This laboratory will require you to connect a hexadecimal keypad, organized as a matrix of momentary (SPST) switches in four columns and four rows. The switches in each column have one leg connected to the column wire. The switches in each row have the other leg connected to the row wire. Thus, pressing one of the buttons on the keypad will cause one column wire to short across to one row wire. This shorting determines uniquely the switch that is closed (note that, if you press multiple switches, you cannot in general determine which switches are closed.)

If you just had sixteen switches you could just connect them to 16 digital I/O pins. Arranging them as in the keypad uses only 8 pins. The relationship is an inverse square, so that, as the number of keys goes up, the advantage of multiplexing becomes overwhelming.

The seven-segment displays operate by each segment being lit by an LED. Connect pin number three (refer to the pin diagram) to VCC and then connect one of the active pins to ground, through a current-limiting resistor, and one of the segments will light up. There are actually 8 LEDs, as the decimal point is also available.

To display a digit on the display, you need to drive the segments either high or low. To display the numeral '8', for example, you would drive them all low. To display the numeral '1', you would drive only segments B and C low.

The problem here is that, if you want to drive each display independently, it will require 8 digital I/O lines per digit. This is an insurmountable problem for a two-digit display, since it would require 16 I/O lines (and you only have a total of 21) – but what if you wanted a four-digit display with any additional I/O? The solution to this problem is to **multiplex** the signals driving the display digits.

Multiplexing in this context takes advantage of the response of the human eye to transient lights. They appear to last longer than the actual flash, if the light is reasonably bright. (Those of you interested in bio-engineering can look up the "integrating operation of retinal reception".)

We take advantage of this phenomena by switching each digit on for half the time, but doing this so quickly that the human eye is fooled into thinking the digit is on (at half its normal brightness) all the time. Since each digit is on only half the time (for a four-digit display each digit would be on only 1/4 of the time) we can drive each display with different data **using the same output signals.** The cost is an additional I/O signal (to turn on the "addressed" digit) per digit.

For two digits, this will require a total of 10 bits (as opposed to 16 if we drove each display independently). For four digits this would require 12 bits of I/O (as opposed to 32.)

For this lab you will need to perform certain operations each millisecond. That means that most of the time your program is going to be looping, waiting for the proper time to perform an action.

You need to drive the digital display as follows. You'll need some variables, one (d) to hold the digit being displayed (for two digits this should be a 1-bit variable), and two 8-bit values (unsigned char digit[2]) which hold the patterns

to be displayed. Then,

1. Turn off both digits (Port B bits 10 and 11 to 0).
2. Drive the output signals to the individual segments (Port B pins 2-9) to the values (digits[d]) for the proper digit (even or odd, depending on the variable d.
3. Turn on the digit. We are using NPN transistors to turn them on, so raise the digital I/O signal connected to the base of the transistor connected to the even digit anode (Port B, bit 11 if (d is 1) Port B, bit 10 otherwise.)
4. Change d. If d = 1, make it 0. If d = 0, make it 1.
5. Wait 1 millisecond and repeat the entire sequence.

To scan the keypad, you would also want to do it no more often than necessary. Scanning too fast just means you would have to debounce more, but scanning too slow means you'll miss keypresses. You'll need some variables to hold the row being scanned, and you'll need to have a record of what the last value you read from each row was (so you can tell when it changes.) Here is how you scan the keypad:

1. At initialization, set the row to 0, and set the previous values ("lastval[0-4]") for each row to 0x0F – I'm assuming you'll use RB12 - RB15 as outputs, RA0 - RA3 as outputs. You'll also need a variable called "debounce" – initialize it to 0.
2. Turn on the port A pullups. Initialize Port B 15-12 to 0b1111, set PORTA bits 0 - 3 to input, PORTB 15 - 12 to output.
3. Write a 0 to the output bit (in RB15 - RB12) that corresponds to the row you want to scan.
4. Wait 1 millisecond.
5. Read PORTA. If the value read from port A, RA0 - RA3, is any different from the previous value, and if debounce is 0, then you know the keypad has changed since you last read it.
6. If the keypad status has changed, update the value (lastval[row]) to the new value, and also set debounce to a proper value.
7. You may also want to do something with the value.
8. increment row, rolling it over to 0 if it goes to 4.
9. repeat from step 3 above.

I like to use tabular data to hold values and I like to use the logical and operation (it's fast and pretty hard to screw up.) Observe:

```
static unsigned char scanValues[] = { 0x8000, 0x4000, 0x2000, 0x1000};
                  // values to put out on B 15-12
```

and here's another couple of tricks:

```
row = ++row & 3; // increment row, truncate
            // (makes row go 0, 1, 2, 3, 0, 1, 2, 3, ...)
d = 1 - d;  // if d = 0 initially, then d goes 0, 1, 0, ...
```

In this laboratory you are going to build a nearly-trivial application which will accept characters from the keypad and display them, two at a time, on the display. To reduce the number of digital I/O pins required, the keypad will be driven in a multiplexed manner, and so will the displays.

- The keypad will be connected to Port B 12-15 Port A 0 - 3.
- The seven-segment display segment lines (a-f and dp) will be connected in parallel (a1 and a2, b1 and b2, for example, not a1 and b2) to port B, pins 2 - 9 as shown in the diagram.
- The enabling signal for the display digits are connected to port B, pins 10 and 11.
- Operation of the application will be to initially have all segments turned off.
- The keyboard will be scanned at 1 millisecond per row (or column, if you scan columnwise) so that the entire keypad will be scanned every four milliseconds.
- If a key is pressed, the value of the key will be displayed on the least-significant (rightmost) of the display digits. If another key is depressed, the previous key will be shifted to the most-significant (leftmost) of the digit of the display and the new key will be displayed on the least-significant digit. This should continue indefinitely. Only the last two keys pressed will be kept and displayed.
- The display multiplexing rate will be 1 milliseconds per digit, or 2 milliseconds for each display cycle.
- You should write your display routines as functions, so that you can re-use them in the next few labs.

Note: We've used different keypads for this lab over the years. The ones we use now may have a different pinout than shown in the diagram. Also, if it bothers you that the diagram shows rows running up and down and columns

running horizontally, ..., turn the diagram 90 degrees counterclockwise. It won't affect the operation of the keypad, but it may make you feel better.

To get credit for this laboratory, display your application for the TA. Let him (or her) press the keypad buttons and observe that the device works correctly. Also turn in the answers to the following questions:

1. Write an assembly[1], pseudocode or C implementation of your program (if you didn't write in C to begin with – and in this case, why didn't you?)

2. Did you use tabular data in your implementation? Why or why not?

3. Calculate the number of digital I/O pins required for a 48-key keyboard, a 104-key keyboard, and a 120-key keyboard (I saw one offered the other day), assuming a multiplexed interface. Make the assumption that, in each case, the number of rows and columns will be about the same (so a 120-key keyboard would have 11 rows and 11 columns.)

We're going to use the hexadecimal display again. Don't disassemble it (unless you really enjoy rewiring it.)

Look for the lab circuit on the moodle website. It's a bit confusing – I had to make all the wires fit. But basically the wiring is as described in the text above.

[1]From now on we will no longer require you to use a particular programming language. If you choose to write everything in assembly language, we'll merely wonder about your sanity.