# **Microprocessors (662-133) NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Lab #6- Lookup tables and Arrays.

Description

This lab will introduce the student to the methods of programming used with looping and branching instructions and will be explained and practiced in the lab.

Learning objectives

1. Create the proper file structure needed by Eclipse.
2. Download an assembly language program into the processor.
3. Use the Nios II debugger to edit and view the program.
4. Use the stepping functions to debug a program.
5. Insert a “break” point to help debug a program.
6. Use pseudo codes to help develop a program.
7. Use the branching instructions to make program decisions.
8. Use branching instructions to create loops.
9. Use the “for” loop to execute a loop a set number of times.
10. Use shift commands to “bit pack” bytes into a 32 bit word.
11. Create a lookup table to be used as a seven segment display decoder.
12. Load an array into memory.

Procedure

Set up a project as you have done in the past. Call this one lab6. Create a source file lab6.s and copy the contents of the provided lab6.s

For this program, we will be calculating the average of an array, and then displaying the result on the 7-segment display.

The existing shell contains the following:

1. Two data structures, an array of 10 values, and a lookup table for the 7-segment conversion
2. Code to extract the 1000's, 100's, 10's and 1's place from a given register

What you need to do:

Write code to calculate the average of the array. Use a summing for loop, and then use the div instruction. Load the number of elements in the array (10 by default) into a single register once, or use a single .EQU expression. That way you can easily modify the program to handle different length arrays. Be sure to store the result (the average) in r10.

Code is provided that splits a single number in r10 into a 1000's place, 100's place, 10's place, and 1's place numbers, in r12-r15. This code also uses r9 and overwrites the value in r10, so *don't* count on any of those registers being the same after that section of code runs!

Then you will need to use the lookup table to convert each digit of the average into a binary value suitable for the 7-segment display. To save registers, you can simply load the binary values back into r12-r15, since those numbers won't be needed again. Remember to use ldbu so you only load 8 bits at a time, and since these values are not numbers, only particular binary patterns, we use the 'u' version.

Lastly, you must combine the 4 byte values you found in the lookup table for the 7-segment displays into a single 32-bit value that can be written to the register located at the HEX\_BASE address (0x10000040). Note: The highest value (the 1000's place) should be in the upper bits (bits 32-25), while the lowest value (the 1's place) should be in bits 8 through 1.

Bit packing

Combining smaller values into larger variables is known as bit packing. For instance, if you had a large number of 4-bit values, you might put them into bytes as pairs, so as not to waste half the available memory. In this lab you will need to pack four 8-bit values into a single 32-bit value for the 7-segment displays.

For instance, given a 16-bit I/O port, if we wanted to output four 4-bit values, (call them A, B, C, and D) First we must determine which order we would like them in, in this case the desired output 16-bit value is:

AAAABBBBCCCCDDDD

So given four input values, where the upper bits are 0's:

000000000000AAAA (in r5)

000000000000BBBB (in r6)

000000000000CCCC (in r7)

000000000000DDDD (in r8)

Note: registers are really 32-bits, of course, we're ignoring the other 16-bits of zeros for now, especially since our output register is only going to be 16-bit anyway.

It's clear that we need to shift some of the bits to the left so they will be in the correct position. We can do this using the slli instruction, which shifts the bits in a register to the left a set number of positions, filling in 0's on the right. So if we used it on the four registers containing the above values, we would get:

AAAA000000000000 (after calling slli r5,r5,12)

0000BBBB00000000 (after calling slli r6,r6,8)

00000000CCCC0000 (after calling slli r7,r7,4)

000000000000DDDD (after calling slli r8,r8,0 ...or do nothing)

The only trick left is to combine the values into a single register, which can be done easily. Since all the bits we don't care about in each register are zero, we can simply OR all these values together to get the desired result. Start with the zero register and or everything together:

or r9,r0,r5 (Could also just mov r9,r5)

or r9,r9,r6

or r9,r9,r7

or r9,r9,r8

r9 now contains: AAAABBBBCCCCDDDD