Engineering 304
Lab #4: NIOS II Assembly Language – Subroutines (cont.) & Arrays

Purpose and Overview

The purpose of this lab is to practice with the use of function calls and to learn the way that 2-dimensional arrays are stored in memory.

Part I

In this part of the lab, you will develop assembly language code for generating values for an array in memory called SumArray. SumArray[n] will be the sum of the numbers from 0 to n. For example, SumArray[5] = 0 + 1 + 2 + 3 + 4 + 5. The structure of the program will be similar to the factorial program from SW03 with a main function that calls a SumCalc(n) function and then writes the return value in the array. In addition, the SumCalc(n) function should be written in a recursive way with SumCalc(0) = 0 as the end state of the recursion and SumCalc(n) = n + SumCalc(n-1). Below are the specific requirements.

- The main function shall be written in assembly code and shall be treated as the "_start" function of the system,
- The main function shall generate the sum for 0 <= n < 50 by calling the SumCalc(n) function 50 times,
- All summing calculations are done recursively in the SumCalc(n) function and not in the main function,
- The main routine shall store each sum value in an array (global variable) in memory,
- SumCalc(n) is **not** allowed to access the global array, any other global variables, or any register used by the main function (except sp, ra, and when explicitly passing arguments and results),
- The main function shall only supply the argument "n" to the SumCalc(n) function,
- The SumCalc(n) function shall be written in assembly code,
- The SumCalc(n) function shall only return the summation value to the main routine after completing the recursive calculation,
- All assembly language functions shall properly use registers according to the conventions outlined in the CPU reference manual and shall use <u>at least one</u> of the registers R8-R23 (R8-15 in the main routine or R16-23 in the subroutine) that must be saved,
- All functions shall properly use the stack according to the conventions outlined in the CPU reference manual, and
- The NIOS cpu shall be based on the provided *.sof file, *.jdi, and *.qsys files. Recall that this version of the NIOS CPU has 32,768 bytes (32 kbytes) of memory, starting at 0x8000 and ending at 0xffff.

Each function should be placed in its own assembly language file. Remember, you will need all of the "_start" and other setup commands as were found in the sampleprog.s file, as well as the global array declaration, <u>and you will need to properly initialize the stack pointer</u>. After debugging your program, get a screenshot of the array in memory in decimal.

Part II
In this part of the lab, you need to look at how the stack grows and potentially reaches the part of memory where your global variables and instructions are stored.

Begin by studying the use of the stack when calculating the sums in Part I.

1. How many items are pushed on the stack each time SumCalc is called?
2. If the recursion depth is "n", how many stack entries are needed (in terms of n) for a single top-level SumCalc(n) call?
3. Develop an equation (in terms of n) for the memory space (in bytes) needed for the stack as a function of "n," based on the way you wrote your _start and SumCalc functions.

Next, consider the amount of memory used for the executable code and the global array.

4. How much memory is needed for your machine code and how much is needed for your global array, based on n?
5. Determine an equation for the amount of memory (in bytes) needed for both as a function of the array size "n."

Notice that the stack will grow upward (toward the first memory location) with "n" and the program + global array grow downward (toward the last memory location) with "n."

6. Knowing the size of your actual memory, determine the largest value of "n" that is safe such that the two sections of memory do not overlap.

Test it out and show a screenshot of the area of memory where the two sections are coming together for your maximum safe "n." Determine the ranges of addresses used to store the instructions (.text), the global array (.data), and the stack. Include in your writeup an explanation of your equations and how you arrived at the maximum safe "n."

Part III
In this part of the lab, you will investigate how arrays are stored in memory. When you write assembly code and create your own 2-D arrays, then you can use either “row-major” or “column-major” for saving the array. Since memory is linear (1-demnsional), a 2-D array will have to be “linearized.” In a row-major implementation, the first row’s elements fill the first memory locations, the second row’s elements fill the next memory locations, and so forth. In a column-major implementation, the first column’s elements fill the first memory locations, the second column’s elements fill the next memory locations, and so forth.

Write a simple **C-program** that declares a 2-row and 5-column **global** array of integers and initialize the values stored in the array to those shown in the table below. The main() program can simply return the value 0.

11	12	13	14	15
21	22	23	24	25

Compile and debug your program. Locate your array in the memory tab. It should be found near the end of the machine code generated by the compiler. Show one item per row (instead of the usual 4 per row) as decimal values and get a screen capture. Does C use row-major or column-major for storing 2-D arrays? Fill in the following table with “X”s after you look up how other languages store 2-D arrays. Place an “X” in the appropriate column for each language.

Language	Uses Row-Major	Uses Column-Major
C		
C++		
Fortran		
Python		
Mathematica		
Pascal		
MatLab		
R (statistics)		

Extra Credit
Determine an expression for the number of instructions executed when calculating the “SumArray[]” values as a function of “n.” This would be proportional to the complexity of the algorithm. Also, estimate the time to execute the program with the maximum safe “n” given that the processor runs at 50 MHz and will ideally execute 1 instruction each clock period. Compare your estimate with the actual execution time measured as best you can with a stopwatch. Hint, run the program multiple times programmatically to increase the overall duration to reduce the effect of human perception on your result.

Hand In
Create, print, and turn in a document using the template available on the server.