1 Function

Any function written in the most general form for the ARM architecture will observe the following structure.

- Return address is the link register (LR=R14)
- Pointer to the stack frame is the instruction pointer register (IP=R12)
- Reserving space for local variables requires only adjusting (subtracting from) the stack pointer by the required amount (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 function, and all parameters, registers, and local variables are referenced by the same offsets relative to IP.

The common structure to handle function call using stack:

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

Equivalent ARM assembly code

```
Label   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
          ...
; 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 pointer
  LDR PC, [SP],#4 ; return
```

NOTE: there are special LDR and STR instructions on the ARM: LDM and STM which can store and load multiple registers.

2 Recursive Function

Recursive functions 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.

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

Example: Assembly code to call the function (Factorial(4)) from the main program:

```
MOV R0,#4 ; Set up N=4
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
```

The framework for the function 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
          ...; do useful work
  LDR R1,[SP],#4 ; restore general register
  LDR R0,[SP],#4 ; restore general register
  LDR IP,[SP],#4 ; restore stack frame pointer
  LDR PC, [SP],#4 ; return
```

Calling the function places an activation record on the stack, which is a complete environment for the current function call. In our example, we want the "do useful work" part of the function to have the view of the current activation record on the stack, shown in fig. 1:

Study the ARM code carefully, and try running it for N=4 (your final result should be 4! = 24). At the deepest level you will have four activation records on the stack, one each for the environments where 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 24 words (96 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.

3 Nested Function

When the first function is called, the **lr** will be set to point to the return address that function should use. If that function calls a second function using **BL**, the **lr** will be replaced with the

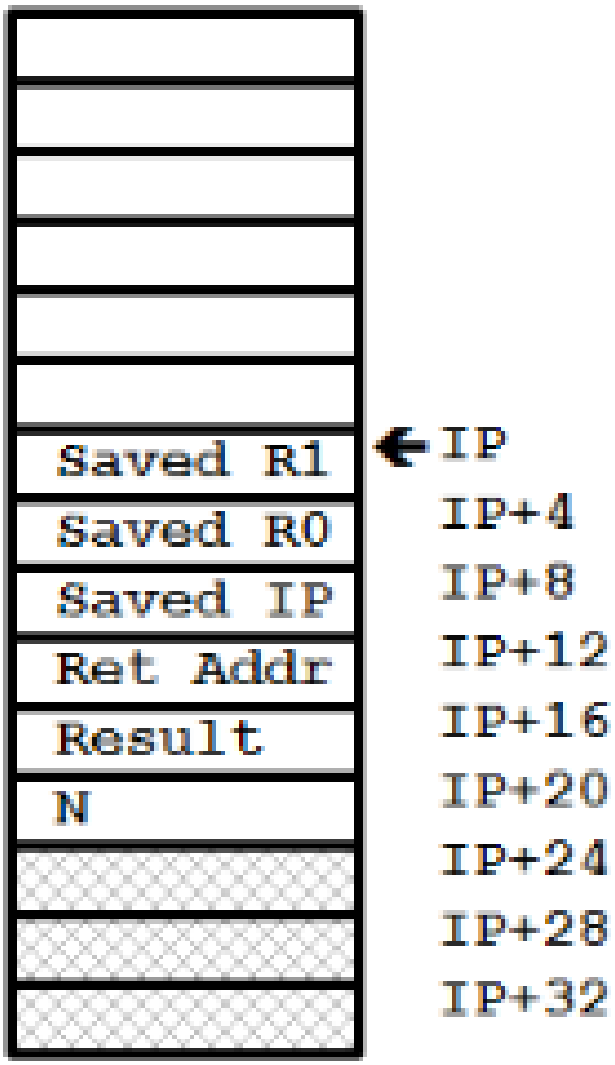

Figure 1: Stack Contents

address to return to within the first function and wipe out where the first function needs to return to!

To solve this problem, we need to store the current value of the **lr** to the stack as we enter the function, and restore it at the end of the function. That way it does not matter if the **lr** gets changed during the execution of the function.

This simple program demonstrates how saving and restoring the **lr** allows a function to call another function and still successfully return to its caller.

```
MOV r0, #5
BL X
MOV r6, r0 ; get my return value, store into r6
end: B end


X   push {lr}
    BL Y
     .....
     pop {lr}
     BX lr


Y   push {lr}
    ADD r0,r0,#1
     .....
     pop {lr}
     BX lr
```