Laboratory Exercise 3

Subroutines and Stacks

This exercise is about subroutines and subroutine linkage using the ARM A9 processor. You will learn about the concepts of parameter passing, stacks, and recursion. For this exercise you have to know the ARM processor architecture and its assembly language, and you should have a basic understanding of the C programming language.

Part I

You are to write an ARM assembly language subroutine called FINDSUM that uses a loop to compute the summation $\sum_{i=1}^{N} i$. Equivalent code for your subroutine in the C language is shown below.

```
int FINDSUM(int N)
{
    int sum = 0;
    while (N != 0)
    {
        sum = sum + N;
        N = N − 1;
    }
    return sum;
}
```

You need to provide a main program that calls your FINDSUM subroutine. Part of this main program is shown below. The value of argument N that is used for your subroutine is stored in memory as shown in the code—your main program needs to load this value from memory and pass it to the subroutine, using processor register R0. Return the result from the subroutine in the same register, R0.

```
            .text
            .global  _start
_start:
            . . . get N and pass to subroutine
            BL       FINDSUM
END:        B        END                 # wait here

FINDSUM:    . . .
            . . .

N:          .word    9
            .end
```

Perform the following:

1. Create a new folder and make a Monitor Program project for your summation code. Select the ARM A9 processor and use the DE0-Nano-SoC Computer.
2. Assemble and download your program. Test it for various values of N.

Part II

You are to write an assembly language program that sorts a list of 32-bit numbers into descending order. The first entry in the list gives the number of data elements to be sorted, and the rest of the list provides the data. The list of data must be sorted "in place", meaning that you are not allowed to create a copy in memory of the list to do the sorting.

A program written in the C language that performs the required sorting operation is shown in Figure 1. This program implements a simple bubble-sort algorithm. It uses an outer loop to traverse the list a number of times until sorted. An inner loop calls the *SWAP* subroutine, not shown in the figure, which swaps the list elements in memory when needed.

Write a main program and subroutine in the ARM assembly language that is equivalent to the C code in Figure 1. Your SWAP subroutine should be passed the address of a list element in processor register R0, and should provide its return value to the main program in the same register.

The list can be defined as part of the data for your assembly language program as follows:

```
List:    .word    10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Perform the following:

1. Create a new folder and make a Monitor Program project for your sorting code. Select the ARM A9 processor and use the DE0-Nano-SoC Computer.
2. Test your algorithm with various data sets and ensure that the list of data is properly sorted in-place in the memory. A good debugging technique for this code is to use the Memory tab in the Monitor Program to view the contents of the list as the sorting algorithm progresses. Each time a breakpoint is reached by the processor (or an instruction is single-stepped), the list can be examined to see how the items are being swapped.

```
int LIST[ ] = {10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int main(void)
{
   int i, flag, length, * item_ptr;

   length = LIST[0];                  // number of elements to be sorted
   do
   {
      flag = 0;                       // indicates if nothing remains to be sorted
      item_ptr = LIST + 1;            // pointer to the first element of data
      for (i = 1; i < length; i++)
      {
         flag |= SWAP (item_ptr);
         ++item_ptr;                  // point to the next word of data
      }
      - -length;                      // last item in the list is in the right place
   }
   while (flag);
}
```

Figure 1: A bubble-sort algorithm.

Part III

For this part of the exercise you are to rewrite your FINDSUM subroutine from Part I to make it recursive. Equivalent C code for your subroutine is shown below.

```
int FINDSUM(int N)
{
    if (N != 0)
        return N + FINDSUM (N-1);
    else
        return 0;
}
```

In your assembly-language code, make sure to initialize the ARM A9 stack pointer to a suitable value, and use the stack to save the state of the FINDSUM subroutine each time that it recurses.

Create a new folder and make a Monitor Program project for your recursive code. Assemble, download, and test your program.

Part IV

You are to write an assembly-language subroutine that computes the 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.

Your subroutine has to be recursive. Equivalent C code for such a subroutine is shown below.

```
int FIBONACCI(int N)
{
    if (N < 2)
        return N;
    else
        return FIBONACCI(N−1) + FIBONACCI(N−2);
}
```

You need to provide a main program that calls your FIBONNACI subroutine, in the same way as for the earlier parts of the exercise. The value of the argument N should be loaded from memory and passed to your subroutine. You can assume that $N > 1$.

Make sure to initialize the ARM A9 stack pointer to a suitable value, and use the stack to save the state of the FIBONNACI subroutine each time that it recurses.

Create a new folder and make a Monitor Program project for your Fibonacci code. Assemble, download, and test your program.