of random numbers for testing functions. Bit level operations are used to control light emitting diodes connected to PORTA as well as for monitoring external circuitry.

2 What you should do

You will be writing several functions in this laboratory exercise. You should have only one file and you should add the new function at the end of your older functions. Also, you will be adding items to the data section. You should **not** delete earlier data items. Your main code will be changing. You should insert your new main code before the older one, so that the most recent main code will start immediately after the `ORG` statement. Try not to delete any code from your file.

3 String outputs

In the last lab, we saw how to write a single byte as an ascii character. To write a string, it is tedious to load A with one character at a time and then calling `OUTA` each time. A better approach is to put all the characters in consecutive

memory locations and print them all in a loop. To do this, we need two pieces of information, where to start and where to end. The common approach to such situation is to specify where to start and use a special value, known as sentinel, to indicate the end. Some of you may have used special values such as zero, one, or 9999. It is entirely up to the programmer, but for character strings, the three most often used sentinels are zero (also known as ASCII-Z string), 26 (also known as CONTROL-Z string, or old DOS string), 4 (EOT string).

The programmers of BUFFALO use EOT string and you have one of two choices: rewrite BUFFALO routine and use some other sentinel, or use 4 as the sentinel and remember to place it after each string. The rest of the lab assumes that you will use the EOT string. Two functions that BUFFALO provides for printing strings are `OUTSTRG` at location `$FFC7` and `OUTSTRGO` at location `$FFCA`. The difference between them is that the former will print the string on a new line, while the latter will continue the string from wherever the cursor happens to be. Both these functions must be told where the string is. You do this by loading the **X** register with the starting address where the string is stored before calling the function. To enter strings in memory, we use FCC directive. The label associated with the FCC will be automatically `EQU`ated to the starting address of the string. Type the following code and verify that `OUTSTRG` does indeed print a string.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;
OUTSTRG     EQU $FFC7
OUTSTRGO    EQU $FFCA

; program section. set origin to $C100
        ORG $C100
        LDX #ABOUTME    *STARTING ADDRESS OF THE STRING
        JSR OUTSTRG
        SWI

; data section. set origin to $D000
        ORG $D000
ABOUTME FCC /Hello, my name is ===your name =====/
        FCB 4 ;dont forget this
;;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;
```

4 Writing your first function

Before you write your first function, you should observe some standard conventions. Your programs should always start at location $C100, or as specified by

your TA. Your data should always start at location $D000 or as specified by your TA. A small amount of data can be stored starting at location $0000 (Page 0), but you do not have too much space, 30 bytes or so. It is a matter of taste whether you write the data section first or the program section first. You can not mix and match. I personally prefer the following order: Page 0 section first, data section next and then the program section, though it is easier to follow the code if the program section precedes the data section as in the previous example.

The program section should start with the main code, i.e. the code you want to execute. The main code should be followed by various functions. The order is not important.

What is a function? A function, also known as a *subroutine* is a *self contained* code that implements a well defined functionality. What do I mean by self contained? You should be able to draw a line above and below your code for the function, and make sure that

1. Only way to branch **out of** the two lines is with `JSR` or `BSR` or `RTS` instructions. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect

2. Only way to branch **into** an instruction between the two lines from an instruction outside the two lines is with a `JSR` or `BSR` instruction. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect.

The function should terminate with a return from subroutine `RTS` instruction. It is a good programming practice not to have more than one `RTS` statement in any function.

Once you have written your function, it is there for you to use as many times as you need. To use the function, you should know where its first instruction is located in memory (technically known as the entry point). If you use the assembler to create the function, you can place a label before the very first instruction. The assembler will automatically `EQU`ate the label with the first instruction. If the function needs any additional information, they will have to be supplied by the user prior to using the function (technically known as binding). The function you will be writing will use one of the registers for binding. Also, many functions will return some useful value to the caller. In this case, it is a good idea to return the value in one of the registers.

Unless you write functions that do absolutely nothing (technically known as stubs), the function will use and modify one or more registers. If the caller happens to keep valuable data in one of these registers, then you have a potential problem. There are two possible solutions: The caller could save the values in the registers before calling your function, and then restore it after your function

returns. Alternatively, your function can save the values in the registers it uses and then restore the values before it returns. The first approach more efficient but the second approach will result in fewer bugs. I strongly recommend that you get into the habit of writing functions that clean up after themselves and restore the registers the way they were before the functions used them[1].

Here are the basic rules for writing functions

1. Decide on its functionality. Don't try to create a Swiss army knife that has multiple functionalities built in. Your function must do only one thing, and it must do it well. Your documentation for the function must clearly state the functionality

2. Decide on its name. Pick a meaningful name but keep the name to 8 characters or less

3. Decide on the registers that will be used to pass information **to** the function. 8-bit values can be sent using **A** or **B** registers. 16-bit values can be sent using **X** or **Y** registers[2].

4. Decide what registers will be used to **return** values back to the caller. 8-bit values can be returned using **A** or **B** registers. 16-bit values can be returned using **X** or **Y** registers.

5. Decide what registers the function will use and which of these will be restored back at the end. Clearly document which registers will be used and **not** restored back as the registers that are modified by the function.

6. Write the function. Avoid the temptation of writing the function first and then worrying about the other items!

As an example, we will write a simple function called `RAND` that will return a random value every time it is called. How does this function work? The function starts with a seed. We use an 8-bit number as a seed. The seed is used to calculate a random number using some formula. To make sure we get a different number every time the function is called, the seed value is changed. Typically, the random number that is generated is used as the seed for the next random number. Thus, we are actually generating a random sequence starting with some initial seed. The formula[3] it uses is simple: It shifts the seed value left, and adds with carry the

[1]The only exception is the register that is used to return a value to the user. Clearly, these registers should not be restored to their original value!

[2]For example, `OUTA` expects the data in the**A** register, while `OUT1BYT` expects the information in the **X** register.

[3]I use this as a quick and dirty 8 bit random number generator. It is not the best, but the code is only 5 line long!

value 20. The function needs one byte of storage to keep track of the seed. This storage will be allocated in the data section using the `FCB` directive as

