# Laboratory Exercise 2

Subroutines and Stacks

The purpose of this exercise is to learn about subroutines and subroutine linkage in the Nios II environment. This includes the concepts of parameter passing and stacks.

The background knowledge needed to do this exercise can be acquired from the tutorial *Introduction to the Altera Nios II Soft Processor*, which can be found in the University Program section of the Altera web site.

We will use the DE2 Basic Computer with your application programs being loaded in the SDRAM memory.

## Part I

We wish to sort a list of 32-bit unsigned numbers in descending order. The list is provided in the form of a file in which the first 32-bit entry gives the size (number of items) of the list and the rest of the entries are the numbers to be sorted. Implement the desired task, using the Nios II assembly language, as follows:

1. Write a program that can sort a list contained in a file that is loaded in the memory starting at location LIST_FILE. Assume that the list is large enough so that it cannot be replicated in the available memory. Therefore, the sorting process must be done "in place", so that both the sorted list and the original list occupy the same memory locations.
2. Download the DE2 Basic Computer onto Altera's DE2 board, as was done in Lab 1.
3. Compile and download your program using the Altera Monitor Program.
4. Create a sample list and load it into the memory. (Make sure that the list and your program code do not overlap in the memory.)
   Note: The file that contains the list can be loaded into the memory by using the Monitor Program, as explained in Section 11 of the Altera Monitor Program tutorial.
5. Run your program and verify that it works correctly.

## Part II

In this part, we will use a subroutine to realize the sorting task. To make the subroutine general, the contents of registers used by the subroutine have to be saved on the stack upon entering and restored before leaving the subroutine. Note that the stack has to be created by initializing the register *r27*, which is used as a stack pointer and can be referred to as *sp* in assembly language code. It is a common practice to start the stack at a high-address memory location and make it grow in the direction of lower addresses. The stack pointer has to be adjusted explicitly when an entry is placed on or removed from the stack. To make the stack grow from high to low addresses, the stack pointer has to be decremented by 4 before a new entry is placed on the stack and it has to be incremented by 4 after an entry is removed from the stack. The value 4 is used because the memory is organized in 32-bit words and it is byte-addressable, hence there are 4 bytes in a word.

Implement the previous task by modifying your program from Part I as follows:

1. Write a subroutine, called SORT, which can sort a list of any size placed at an arbitrary location in the memory. Assume that the size and the location of the list are parameters that are passed to the subroutine via registers, such that
   - The parameter *size* is given by the contents of Nios II register *r2*.
   - The address of the first entry in the list is given by the contents of register *r3*. (Note that this address is not the same as the starting address of the file LIST_FILE.)
2. Write the main program which initializes the stack pointer, places the required parameters into registers *r2* and *r3*, and then calls the subroutine SORT. The list is loaded into the memory at location LIST_FILE. The subroutine has to save on the stack the contents of the registers it uses, and restore these registers prior to returning to the main program.
3. Compile and download your program.
4. Create a sample list, load it into the memory, and run your program.

## Part III

Modify your program from Part II so that the parameters are passed from the main program to the subroutine via the stack, rather than through registers.

Compile, download, and run your program.

## Part IV

A Nios II processor uses the *ra* register (*r31*) to hold the return address when a subroutine is called. In the case of nested subroutines, where one subroutine calls another, it is necessary to ensure that the original return address is not lost when a new return address is placed into the *ra* register. This can be done by storing the original return address on the stack and then reloading it into the *ra* register upon return from the second subroutine.

To demonstrate the concept of nested subroutines, we will use the computation of $n^{th}$ number in the Fibonacci sequence. The $n^{th}$ Fibonacci number is computed as

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Note that Fib(0) = 0 and Fib(1) = 1.

Write a program that uses recursion to compute the $n^{th}$ Fibonacci number. The program has to include a subroutine, called FIBONACCI, which calls itself repeatedly until the desired Fibonacci number is computed. The main program should pass $n$ as a parameter to the subroutine by placing it on the stack. The subroutine may return the result through register *r4* (to make your task somewhat simpler!).

Compile, download, and run your program. Verify its correctness by trying different values of $n$.

## Preparation

Your preparation should include the assembly language programs for Parts I to IV.