ARM Activation Frame Stack

Tom Kelliher, CS 220

1 Administrivia

Today's Objectives

1. Write ARM programs that pass function parameters, and return function values.

2. Write ARM programs that use the stack frame for the storage of local variables, allowing the implementation of recursion and freeing programs from the limited number of available registers.

Next Up

Read 2.9–2.11

1. Achieve a basic understanding of combinational and sequential digital logic:

   (a) AND, OR, NOT, etc. gates.

   (b) Basic combinational circuits: full- and half-adders, decoder, multiplexer, etc.

   (c) Flip-flops, registers, and counters.

   (d) Basic implementation of buses.

2 Warm-Up

1. The activation frame stack grows toward

   (a) It doesn't grow.

   (b) the first memory address — 0.

   (c) the last memory address — whatever it may be.

   (d) None of the above.

2. Activation frames are necessary

   (a) never.

   (b) always.

   (c) for preserving register values across function calls when functions call other functions.

   (d) as backup storage when the number of local variables exceeds the number of available registers.

   (e) (c) and (d).

3. If a leaf routine only uses registers r0–r3, it doesn't need an activation frame.

   True/False.

4. sp (r13) points to an in-use memory location.

   True/False.

5. fp (r11)

   (a) points to the base of the current activation frame.

   (b) can be used to access parameters passed on the stack.

   (c) points to a linked list of all activation frames on the stack.

   (d) All of the above.

   (e) None of the above.

6. According to our convention, we have seven registers (r4–r10) for storing local variables. A function with more than seven locals

   (a) will not "compile."

   (b) Who would write such a function???

   (c) must have space allocated in its activation frame to temporarily store unused-at-the-moment locals.

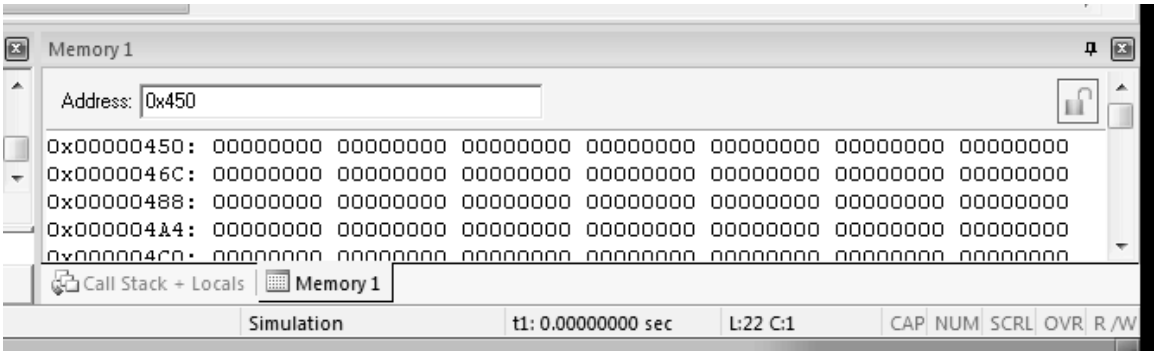
   (d) will use an external set of arrays to store the locals that don't fit into registers.

3 Problems

1. Create a new uVision project for factRecursive.s, available on the course web site. You'll also need to add map.ini under the debugger settings.

2. Build the project, open the debugger, and set breakpoints on the three lines indicated in the program (these will be at/near lines 39, 88, and 124).

3. Set the memory window so that it displays unsigned ints, starting at memory address 0X450, similar to this:



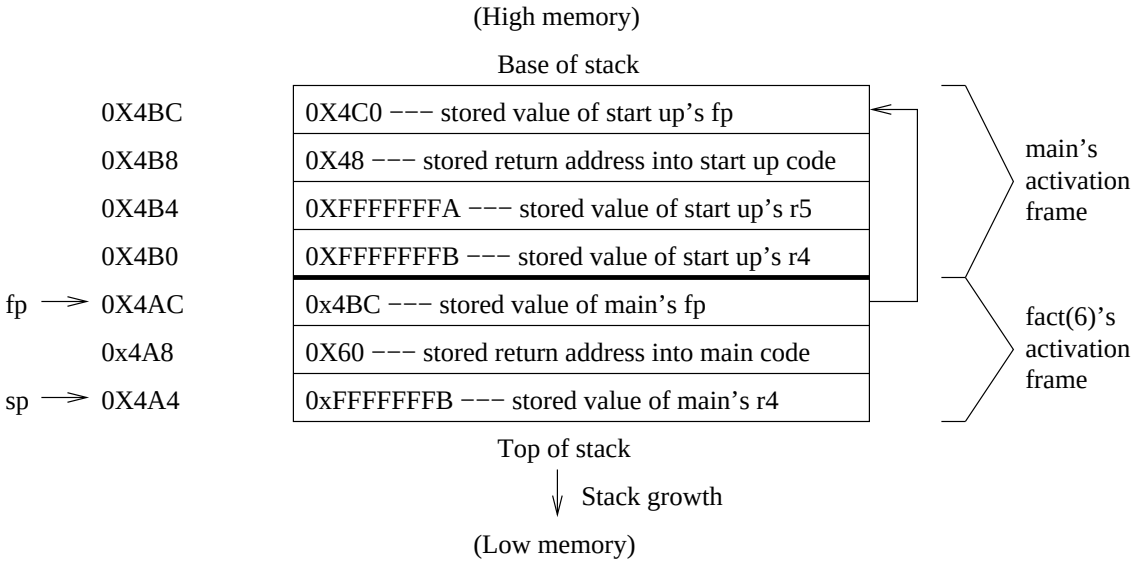
   (Select the Memory1 tab, and then right-click on the memory window itself to select unsigned int as the display type.)

4. Run the program. At the first breakpoint, you'll notice that both sp and fp (r11) are set to 0X4C0, which is a word-aligned address. With our sp-full protocol, the first push to the stack will occur to the memory word (address 0X4BC) preceding this memory word.

5. Continue execution to the next breakpoint, which immediately follows the creation of main()'s activation frame. In the memory window, note that four words have been pushed onto the activation frame stack. These four words are the activation frame for main().

6. Continue execution to the next breakpoint, which immediately follows the creation of fact(6)'s activation frame. In the register's window, note that r0 has the value 6, which is how we know that fact(6) is being executed. In the memory window, note that another three words have been pushed onto the activation frame stack. These three words are the activation frame for fact(6).

   Here is the annotated activation frame stack at this point in the program's execution:



7. Produce the complete, annotated activation frame stack at the point of execution of fact(1)'s activation frame having been pushed onto the stack.

8. Single-step through the execution of fact(1), and continue single-stepping to observe the recursion "unwind" as fact(n) values are computed, returned, and activation frames popped from the stack.