# Lecture #19 – March 26, 2004

# Parameter Passing & Call Mechanisms

## Parameter Passing

We have looked at three assembly-language specific methods for passing parameters into and out of a subroutine; through registers, through main memory, and through the stack. In general, simple subroutines should probably pass parameters through the registers, particularly if, as on the ARM, there are plenty of registers. Subroutines that work on a large, single-occurrence data structure can sometimes get away with directly referencing the data structure in memory. The most general form which supports the largest number of cases (large and small data structures, nested subroutine calls, and recursion) requires that parameters be passed through the stack.

## Calls

High level languages approach the issue of parameter passing from a slightly different angle. At that level it is rare to deal directly with the registers, and while passing parameters through main memory is as easy in a high level language as in assembly language it is usually considered to be poor programming practice. Using the stack for parameter passing allows us to consider some more advanced techniques. Those techniques are named ***call-by-value***, ***call-by-return***, ***call-by-value-return***, and ***call-by-reference***.

Call-by-value-return most closely matches the method described in the previous lecture; a parameter value is pushed onto the stack, the subroutine is called (which usually changes the value on the stack), and upon exit the stack is popped and the return value stored. In ARM assembly language this is done with the following code:

```
LDR R0,Temp
STR R0,[SP,#-4]!
BL  SUB
LDR R0,[SP],#4
STR R0,Temp
```

Call-by-value is a very slight variation that involves pushing the value of the parameter onto the stack, but then discarding it upon return. The subroutine has access to the parameter's value, and may change that value as if it was a local variable, but none of the changes are visible outside of the body of the subroutine. In ARM code the parameter is discarded by changing the stack pointer to its pre-push value, as follows:

```
LDR R0,Temp
STR R0,[SP,#-4]!
BL  SUB
ADD SP,SP,#4
```

In 8088 code call-by-value is partially supported by the hardware. The `RET` (return from subroutine) instruction has a variation which specifies the number of bytes to discard after the return address is popped from the stack. For example, pushing two bytes onto the stack before the call requires a `RET 2` to return and then pop those two bytes. Here is the call and corresponding return code:

```
MOV  AX,Temp              SUB  …
PUSH AX                        …
CALL SUB                       RET 2
```

Call-by-return is also a variation on call-by-value-return in which space is allocated on the stack before the call, but not initialized to any particular value. The subroutine uses that space for computations and the value is returned to the calling routine, where it is stored. Reserving space on the stack requires only a simple modification of the stack pointer before the call, to match the number of bytes popped after the subroutine exits. The ARM code is as follows:

```
SUB SP,SP,#4
BL  SUB
LDR R0,[SP],#4
STR R0,Temp
```

Call-by-reference looks very much like call-by-value in that something is pushed onto the stack and discarded upon return, but what is pushed is the *address* of a variable, not its value. Within the subroutine the address is used to reference the corresponding data structure in main memory. This is a very appropriate mechanism for dealing with very large data structures and dynamic data structures such as linked lists and trees. Here is the ARM code:

```
ADR R0,Temp
STR R0,[SP,#-4]!
BL  SUB
ADD SP,SP,#4
```

Most languages such as Pascal and C support both call-by-value and call-by-reference. Functions return values to their calling routines through call-by-return, and the Ada language has an `OUT` parameter type which roughly corresponds to call-by-return. Few other languages directly support call-by-return. A few implementations of some languages use call-by-value-return instead of call-by-reference, but only for simple variables. In the following Pascal function call, parameter `N` is call-by-value, parameter `R` is call-by-reference, and the value of the function `FX` uses call-by-return.

```
Function FX (N:Integer ; Var R:Integer) : Integer ;
```

The equivalent call in assembly language must first push the value of `N` onto the stack, followed by the address of `R`, and finally the space required for the return value. The ARM code is shown below:

```
LDR R0,N
STR R0,[SP,#-4]!          Push N
ADR R0,R
STR R0,[SP,#-4]!          Push Address of R
SUB SP,SP,#4              "Push" Return Space
BL  FX                    Call Function FX
LDR R0,[SP],#4            Pop Return Value
STR R0,Result
ADD SP,SP,#8              Discard R and N
```

On the ARM, a pop instruction such as `LDR R0,[SP],#4` loads the desired register from the top of the stack, and then updates the stack pointer by adding 4 to its value *after the load is complete*. Since the next instruction that deals with the stack discards the top two items by adding 8 more to the stack pointer, those two instructions can be combined into one by loading the register and adding 12 to the stack pointer. Popping the return value and discarding `R` and `N` is done by the fragment:

```
LDR R0,[SP],#12
STR R0,Result
```

This optimization should not be implemented until the function is complete, tested, and the definitions of all the parameters are known and stable. Discarding the wrong number of bytes from the stack is usually disastrous.

We finally have all the tools we need for writing general purpose subroutines in assembly language. In the next lecture we will put all these tools together to create subroutines which are recursive (call themselves).