



COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

©Copyright: 2007 by Bryan Mealy



CPE 269 Experiment 9



BCD TO SEVEN-SEGMENT DECODING AND DISPLAY MULTIPLEXING

Objectives:

- To learn various strategies required to implement a firmware-based seven-segment decoder
- To learn the concepts of multiplexing seven-segment displays
- To implement an algorithm on PicoBlaze to drive two digits on the development board
- To learn the basic concepts of flowcharting
- To learn the basic concepts of modular coding

Somewhat Meaningful Comments:

Picture this scenario... Someone in the business department of the company you work for got wind of that fact that the PicoBlaze microcontroller in one of your company's products was being under utilized. Over an ensuing conversation at the company watering hole, that person in the business department convinced the engineering manager to remove the discrete BCD to seven-segment decoder and display multiplexer from the PC board. Lowering the cost of the BOM (bill of materials) has a way of exciting most people though it seems that it's the engineers who pay for all those "minor" circuit changes. The thought was that the PicoBlaze microcontroller could be used to implement the same functionality. The engineering manager thought it was a great idea and it was decided that it would be done. And since you're the only engineer who knows anything about PicoBlaze, it's your job to make it work...

Display multiplexing refers to the general practice of only using one seven-segment decoder to drive a set of several BCD inputs to several seven-segment displays. The approach is to actuate only one display at a time for a certain period of time. Each of the individual seven-segment displays is actuated sequentially in a circular manner thus ensuring each display will be activated for an equal amount of time. This multiplexing action takes advantage of the *retinal persistence* characteristic present in the human visual system to make it seem like all the displays are activated all of the time.

The same approach must be taken with the Nexys development board. There are four seven-segment displays. The activation of each of these displays is controlled by individual transistors associated with each display. These transistors are wired as switches and are activated with the voltage represented by a '0' (active low). Each of the seven-segment displays are wired together so writing to one individual segment of a display is actually writing to that segment of all four seven-segment displays on the board. Whether any particular LED is activated or not is decided by the state of the transistor associated with that seven-segment display.

To make the displays appear as if they are constantly on without the appearance of flicker, you need to leave each display actuated for a given amount of time. This is accomplished with a firmware delay function such as the one shown in Figure 1. The idea is to do no further processing for a set period of time after a display has been actuated before going on to actuate the next display. The firmware delay listed in Figure 1 is not an efficient use of the microcontroller's resources since the program execution is in a tight loop that effectively does no processing. It is, however, a viable firmware-based approach to providing a time delay.

```

-----
;- Subroutine: delay
;- Description: creates a delay by iterating loop
;-
;- Registers Used: s7
-----
delay:
    LOAD    s7,FF           ; preset loop count
delay_loop:
    SUB     s7,01           ; decrement
    JUMP    NZ,delay_loop   ; loop if s7 != 0
    RETURN
-----

```

Figure 1: Assembly code for firmware delay.

Modular Coding Techniques

The time delay shown in Figure 1 is part of the firmware approach to multiplexing the displays. In this experiment, you'll need to multiplex between two displays. Figure 2 shows a useful multiplexing algorithm.

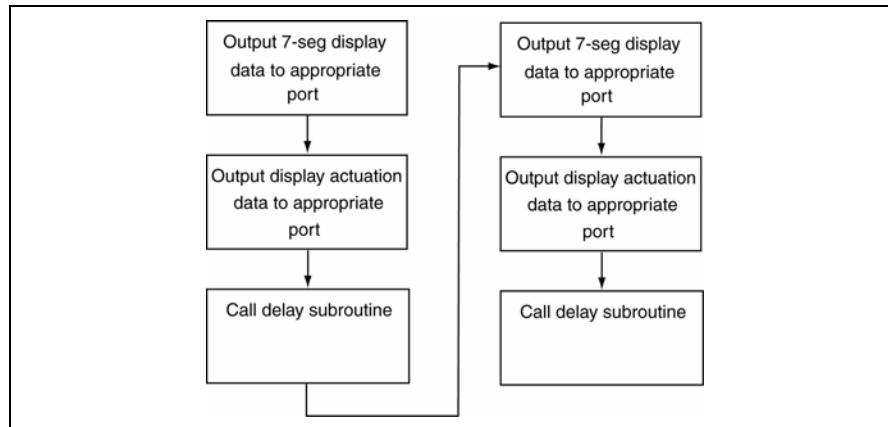


Figure 2: Process flow for firmware multiplexing algorithm.

As you can imagine, the source code to implement this algorithm can easily be written as one continuous section of code. Writing the source code in this manner makes that section of source code specific to driving two seven segment displays. A better approach would be to write your code in such a manner as to be able to reuse portions of the code. In this way, you could quickly modify your code to drive three seven-segment displays instead of two, if this was required. This technique is referred to as a modular approach and will save you time and effort in this and the following experiments as well as life in general. The approach is outlined in Figure 3. The algorithms shown in Figure 3 perform the identical functions as the algorithm shown in Figure 2 except the original functionality is separated into two modules.

