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

Lab #10 – Arrays and Buffers

Description

This lab will detail how to create a complete C program in the Altera NIOS EDS. We will take the program written in assembly language in lab 6 and rewrite it using C.

Learning objectives

1. Create the proper file structure needed by Eclipse.
2. Use expanded debugging techniques available in C.
3. Create a C program.
4. Use the expression window to debug and evaluate program operation.
5. Observe function calls.
6. Use the disassembly to view the assembly program created by the complier.
7. Use C to construct and array.
8. Use C to construct a ring buffer.
9. Use C to access the seven segment displays.

**Procedure**

We will be using arrays and loops to do some of the same things we did in lab 6 (with the averages and seven segment display) but with a few twists:

Instead of displaying the average of a few numbers in a fixed array, we will display the average of the last 10 numbers entered on the switches. This can be done using something known as a ring buffer. That is, an array that holds the most recent data points we care about. This can be done by moving the point we insert data each time we insert a number.

For instance, the first 10 numbers read in will be stored in the array at indexes 0,1,2,3,4.. and so on up to 9, but then when the 11th number comes in, we wrap around and store it back in index 0, and then continue through again. In this way the array always holds the 10 most recently added numbers, and older ones are overwritten.

You would implement a ring buffer like this:

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int buffer\_index = 0;

int data;

while (1)

{

data = get\_a\_number(); // Use whatever mechanism you need to get a value

buffer[buffer\_index] = data; // Set the current position in the array equal to the data

buffer\_index++; // Increment buffer\_index by 1

if (buffer\_index == BUFFER\_SIZE)

buffer\_index = 0;

}

Note that instead of just hard-coding the value '10' in, we used a #define statement. This is virtually identical to the .EQU statements we used in assembly. The compiler goes through and replaces every instance of BUFFER\_SIZE with 10. Note that the #define statement DOES NOT have a semicolon at the end, just like the other compiler directives like #include do not have one either.

In this code snippet, you see that when buffer\_index reaches the buffer size, we loop back to zero. Remember that this is because valid indexes for an array are from 0 to size – 1, so as soon as our index is equal to size, we've gone past the end and should start over at the beginning.

Also provided for you is a function called separate\_number(int num, int \*array) that makes it easy to split a number from 0 to 9999 into the four base-ten digits. The comments on the function explain its use.

So here is some pseudo code providing an explanation of how your program should operate:

main()

{

// Setup variables, arrays for the buffer and a length 4 array for the digits.

while(1) // the main loop

{

// Wait for a pushbutton press

// Read the value from the switches and store it into the buffer at the current position

// Increment the position, and if it reaches end of buffer, wrap back around to 0

// Wait for the pushbuttons to be released

// Sum up the buffer array and calculate the average

// Use the separate\_number function to separate the average into digits

// Use the provided lookup table array to convert the 0-9 digits into hex values for the

// seven segment display.

// Shift the digits as appropriate (thousands place gets shifted 24 bits to left, hundreds

// shifted 16 bits to the left, tens 8 bits, and ones place none) Use the << operator, for

// example:

// number = number << 8;

// would shift the variable 'number' to the left 8 bits

// OR all these values together into a single integer

// Output the combined value to the seven-segment display. The base address to use

// with the IOWR\_ALTERA\_AVALON\_PIO\_DATA() is HEX3\_HEX0\_BASE

}

}

Questions:

1) Try putting a single value onto the switches and press a pushbutton several times... How might code like this be useful?

2) Try putting a constant value in, and filling up the buffer (say, set the switches to 16 and press the button several times until the average is 16). Then set a very high value (say the highest switch) and press the button once (to simulate a single erroneous value, or a moment of noise). Then return the switch to 16 and press the button several more times. How much does the value change? Explain the tradeoff between noise peak and duration.

3) Try very large numbers for BUFFER\_SIZE, say 50. This might limit the effect of a noise spike, but what is the tradeoff?

Extra:

After using the separate\_number() function, you may have hard-coded all the four lookups using the lookup[] array. You might have also hard-coded the shifting for the four digits. Can you rewrite all of those into loops?