Lecture #20 – March 29, 2004

Recursion

Subroutines written in the most general form will observe the following structure. On the ARM, of course, the return address is the link register (`LR=R14`), and the pointer to the ***stack frame*** is the instruction pointer register (`IP=R12`). Pushing registers onto the stack to enforce transparency will also include these two. Reserving space for local variables requires only adjusting (subtracting from) the stack pointer by the required amount (on the ARM this should be a multiple of 4 bytes). Once the stack has been configured, setting up the new stack frame is done by moving the final value of the stack pointer (`SP=R13`) into `IP` so that the stack is free to receive pushes and pops within the subroutine, and all parameters, registers, and local variables are referenced by the *same* offsets relative to `IP`.

```
Sub  Push Return Address
     Push Registers (Including Stack Frame)
     Push Space for local variables
     Set up new Stack Frame
     …
     …
     …
     Discard local variables
     Pop Registers
     Return
```

Using ARM code, the framework above is written as:

```
Sub  STR  LR,[SP,#-4]!   Save Return Address
     STR  IP,[SP,#-4]!   Save Stack Frame Pointer
     STR  R0,[SP,#-4]!   Save General Register
     STR  R1,[SP,#-4]!   Save General Register
     STR  R2,[SP,#-4]!   Save General Register
     SUB  SP,SP,#12      Reserve 12 Bytes Locally
     MOV  IP,SP          Set Up New Stack Frame
     …
     …                   "Do Useful Work"
     …
     ADD  SP,SP,#12      Discard 12 Local Bytes
     LDR  R2,[SP],#4     Restore General Register
     LDR  R1,[SP],#4     Restore General Register
     LDR  R0,[SP],#4     Restore General Register
     LDR  IP,[SP],#4     Restore Stack Frame
     LDR  PC,[SP],#4     Return
```

NOTE: there are special `LDR` and `STR` instructions on the ARM which can store and load multiple registers. Using these instructions you can push the `LR`, `IP`, and any other registers that must be saved in one `STR` instruction, and can pop them back in one `LDR` instruction. This is far more efficient than the individual pushes and pops shown in the example, but I prefer to be explicit here in order to get the code correct first. We can always optimize later.

Once a subroutine is properly initialized, it may call any other subroutine safely, *including itself*. Such ***recursive*** subroutines are reasonably straightforward to create, as long as the programmer is careful and disciplined enough to insure that the stack is always correctly configured.

The classic first example of a recursive routine is the factorial function. In some sense this is the worst possible example to use here, as no one in their right mind would ever write factorial recursively. The factorial function is much more easily written using non-recursive iterative techniques. I will not violate tradition, however, and we will go through the description of recursion in assembly language using the old, hoary example of factorial. It's too useful an example not to. For review, the factorial of an integer N is the product of all integers from 1 up to N. Under normal circumstances factorial is meaningless for negative numbers. The factorial of N is written as N!, so 4! = 1×2×3×4. Expressed recursively, N! = N × (N-1)! The basis case, which stops the recursion, is that 0! = 1, although we can also stop at 1! = 1 as well.

In a high-level language such as Pascal, the factorial function is written in the following form. Notice that the parameter `N` is a call-by-value parameter. Notice, too, that there is an `If` statement; in one branch of the `If` statement there is a recursive call, but in the other branch the function simply returns the value 1. All recursive functions require at least one such `If` in order to stop the recursion.

```
Function Factorial (N:Integer) : Integer ;
Begin
     If N <= 1 Then
          Factorial := 1
     Else
          Factorial := N * Factorial(N-1) ;
End ;
```

Calling this function from the main program may be as simple as writing out the value of some factorial, as in:

```
Write (Factorial(5)) ;
```

Since the function subroutine expects to find its parameters for the current call in the same place in the stack as for any other (recursive or non-recursive) call, every call must be set up in exactly the same way, whether it is in the main program or in the subroutine itself.