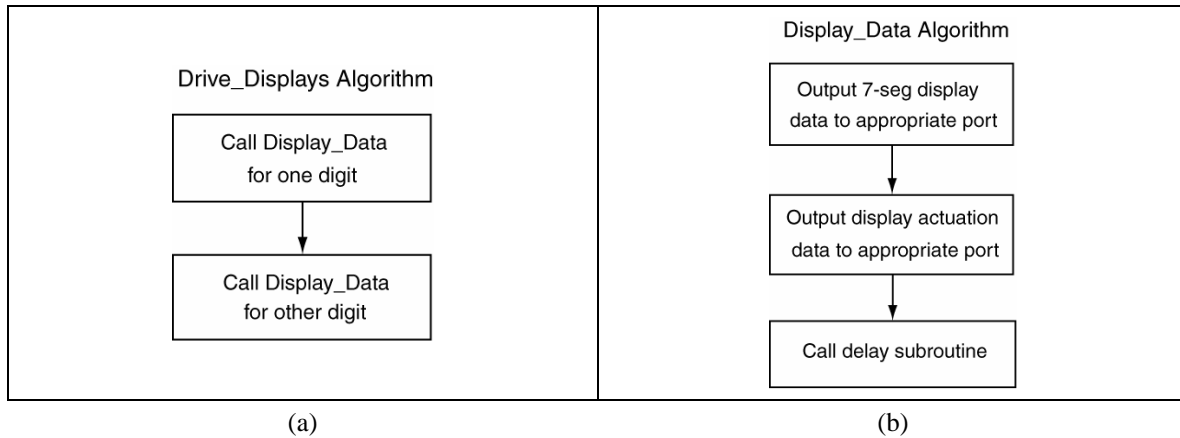


Figure 3: Flowcharts for algorithms that modularize the algorithm shown in Figure 2.

The key to making this modular approach work is to take a generic approach to writing the algorithms shown in Figure 3. This code that implements the algorithm in Figure 3(b) should expect to find data to drive the segments and data to actuate a single display in two different registers. The algorithm in Figure 3(a) should place the proper data in those registers before the algorithm in Figure 3(b) is called.

Programming Assignment:

Write an assembly language program that will continuously repeat the following steps:

- Read the status of the eight slide switches on the Nexys board. A switch in the *up* position is interpreted as a “1”; in the *down* position the switch is interpreted as “0”.
- Display the data represented by the switch positions as follows:
 - The four upper and four lower switches (upper and lower nibbles) are interpreted as two BCD numbers, respectively.
 - Use the data on the lower nibble to drive the decimal equivalent value on the right-most seven-segment display.
 - Use the data on the upper nibble to drive its decimal equivalent value to the second-to-right-most display.

NOTE: If an invalid BCD number (numbers 10-15) appears on an input, the seven-segment display will disable all display segments for its respective display. Also, the segments on the seven-segment displays on the Nexys board are active-low.

Hint: Most of the work in this experiment involves the creating the PicoBlaze code that decodes a BCD number. The main section of code should then be an endless loop that continuously monitors the switches and calls the subroutines that displays the data represented by those switches. All major functions should be placed in subroutines. The main body of your code should primarily be comprised of a series of subroutine calls. The best approach before partaking on any programming project is to properly engineer your firmware before you start coding. The high-level understanding provided by this approach helps you to understand the system and bypass problems generated by poorly designed firmware. But whatever you do, do not include two separate pieces of code for decoding a BCD number: you must practice code reuse in this experiment.

Deliverables:

1. Demonstrate your working circuit to the lab instructor or Teaching Assistant.
2. Provide a listing of your assembly source code with your lab write up. Make sure you take a look at the PicoBlaze assembly language style file before you submit your source code.
3. Answers to the questions below.

Questions:

1. Assume the system clock on the Nexys board runs at 50MHz. Using this value and your knowledge of the PicoBlaze architecture, calculate the delay time (in seconds) provided by the delay subroutine shown in Figure 1. State your assumptions and show your work for this calculation.
2. Provide some PicoBlaze assembly code that would allow you to generate as close to an 18 μ s delay as possible. State how far off from 18 μ s your solution is. State your assumptions and show your work for this calculation.
3. Most microcontrollers have instructions that officially do nothing; they are referred to as “nops”¹. PicoBlaze does not have an official nop instruction but instead must use existing instructions to officially do nothing. List as many PicoBlaze “nops” as possible.
4. Although PicoBlaze does not have one, many processors and microcontrollers have a flag bit know as an *auxiliary carry* or *half carry*. This bit indicates whether there is a carry from the fourth bit to the fifth bit. Briefly describe this bit’s purpose.
5. Using what you know about video and film, speculate on the frequency where the display multiplexing rate becomes noticeable. In other words, at what frequency does the display start to have a noticeable flicker?
6. PicoBlaze is an 8-bit processor which roughly means that width of the databus is eight bits. Write a fragment of PicoBlaze code that considers registers sA and sB as one 16-bit register and performs a 16-bit rotate right.

¹ Nerd terminology alert: people or professors who seem to slide by doing nothing or getting someone else to do their work for them are often referred to as “nops” (pronounced: *know-ops*). We all know who these people are.