Lecture #11 – February 23, 2004

Subroutines, PrintHex, and Transparency

Recall from the previous lecture how we created the code for the `PrintHex` subroutine. In this version of the code there is a "placeholder" of question marks just after the beginning of the subroutine:

```
PrintHex  STR   R0,SaveR0
          ???                  placeholder (new)
          CMP   R0,#9
          ADDGT R0,R0,#'A'-10
          ADDLE R0,R0,#'0'
          SWI   &0
          LDR   R0,SaveR0
          MOV   PC,LR
```

The placeholder is where we will examine the question of what to do when the input argument in `R0` is outside the range of 0 through 15. In the previous version, we *assumed* that the value in `R0` was in the correct range. Assumptions like this are generally pathways for errors to creep into a program. What we wish to do here is either detect the presence of an error or insure that no error can possibly occur. If we choose to detect the error, we must also choose whether to correct the error, report the error, ignore the error (unlikely), or exit the subroutine without printing anything. We will examine first the case of looking for an error and exiting the subroutine if one is found.

This first approach is to insert "guard code" in place of the question marks that compares the input value in `R0` both to 15 and to 0, and exits the subroutine when an out-of-range value is found. In high-level pseudo code, our `PrintHex` task is roughly as follows (liberally mixing high-level variables and ARM register names):

```
If (R0 >= 0) And (R0 <= 15) Then
     If R0 > 9 Then
          CH := 'A' + R0 - 10
     Else
          CH := '0' + R0 ;
     Print (CH) ;
Else
     Flag_an_Error ;
```

The code fragment for doing the test is:

```
CMP  R0,#15    If R0>15 Then Goto Done
BGT  Done
CMP  R0,#0     If R0<0 Then Goto Done
BLT  Done
```

The phrase `Flag_an_Error` in our pseudo code indicates where the action should be taken when the error is detected, and the corresponding ARM code will start with the label `Done` on that line. If no action is to be taken, `Done` is placed on the line that restores `R0` just before exiting the subroutine.

If you examine the guard code, you can determine that because of the way it is written we are assuming that the input value in `R0` is treated as a signed integer. The first comparison checks to see if `R0` is in the range from 16 up through 2^{31}-1 (all supported positive numbers greater than 15), and the second comparison checks to see if `R0` is in the range from -2^{31} up through –1 (all supported negative numbers). If we assume *for the purposes of the guard code* that the value in `R0` is unsigned (whether or not it really is), we can replace the guard code with the following fragment:

```
CMP  R0,#15
BHI  Done       Unsigned greater-than
```

The `HI` condition on the branch instruction treats the result of the comparison as unsigned, and branches if the value is greater than 15. Since there are no negative numbers in unsigned integer representations, this effectively eliminates everything illegal in one step, all numbers in the range from 16 through 2^{32}-1. It doesn't matter if we originally thought that `R0` contained a signed integer; all negatives map onto large positives and are eliminated at the same time as the "normal" (too large) positive values.

This shifting of representations occurs very frequently in assembly language. Since every data value is fundamentally nothing more than a package of bits we can treat each package in the most convenient representation for each specific task, and in multiple representations, simultaneously.

Instead of detecting an error, whether or not we decide to do anything about it, we can take the approach of insuring that no error can possibly occur. If the value in `R0` must be between 0 and 15, we could extract the remainder of dividing `R0` by 16. This operation, `R0 := R0 mod 16`, always guarantees that the argument is in the range from 0 through 15. Unfortunately, there is no integer division instruction on the ARM, and without division it is difficult to perform a general-purpose "mod" function.

As it turns out, we need neither a general-purpose division nor a general-purpose mod capability. The divisor, 16, is a power of two, so division can be performed with a simple right-shift. Our first attempt might be to shift the value in `R0` to the right by four bits, then back to the left by four bits (thus setting the rightmost four bits to zero), and subtracting that result from the original value in `R0`. We'll need to burn an extra register to do this (and must explicitly save and restore its value in the subroutine):

```
MOV  R1,R0,LSR #4    xxx…xxxyyyy → 0000xxx…xxx
MOV  R1,R1,LSL #4    0000xxx…xxx → xxx…xxx0000
SUB  R0,R0,R1        xxx…xxxyyyy - xxx…xxx0000 =
                                          000…000yyyy
```

The result in R0 is 28 zero-bits followed by the rightmost four bits of the original value (000…000yyyy). With four bits you can represent any value between 0 and 15, which exactly matches the requirements of the PrintHex subroutine.

