Lab 04: What's Your Calling? Subroutines, Recursion, and the Stack
Jonathan Gusler
2/12/2019

Overview

This lab's primary focus was creating a subroutine that didn't require the use of the general-purpose registers outside the subroutine. The subroutine, called factN, calculates the factorial of numbers less than six. Having been implemented as a recursive subroutine, not using the general-purpose registers meant that values for the calculation could not be stored between subroutine calls in the general-purpose registers, they had to be stored in the stack. The only use of the general-purpose registers came from pushing the initial value before the subroutine call, popping the answer after the subroutine was done, and storing values during a subroutine operation for purposes of arithmetic.

Background

This lab requires knowledge of Atmel assembly language, Atmel's debugger tool, recursion, what a factorial is, how stack operations like push and pop work, how a stack "grows" and "shrinks" in relation to the memory addresses, and what a stack pointer is.

Discussion

The first step of creating factN was planning it out in pseudocode and then writing it in assembly language by hand. This allowed for solving the problem with simple steps that progressively built on each other. While writing the subroutine in assembly code by hand, sketches of the stack frame at important steps were also drawn for visual aid. Finally, the code was tested in the Atmel IDE. The pseudocode for recursive factorial, Figure 1, is:

```
factN(n)
{
      if (n ==1)
            result = 1
      else
            temp1 = n
            temp2 = factN(n-1)
            result = temp1 x temp2
      end
      return result
}
```

Figure 1: Pseudocode for recursive factorial [1].

The assembly code for this, given the caveat of not storing values in registers between calls and returning the stack to its original state before the call with the result replacing the n initially pushed to the stack, is shown in References

[1] D. G. Nordstrom, "EECE4253Lab04," [Online]. Available: https://drive.google.com/drive/folders/1Av3dayhOQwMNdSrBdB2UJXu8f_4GPd57. [Accessed 14 February 2019].

Appendix A. An example of the stack frames in sequence, assuming the value of n is 3, with SP written in the box where the stack pointer is pointing, is shown in Table 1 on the next page.

			SP							
			00							
			47	SP	SP					
		SP	1	1	1	SP				
		00	00	00	00	00				
		47	47	47	47	47	SP	SP		
	SP	2	2	2	2	2	2	2	SP	
	00	00	00	00	00	00	00	00	00	
SP	14	14	14	14	14	14	14	14	14	SP
3	3	3	3	3	3	3	3	6	6	6

Table 1: Stack frames example with n equal to 3.

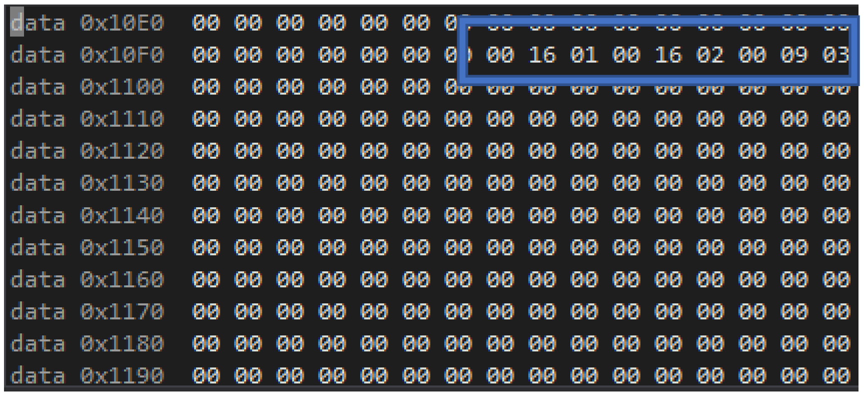
The sequence of events that describes the stack frames from left to right is as follows:

1) Push n onto the stack
2) Call factN
3) n is not one, it's three, so decrement n and push that onto the stack and call factN
4) n is not one, it's two, so decrement n and push that onto the stack and call factN
5) n is one, now in a stage of unrolling the stack, so return.
6) multiply one by two and replace SP + 4 with that product
7) pop top value because value is no longer needed
8) return as part of unrolling

9) multiply two by three and replace SP + 4 with that product
10) return leaving final value on stack for user to pop when value is wanted

Analysis and Results

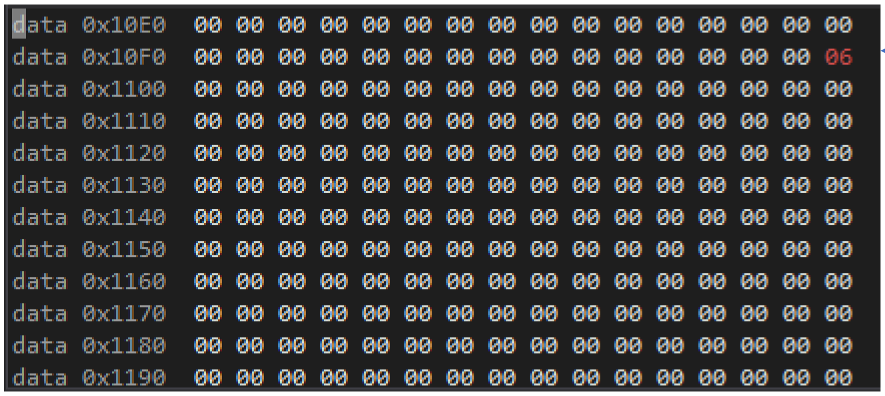
Figure 2 and Figure 3 show, from the Atmel IDE, the registers of the stack after the base case has been reached, but before unrolling the stack, and after factN is done, so the result is still in the stack, respectively. It is important to note that the register address values in between the factorial numbers are different than in the stack frames from Table 1 because they are from a real program, whereas in the handwritten version the values are more arbitrary.



```
data 0x10E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x10F0  00 00 00 00 00 00 00 00 16 01 00 16 02 00 09 03
data 0x1100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

factN results in stack before unrolling.

Figure 2: Stack after base case has been reached, before unrolling.