```
SEED     FCB     0
```

You use FCB to initialize consecutive memory loication (when you transfer the code from the PC). The label associated with the instruction is needed to determine the address where the instruction forms the constant. The assembler will automatically `EQU`ate the label with the address associated with the `FCB` directive.

Type the following code, assemble it, transfer the S19 file to the HC11, and test your program by `CALL $C100`. Repeat[4] the `CALL` statement and verify that the value in the **A** changes after each call.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;;
    ORG $C100
    JSR RAND
    SWI


;;;;;;;;;;;;;;;;;;;;;;;;Start: RAND ;;;;;;;;;;;;;;;;;;
; Function: RAND
; Purpose: Generate a random number
;
; Inputs: None
; Outputs: A random value returned in A registers
;
; Registers modified: A register (which has a random value)
;
; Memory usage: The most recently generated random number is
;               stored in memory with label LSTRAND
;               This value is used to generate the next value
;
; Notes: Not the best random number generator around but does a
;        halfway decent job.
;
;        works as follows:
;        shift the last random value left and add 20 with carry
;
;
```

[4]In BUFFALO, if you press the enter key at the prompt, BUFFALO will repeat the last instruction. So you don't have to type the CALL every time. Just press the enter key.

```
RAND        LDAA    SEED
        LSLA
        ADCA    #20
        STAA    SEED ;don't forget to save it back
        RTS
;;;;;;;;;;;;;;;;;;;;;;;;End: RAND ;;;;;;;;;;;;;;;;;;;;

;
; DATA SECTION
;

    ORG $D000
SEED     FCB     0

;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;;
```

4.1 On random sequences

When you are done running the program a few times, you reload the s19 file and try again. *You will get the same sequence all over again!* Now this is useful when you are debugging programs but is absolutely useless if you are writing a game program. *Every game will be 100% predictable!* The way to overcome this is to change the seed everytime you start your program in some unpredictable way. One simple solution is to use the low order byte of the clock inside the HC11. This clock is in locations `$100E-$100F`. So you can use the second location `$100F` as the initial seed.

5 Your second function

For your second function, you will be writing a function that will print an 8-bit value in HEX first and then in binary. We will call this function `PRBINARY`

An 8-bit number requires 2 hex-digits to print and BUFFALO has two routines to help you, one to print the left digit and the other to print the right digit. These are called `OUTLHLF` for out-left-half and `OUTRHLF` for out-right-half respectively.

To print it in binary, we will use the shift left instruction `LSL` . This instruction will shift all bits left by 1 place and the (left most) bit that is shifted out will be stored in the carry flag. I.e. if an 8-bit value before shifting was `abcdefgh` then

the value after shifting will be `bcdefgh0`. Here each of the letters `a` to `h` represent bits. The carry flag will be set to `a`. Thus the value to be printed will be shifted left 8 times. After each shift, the A register will be loaded with either the ascii code for `0` or the code for `1` depending on whether the carry is cleared or set[5]. Note the function involves a counting loop.