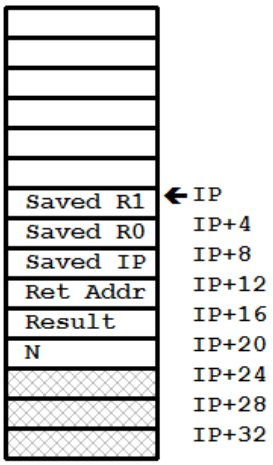
For the call from the main program, `Write (Factorial(5))`, the setup for the call in ARM code will look like this:

```
MOV R0,#5           Set up N=5
STR R0,[SP,#-4]!    Push N
SUB SP,SP,#4        Reserve Space for Result
BL  Factorial       Call Factorial
LDR R0,[SP],#4      Pop Result
ADD SP,SP,#4        Discard N
```

Setting up the stack frame in Factorial is similar to the forms we have already discussed, except that in this case we will not need any local variables (all of our computations can be done in the registers that we save). The framework for the subroutine is as follows:

```
Factorial       STR  LR,[SP,#-4]!   Save Return Address
                STR  IP,[SP,#-4]!   Save Stack Frame Pointer
                STR  R0,[SP,#-4]!   Save General Register
                STR  R1,[SP,#-4]!   Save General Register
                MOV  IP,SP          Set Up New Stack Frame
                …
                LDR  R1,[SP],#4     Restore General Register
                LDR  R0,[SP],#4     Restore General Register
                LDR  IP,[SP],#4     Restore Stack Frame
                LDR  PC,[SP],#4     Return
```

Calling the subroutine places an ***activation record*** on the stack, which is a complete environment for the current subroutine call. In our example, we want the "do useful work" part of the subroutine to have the following view of the current activation record on the stack:



As far as the current activation record is concerned, it doesn't know and doesn't care what is below it in the stack. There may not anything below it in the stack, or there may be the activation record for a previous subroutine call. If the subroutine calls itself, it will create a new *identical* activation record on the stack, which will then be the current environment for the life of that call. This happens for as many recursive calls as required (as long as the stack isn't full). When any recursive call exits its activation record will be discarded, and the one below it in the stack will become active once again. One activation record will be discarded every time the recursion "unwinds", eventually unwinding all the way back to the original call from the main program.

We will define symbols `ParamN` to have the value 20 and `ParamResult` to have the value 16. These symbols indicate the *offsets* into the current activation record of input parameter `N` and the space reserved for the function result. The complete factorial function in ARM assembly language is written as follows:

```
ParamN          EQU  20
ParamResult     EQU  16

Factorial       STR  LR,[SP,#-4]!    Save Return Address
                STR  IP,[SP,#-4]!    Save Stack Frame Pointer
                STR  R0,[SP,#-4]!    Save General Register
                STR  R1,[SP,#-4]!    Save General Register
                MOV  IP,SP           Set Up New Stack Frame
-------------------------------------------------------------
                LDR  R0,[IP,ParamN]  If N <= 1 Then
                CMP  R0,#1
                BGT  Main
                MOV  R0,#1                Store 1 into Result
                STR  R0,[IP,ParamResult]
                B    Done            Else
Main            LDR  R0,[IP,ParamN]
                SUB  R0,R0,#1
                STR  R0,[SP,#-4]!         Push (N-1)
                SUB  SP,SP,#4             Reserve Space
                BL   Factorial            Call Factorial
                LDR  R0,[SP],#4           Pop Result
                ADD  SP,SP,#4             Discard (N-1)
                LDR  R1,[IP,ParamN]
                MUL  R0,R1,R0             Compute N * (N-1)!
                STR  R0,[IP,ParamResult]  Store into Result
Done
-------------------------------------------------------------
                LDR  R1,[SP],#4      Restore General Register
                LDR  R0,[SP],#4      Restore General Register
                LDR  IP,[SP],#4      Restore Stack Frame
                LDR  PC,[SP],#4      Return
```

Study the ARM code carefully, and try running it for `N`=5 (your final result should be 5! = 120). At the deepest level you will have five activation records on the stack, one each for the environments where `N`=5, `N`=4, `N`=3, `N`=2, and `N`=1. Since each activation record requires six 32-bit words on the stack, the stack at its deepest will be 30 words (120 bytes) deep. For each activation record the offset of the function's return value is 16 bytes into the stack (based on `IP`), which is popped off when the routine returns back to the previous activation record.

As you can see, subroutine calls, and in particular recursive subroutine calls, are a synergistic coordination between the calling routine and the called routine. Each places specific items onto the stack and each is responsible for removing those same, possibly modified, items from the stack. This dance must be perfect or the program will crash (usually by branching to an incorrect location in memory).