```
data 0x10E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x10F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06
data 0x1100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data 0x1190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

factN result in stack before user pops it.

Figure 3: Stack after unrolling, before user has popped off the final result.

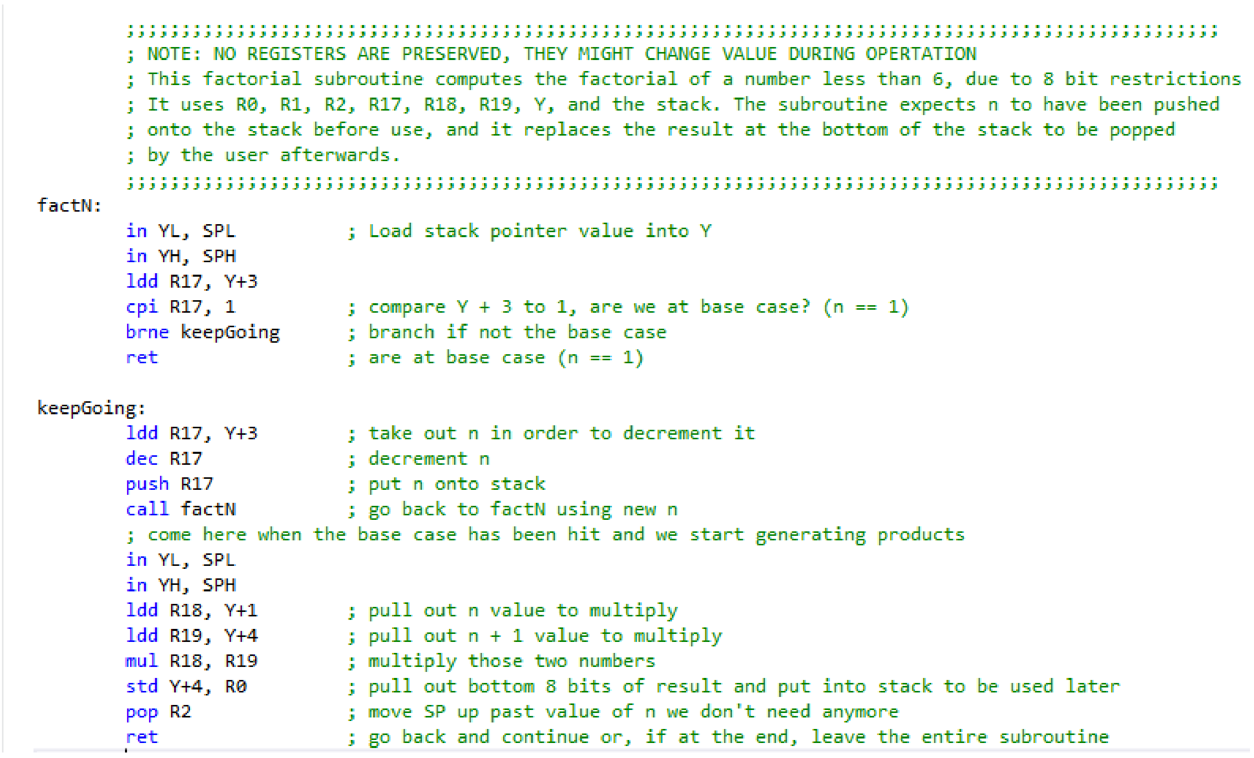
Conclusion

The recursive subroutine, factN, calculates the factorial for any value, n, less than six. Registers are not needed between function calls, only the stack is used for saving values. As shown in the example of Figure 3, using an n value of three, the program works as intended. Further development will be needed if values greater than five are to be used. This development requires adjusting for 16-bit values being generated by the factorial, rather than just 8-bit values.

References

[1] D. G. Nordstrom, "EECE4253Lab04," [Online]. Available: https://drive.google.com/drive/folders/1Av3dayhOQwMNdSrBdB2UJXu8f_4GPd57. [Accessed 14 February 2019].

Appendix A



```
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ; NOTE: NO REGISTERS ARE PRESERVED, THEY MIGHT CHANGE VALUE DURING OPERTATION
        ; This factorial subroutine computes the factorial of a number less than 6, due to 8 bit restrictions
        ; It uses R0, R1, R2, R17, R18, R19, Y, and the stack. The subroutine expects n to have been pushed
        ; onto the stack before use, and it replaces the result at the bottom of the stack to be popped
        ; by the user afterwards.
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
factN:
        in YL, SPL          ; Load stack pointer value into Y
        in YH, SPH
        ldd R17, Y+3
        cpi R17, 1          ; compare Y + 3 to 1, are we at base case? (n == 1)
        brne keepGoing      ; branch if not the base case
        ret                 ; are at base case (n == 1)

keepGoing:
        ldd R17, Y+3        ; take out n in order to decrement it
        dec R17             ; decrement n
        push R17            ; put n onto stack
        call factN          ; go back to factN using new n
        ; come here when the base case has been hit and we start generating products
        in YL, SPL
        in YH, SPH
        ldd R18, Y+1        ; pull out n value to multiply
        ldd R19, Y+4        ; pull out n + 1 value to multiply
        mul R18, R19        ; multiply those two numbers
        std Y+4, R0         ; pull out bottom 8 bits of result and put into stack to be used later
        pop R2              ; move SP up past value of n we don't need anymore
        ret                 ; go back and continue or, if at the end, leave the entire subroutine
```