We now have to decide register usage: We will use the **A** register to pass the value to the function. Internally, this value will be moved to **B** as **A** is needed in all the calls to BUFFALO routines. We need a counter, and we will use the **X** register to keep count. As a good programming practice, we will store and restore all registers we will use.

Here is the complete code for the function.

```
;;;;;;;;;;;;;;;;;;;;;;Start: PRBINARY;;;;;;;;;;;;;;;;;;;;
; Function: PRBINARY
; Purpose: To print a value in binary
;
; Inputs: Value to be printed is passed in the A register
;
; Returns: None
;
; Registers affected: None. The values in the registers are stored
;             first and these values are restored at the end.
;
; Notes: The output consists of two parts. The value in A is first
;        printed in HEX, and then in binary
;

PRBINARY

; first save the registers we will be using: a, b and x
    PSHA
    PSHB
    PSHX

; copy a to be for later use

       TAB

; print a as hex number (2 digits).
```

[5]Conveniently, the code for `1` is one more than the code for `0`. So we load **A** with the code for `0` and add the carry to it

```
; print the left digit
    JSR OUTLHLF

; the function destroys the value in a,
; so re load it! then print second digit
    TBA
    JSR OUTRHLF

; now print a colon and some spaces
    LDAA #':'
    JSR OUTA

    LDAA #' '
    JSR OUTA
    JSR OUTA
    JSR OUTA

; now print it in binary
; b has the value to be printed (recall the old tab)
;
; shift b  to the left by one bit and print '0' or '1' depending
; on what is in the carry flag
; repeat 8 times.
;
; we will use X register as counter
; to print what is in carry flag, we will load A
; with the code for '0' ; and add the carry to the code
; prior to calling OUTA

    LDX #8 *COUNTER

PRBLOOP CPX #0
    BEQ PRBDONE

    LDAA #'0'
    LSLB
    ADCA #0
    JSR OUTA
    DEX
    BRA PRBLOOP
```

```
PRBDONE
    JSR OUTCRLF

; restore the registers
    PULX
    PULB
    PULA
    RTS

;;;;;;;;;;;;;;;;;;;;;;;End: PRBINARY;;;;;;;;;;;;;;;;;;;
```

5.1 Test your function

We can test the function by loading different values in the A register. The random number generator we wrote first comes in useful here! Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;;
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

UCASE       EQU $FFA0
WCHEK       EQU $FFA3
DCHEK       EQU $FFA6
INIT        EQU $FFA9
INPUT       EQU $FFAC
OUTPUT      EQU $FFAF
OUTLHLF     EQU $FFB2
OUTRHLF     EQU $FFB5
OUTA        EQU $FFB8
OUT1BYT     EQU $FFBB
OUT1BSP     EQU $FFBE
OUT2BSP     EQU $FFC1
OUTCRLF     EQU $FFC4
OUTSTRG     EQU $FFC7
OUTSTRGO    EQU $FFCA
INCHAR      EQU $FFCD
VECINIT     EQU $FFD0
```

```
    ORG $C100
    LDX  #ABOUTME
    JSR  OUTSTRG
    JSR  RAND
    JSR  PRBINARY
    SWI
;;; INSERT YOUR CODE FOR PRBINARY HERE
;;; INSERT YOUR CODE FOR RAND HERE

    ORG $D000
;;; INSERT ALL YOUR DATA (FCB, FCC, RMB etc.) here
ABOUTME FCC / INFORMATION ABOUT YOU /
    FCB 4
SEED     FCB     0

;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;
```

6 Setting bits

We will now write a function that will set a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will set bit #4 in memory location $00. Recall that the bits are numbered right to left starting with bit #0. To set a bit, we use the `ORA` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here

    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00
    JSR     PRBINARY *PRINT BEFORE
    JSR     SETBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER
    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: SETBIT4
```

```
; Purpose: SETS bit #4 in memory location $00
; Registers modified none
;

SETBIT4
    PSHA

    LDAA $00
    ORAA #%00010000
    STAA $00

    PULA
    RTS


;; INSERT THE CODE FOR FUNCTIONS RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.
;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;;
```

7 Clearing bits

We will now write a function that will clear a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will clear bit #4 in memory location $00. To clear a bit, we use the `AND` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here

    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    JSR     SETBIT4
```

```
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER SET

    JSR     CLRBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER CLEAR

    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: CLRBIT4
; Purpose: Clears bit #4 in memory location $00
; Registers modified none
;

