Raffy Santayana

Worked together with Robby Bergers

Lab 2

We have used the following sources:  
 <https://homepage.cs.uiowa.edu/~ghosh/GettingStarted.pdf>

<https://users.dimi.uniud.it/~pietro.digianantonio/arm_documenti/ARMSim_UserGuide.pdf>

<https://community.arm.com/processors/b/blog/posts/how-to-call-a-function-from-arm-assembler>

<http://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

**Factorial:**

The code to calculate a factorial n can be found in factorial.s. I have commented on each line to help explain the process of calculating the factorial. This document is to walk through it being executed step-by-step. In this example, I will be calculating 5!.

First, in main, the values of 5 and 0 are stored in R0 and R2 respectively. We have decided to use R0 to be equal n and R2 to the factorial of n. Then we branch to *factorial*.

In *factorial*, we store the link back to main so that once we pop the program counter later on, we are brought back to main and can terminate the code. After that we check to see if R0 is equal to 0. This is because we will be eventually decrementing R0 by one for every “term” we calculate. “Term” being an integer that is less than or equal to n. I will explain this further ahead. The first time though the code, R0 will not be equal to 0. This means that the code has not executed the section *factorialWhile*, so we must jump there to start calculating the factorial properly.

In *factorialWhile*, we start off by checking if R2 is equal to 0. This is a check to see if the loop has been executed as least once. So instead of branching to *factorialNext*, we will first be executing the instructions after the branch statement.

R4, a temporary value used to calculate n \* (n-1), is pushed onto the stack. We then decrement R0 by one and store it as R4. At this point, we can multiply R0 \* R4 to perform  
n \* (n – 1).

In theory we have performed the following:

Whole factorial: 5! = 5 \* 4 \* 3 \* 2 \* 1

What we have done so far: 5 \* 4 = R2

We then decrement R0 twice because, as I stated before, we must decrement it for every term we calculate. Now, we must find the factorial of 3 because:

3! \* (5 \* 4) = 5!

This is why we have decremented twice. After doing so, we pop R4 from the stack because it is now unnecessary to have and continue to branch to *factorial* and execute it again.

In *factorial*, we compare R0 and 0 again. At this point, R0 contains 3. So we branch to *factorialWhile* once again.

Back in *factorialWhile*, we compare R2 and 0 again in order to find out if the calculation has already gone through the process at least once. Since R2 currently holds the value 20, we will branch over to *factorialNext*.

In *factorialNext*, we push R5 into the stack as a temporary value to calculate the factorial of our initial R0. We then store R2 into R5 so that we can multiply

[n \* (n-1)] \* (n-2)

If n = 5, then we perform

R5 = R2 \* R0

= (5 \* 4) \* (3)

= 60

Once that has been done, we decrement R0 by one so that we can multiply our current R2 by 2!. We then store R5 into R2 and pop R5 once we are done and repeat the process by branching to *factorial* again.

Since the factorial isn’t over with, we branch to *factorialWhile* and then *factorialNext* again and repeat until done with.

R5 = R2 \* R0

= 60 \* 2

=120

R0 = 1

The process is done one last time even though R2 holds the final answer because R0 =/= 0. Once the process is over, we jump to *finish* and terminate.

**Fibonacci:**

The code to calculate the nth term of the Fibonacci sequence can be found in fibonacciSecuence.s. I have commented on each line to help explain the process of calculating the nth term. This document is to walk through it being executed step-by-step. In this example, I will be calculating the 6th term.

First, in *main*, we load the value 6 into R0. R0 = n, which will be the term in the Fibonacci sequence that we will be looking for. Next, we will branch over to *fibStart*.

In *fibStart*, we start off by storing the link back to main and pushing R4, R5, and R6 onto the stack. We then load the values 2, 1, and 1 into the registers respectively.

R4 is a counter that will be compared to R0. If they are equal to each other, then we will branch to finish the calculation. We start R4 equal to 2 because we have already done the first two terms.

R5 is a value in the Fibonacci sequence. We will always be starting the sequence off as

1 1

We decided that R5 should be the third-to-last number in the sequence, so it is currently set to 1.

R6 has the same purpose as R5, but it is the second-to-last number in the sequence.

Now that we have done all the prepwork, we can now being calculating the 6th term in *fib*.

We increment R4 by one for each recursion first. Then we push R7 onto the stack. R7 is the value of the upcoming term. So we follow up by adding

R5 + R6 = R7 = 2.

Then we store R6 into R5, and R7 into R6 to update the new second and third-to-last values.

Once that is done, we pop R7 off the stack and compare R4 with R0. R4 is contains the value 3, which means we will go through *fib* again. We do so until the condition R4 =/= R0 is not longer true.

After going through *fib* again, R4 is incremented by 1

R4 = 4

R5 and R6 are added together and stored in R7.

1 + 2 = 3 = R7

Then the registers R5 and R6 are updated.

R5 = 2 R6 = 3

Then *fib* repeats.

Fast forward until

R0 = 6 R4 = 6 R5 = 5 R6 = 8

and we just finished popping R7 off the stack in *fib*. Next, we will compare R0 and R4. Now that they are equal to each other, we no longer repeat our process. We continue onwards to *fibEnd.*

Here, we store R6 into R1 and pop all the values into the stack and then pc to return to main. Now, we jump to *finish* and terminate our code. When we terminate:

R0 = 6 R1 = 8