Lecture #18 – March 24, 2004

Parameters and the Stack

In this lecture we are going to examine the use of the stack as a general mechanism for passing parameters in to and out of a subroutine. In the next lecture we will describe how this approach maps onto parameter mechanisms used by high-level languages.

In the most basic form, we want to push parameters onto the stack before calling a subroutine, and we want to pop those parameters off of the stack after the subroutine returns to its calling point. This is illustrated by the following pseudocode:

```
Push Param1
Push Param2
Call Sub
Pop  Param2
Pop  Param1
```

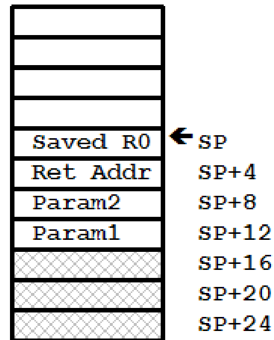
In ARM assembly language, the equivalent code is as follows:

```
LDR  R0,Param1
STR  R0,[SP,#-4]!          Push Param1
LDR  R0,Param2
STR  R0,[SP,#-4]!          Push Param2
BL   SUB                   Call Sub
LDR  R0,[SP],#4            Pop  Param2
STR  R0,Param2
LDR  R0,[SP],#4            Pop  Param1
STR  R0,Param1
```

In the ARM every subroutine is responsible for saving and restoring its own return address (unlike the 6502 and 8088, which push the return address on the stack as an automatic side effect of the call and pop it upon return). In addition, every *properly written* subroutine must save and restore any affected registers in order to prevent unintended side effects in the calling routine. In earlier examples enforcing ***transparency*** was done by saving the registers to fixed locations in memory; an effective technique but one that does not lend itself to recursion. Here we will save the registers, including the link register containing the return address, to the stack.

For our current example we will assume that the only register that needs saving, other than the link register, is `R0`. Then, by the time we are ready to "do useful work" in the subroutine the stack contains four items: `Param1`, `Param2`, the return address, and the saved value of `R0`. The stack pointer `SP` points at the memory word containing `R0` (i.e., the "last full" byte of the stack and not the "first empty" byte).

At this point the stack looks as follows:



The body of the subroutine can now access the parameters relative to the current value of the stack pointer. Parameter `Param1` is at `SP+12`, and `Param2` is at `SP+8`, and those offsets are used in the appropriate `LDR` and `STR` instructions. Suppose, for example, that the purpose of the subroutine is to place into `Param2` the value of `Param1` times 5. The complete subroutine code is as follows:

```
SUB        STR LR,[SP,#-4]!          Push LR
           STR R0,[SP,#-4]!          Push R0
           ------------------------------------------
           LDR R0,[SP,#12]           R0 := Param1
           ADD R0,R0,R0,LSL #2       R0 := R0 * 5
           STR R0,[SP,#8]            Param2 := R0
           ------------------------------------------
           LDR R0,[SP],#4            Pop R0
           LDR PC,[SP],#4            Return
```

This subroutine is too simple to warrant use of the stack in this manner, of course, but it is useful to illustrate the advantages and pitfalls of the technique. The advantages should be obvious: no explicit memory allocations are necessary, and so the subroutine could be potentially recursive. Unfortunately, there are two problems that we need to address. The first problem is that the offsets are meaningless numbers; it is difficult to remember if `SP+12` refers to `Param1` or `Param2`. By using `EQU` directives we can get around this problem, and use symbols anywhere the numeric offsets would be used. If it becomes necessary to add more items to the stack frame and thereby change the values of the offsets, only the symbols need be redefined. This approach is shown as follows:

```
Param1     EQU  12
Param2     EQU  8

SUB        STR LR,[SP,#-4]!          Push LR
           STR R0,[SP,#-4]!          Push R0
           ------------------------------------------
           LDR R0,[SP,Param1]        R0 := Param1
           ADD R0,R0,R0,LSL #2       R0 := R0 * 5
           STR R0,[SP,Param2]        Param2 := R0
           ------------------------------------------
           LDR R0,[SP],#4            Pop R0
           LDR PC,[SP],#4            Return
```