CLRBIT4
    PSHA

    LDAA $00
    ANDA #%11101111
    STAA $00

    PULA
    RTS


;; INSERT THE CODE FOR FUNCTIONS SETBIT4 RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;;
```

8 Toggling bits

We will now write a function that will toggle (make it zero if it was a one; make it one if it was a zero) a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will toggle bit #4 in memory location $00. To toggle a bit, we use the `eor` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO

prompt.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here

    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 1 TOGGLE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 2 TOGGLE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 3 TOGGLES

    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: TGLBIT4
; Purpose: tOGGLES bit #4 in memory location $00
; Registers modified none
;

TGLBIT4
    PSHA

    LDAA $00
    EORA #%00010000
    STAA $00

    PULA
    RTS
```

```
;; INSERT THE CODE FOR  CLRBIT4 SETBIT4 RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
;;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;;
```

9 Testing bits

Often we have to take a decision based on whether a bit is set or not in memory. To test if one or more bits are set, we clear all other bits and see if the result is zero. If so, none of these bits are set. If not, at least one of them was set. The following program will print `YES` if bit #4 in memory location $00 is set. Or else it will print `NO`.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here
BIT4 EQU %00010000 ; THIS EQUATE MAKES THE CODE MORE READABLE
    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    LDAA    $00
    ANDA    #BIT4
    BEQ     NOPE
    LDX     #YESSTR
    JSR     OUTSTRG
    SWI

NOPE
    LDX     #NOSTR
    JSR     OUTSTRG
    SWI


;; INSERT THE CODE FOR  CLRBIT4 SETBIT4 RAND AND PRBINARY HERE
```

```
    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
YESSTR  FCC /YES/
        FCB 4
NOSTR   FCC /NO/
        FCB 4
;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;;
```

10 Hardware Interfacing

Working with random numbers is fine, but we want to do something useful. In HC11, there area special memory locations called `PORTS`. The special nature of these locations allows us to have direct access to the individual bits using external circuitry. Each bit in the port has an I/O line associated with it. This line provides access to the bit. The bits in the port can be one of two types:

1. A bit in a port could be an **input** bit. If this is the case, then we can only set or clear the bit using external electrical circuit connected to the I/O line associated with the bit. *This means that the code you wrote earlier to set or clear a bit will have no effect on the bit.* You can however check the bit and take appropriate action. To set the bit, you have to set the voltage of the I/O line above 3 volts (without exceeding the supply voltage of 5 volts). To clear the bit, you have to set the voltage of the bit to below 2 volts (without going below zero).

2. A particular bit can be an **output** bit. If this is the case, we can set or clear the bit in our program and the bit will control the I/O line associated with the bit. If the bit get set, then the voltage on the line will go to 5 volts. If the bit is cleared, the voltage on the line will go to zero. **Make absolutely sure that you do not connect any external device that can control the voltage (such as a voltage source) to the line.** The single biggest reason why HC11 ports get burnt is when some external device tries to send the voltage on the line to zero while your program tries to send it to 5 volts or vice versa. **If you are concerned about damaging the port. always connect 2.2K or larger resistor in series with the port. This will limit the port current to 1 mA or less.**

10.1 `PORTA` at location `$1000`

In this experiment we shall work with `PORTA` which is at memory location `$1000`. The bits of `PORTA` are designated as `PA7, PA6, PA5,` ··· `PA0`. If you are using

the FOX11 board, the lines associated with these ports are clearly labelled. If you are using CMD11E1 from Axiom, the lines are the first 8 pins in the MCU CONNECTOR. The bits PA0, PA1, PA2 are inputs. This means you can connect external circuits to these pins. The bits PA3, PA4, PA5, PA6 are outputs. This means you can drive devices from these pins (as long as you do not supply more than 5 mA). PA7 is bidirectional and you, as the programmer, can configure the pin as either input or output.

WARNING: You may have wired the port as input and either by accident or oversight, may set the pin as an output pin. This can seriously damage the pin if the pin carries currents in excess of 5 mA or so. To prevent the damage, make sure there is a current limiting resistor (4.7K) in series with the pin. This warning is applicable to any pin that is bidirectional.

WARNING: If you lend your HC11 to anyone, make sure you disconnect any circuit that may be connected to PA7.

Special instructions for CMD11E1 users

1. Locate the jumper JP13 and make sure that it is open.
2. Locate the MCU port. This is a dual row 34-pin Berg style connector. Locate pin #1 on the port. This is identified with the number 1 on the front of the board. On the other side (solder side), pin #1 is identified by a square solder.
3. Connect a ribbon cable to the port. Use a continuity tester to locate and identify the following pins: pin 1 (PA0), pin 5 (PA4), pin 6 (PA5), pins 9 and 10 (5 Volts), pins 11 and 12 (Ground). (See page 15 of the User's guide that came with your board.)
4. Make two test light emitting diode probes. It is a good idea to have several probes handy. To make a test probe, connect a 3.3K resistor to the **anode** of a light emitting diode (See figure 1). Use a light emitting diode with an operating voltage of 1.7 volts and a current rating in the 1-5 mA range. If you use an light emitting diode with 20+ mA operating current, the light emitting diode would be dim when it lights up and you may have to look carefully to see if it is on. Test the probe by connecting the free end of the resistor to pin 9 and the free end of the light emitting diode to pin 11. The light emitting diode should light-up. If not, the chances are you have connected the resistor to the cathode of the light emitting diode. Redo your circuit by connecting the resistor to the anode.

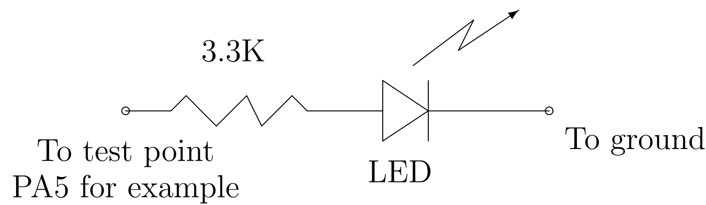

Figure 1: *Test Probe. Make sure that the resistor is connected to the anode of the light emitting diode. The longer lead of the light emitting diode is the anode. To test a pin, connect the cathode to ground and the resistor to the pin.*

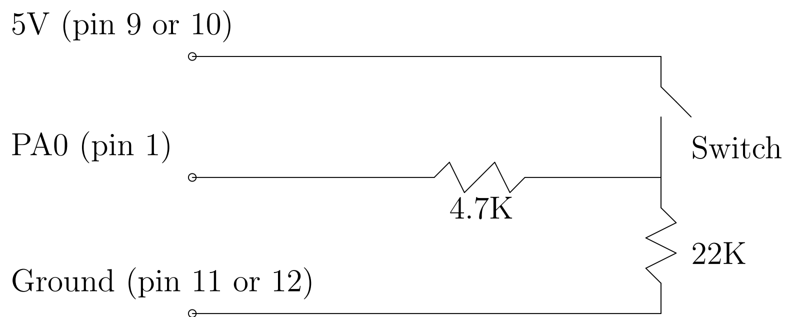

Figure 2: *Typical input connection*

5. Connect a (LEFT) test probe between pin 5 and pin 11. The cathode of the light emitting diode should be connected to pin 11 and the free end of the resistor to pin 5.
6. Connect a (RIGHT) test probe between pin 6 and pin 12. The cathode of the light emitting diode should be connected to pin 12 and the free end of the resistor to pin 6.
7. Label the two light emitting diodes as LEFT and RIGHT so that it is easy to identify them (you can use a magic tape and small pieces of paper).
8. Connect the input circuit as shown in figure 2.
9. Check and recheck all your connections before you connect the power to the HC11.

Special instructions for FOX11 users

1. Connect two LED circuits shown in figure 1 to PA4 and PA5. Label the circuit connected to PA4 LEFT and the one connected to PA5 as RIGHT.

2. Connect the input circuit as shown in figure 2.

10.2 Controlling the LED

Use memory modify, MM $1000, command to modify PORTA. Change the value in the location to 00, 10, 20, and 30. (Note: When you communicate directly with the HC11 using BUFFALO commands, you do not type the $.) After each change, look at the state of the light emitting diodes and write down what you see. Provide a brief explanation of what you see.

Using the methods you learnt in earlier labs, write a program that will read the keyboard and depending on what the user types, perform the following:

Key	Action
Q or q	Turn on the LEFT led. The state of other led should not change.
Z or Z	Turn off the LEFT led. The state of other led should not change.
E or e	Turn on the RIGHT led. The state of other led should not change.
C or c	Turn off the RIGHT led. The state of other led should not change.

10.3 Reading an external switch

Use memory dump to see the contents of $1000. Close the input switch and dump the contents of location $1000. Repeat the experiment with the switch open. Write down what you see and provide a brief explanation of your observation.

Write a program that will, in an infinite loop, print either CLOSED or OPEN, depending on the state of the switch. The program should continually monitor the state of the switch, and print CLOSED is the switch is closed. Or else it should print OPEN.