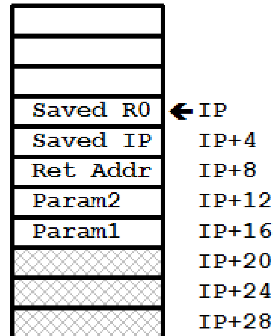
The second problem is worse. Using the stack pointer SP as a base register to index into the stack array works well only as long as the stack pointer *doesn't change* within the body of the subroutine. If Param1 is at SP+12 at the start of the "do useful work" section, pushing a new word onto the stack places Param1 at SP+16. For every new word pushed the offset increases by another four bytes. The use of symbols defined by EQU directives is now completely invalid (the symbols no longer have the correct values), and debugging a program where the offsets change is nearly impossible.

What we need is an approach where the stack contents are allowed to change, but the base address of the array containing the parameters does not. We must use one more register. On the ARM, the register to use is R12, also called the IP register. Once everything goes onto the stack (including the old value of the IP register), the IP is set to the final value of the stack pointer SP. Now the subroutine can push any number of new words onto the stack, changing SP, and the parameters are always referenced by the same offsets relative to IP.

In the following subroutine, the return address, R0, and the IP registers are all pushed onto the stack. (Any local variables needed by the subroutine are allocated on the stack at this time as well.) Once the MOV IP,SP instruction has been executed the stack is free to change as needed.

```
SUB       STR LR,[SP,#-4]!            Push LR
          STR IP,[SP,#-4]!            Push IP
          STR R0,[SP,#-4]!            Push R0
          MOV IP,SP                   IP := SP
          ------------------------------------
               Param1 is always at IP+16
               Param2 is always at IP+12
          ------------------------------------
          LDR R0,[SP],#4              Pop  R0
          LDR IP,[SP],#4              Pop  IP
          LDR PC,[SP],#4              Return
```

Here is the configuration of the stack in the body of the subroutine:

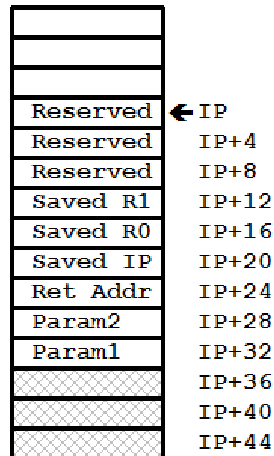


Any new pushes go on top of the stack, but IP stays fixed for the running life of the subroutine.

As mentioned earlier, we may need to reserve space on the stack for variables local to the subroutine. Suppose in our subroutine we need to save both R0 and R1, and reserve three words (12 bytes) for local storage. Those values need not be initialized to any specific value, so reserving 12 bytes is as easy as subtracting 12 bytes from the stack pointer. Deallocating those bytes at the end of the subroutine requires adding 12 back to the stack pointer.

```
SUB        STR LR,[SP,#-4]!          Push LR
           STR IP,[SP,#-4]!          Push IP
           STR R0,[SP,#-4]!          Push R0
           STR R1,[SP,#-4]!          Push R1
           SUB SP,SP,#12             Allocate 12
           MOV IP,SP                 IP := SP
           ----------------------------------------
                Param1 is always at IP+32
                Param2 is always at IP+28
           ----------------------------------------
           ADD SP,SP,#12             Deallocate 12
           LDR R1,[SP],#4            Pop  R1
           LDR R0,[SP],#4            Pop  R0
           LDR IP,[SP],#4            Pop  IP
           LDR PC,[SP],#4            Return
```

Here is the stack configuration for this scenario:



Now we have all of the pieces we need to create general purpose subroutines. In the next lecture we will look at several stack-based parameter passing mechanisms, and in the lecture after that we explore recursion.