To extract the remainder from any division by a power of two you need only change the amount of the shift: use #4 for computing the mod 16 of a number (since 2^4=16), use #5 for computing mod 32 (since 2^5=32), and so on. The smallest value that remains after a mod operation is always zero. For a shift factor of N bits (i.e., mod 2^N) the largest value that can remain will consist of 1-bits in the low order N bits of the number and 0-bits everywhere else. Thus, the result of a mod 2^N operation is always a number between 0 and 2^N-1, which always fits exactly into the lowest N bits of the word.

It is this last observation that allows us to simplify our code. If all we want are the low-order N bits of a word, with the upper bits set to zero, our task collapses to using a single AND instruction with the correct bit pattern as the constant operand. We need only AND our target register with 2^N-1 to ***mask off*** (clear to zero) all but the low-order N bits. Our single instruction to insure that R0 contains no number outside the range 0 through 15 is as follows:

```
AND  R0,R0,#15
```

Note that we cannot use this technique to compute the mod of any arbitrary number such as 12, 20, or 37, because these numbers are not one less than a power of two. We get away with using 15 because it is 2^4-1. Acceptable numbers for this technique include 1 (one bit), 3 (two bits), 7 (three bits), 15 (four bits), 31 (five bits), etc. Should we use a number such as 26, which has binary value 11010, we end up preserving bits 1, 3, and 4 in the result, but we clear bits 0 and 2 along with all bits to the left of bit 4.

Which of the guard code methods we decide to use depends largely on the context of the code that calls PrintHex. For example, a subroutine called PrintWord that prints out all 32 bits of a word as eight hexadecimal digits must call PrintHex eight times. If we choose to print out R0, we must rotate it correctly before each call to PrintHex to align the desired four bits properly in the low-order part of the word. In this case it is easiest to leave the rest of the bits in the word unmodified, so PrintHex should use the AND technique as its guard code. Here is the final code for PrintHex:

```
PrintHex  STR   R0,PrintHexSaveR0
          AND   R0,R0,#15
          CMP   R0,#9
          ADDGT R0,R0,#'A'-10
          ADDLE R0,R0,#'0'
          SWI   &0
          LDR   R0,PrintHexSaveR0
          MOV   PC,LR
```

Here is the code for `PrintWord` (since there is no rotate-left option on a `MOV` instruction, we can use `ROR #28` to simulate rotating left by 4 bits):

```
PrintWord STR   LR,PrintWordSaveLR
          STR   R0,PrintWordSaveR0
          STR   R1,PrintWordSaveR1

          MOV   R1,#8
Loop1     MOV   R0,R0,ROR #28       (i.e., ROL #4)
          BL    PrintHex
          SUBS  R1,R1,#1
          BNE   Loop1

          LDR   R1,PrintWordSaveR1
          LDR   R0,PrintWordSaveR0
          LDR   PC,PrintWordSaveLR
```

Notice that we had to save `LR` in `PrintWord` because it must call `PrintHex`, but `PrintHex` need not save `LR` because it never calls any other routine. Also notice that we save `R0` (and `R1` and `LR`) to memory locations specifically associated with `PrintHex` and `PrintWord`; we cannot save `R0` to a location called `SaveR0` in both subroutines because that allows for the subroutines to communicate through a "back-channel" pathway. We might get lucky and end up with the same value stored by both routines, as will happen in this example (maybe?), but it is nearly always a very bad idea for two subroutines to share storage for preserving their registers. Eventually you will get a case where one subroutine retrieves a different value from what it thought it stored because of a side effect from another routine. These errors are extremely difficult to find.

In `PrintWord` you can make the case that performing `MOV R0,R0,ROR #28` (which as we've seen is the same as `ROL #4`) eight times ends up preserving the original value of `R0`, and so saving and restoring its value is not really necessary. Yes, it will end up with the same value as it started with if you've written the code correctly, and if this is the case you can skip saving and restoring `R0`. *How much do you trust your code?* It is almost always dangerous to continuously modify a register and hope you do it "enough" times to restore its original value. You should always explicitly save and restore any register that gets modified by a subroutine, even if you think it'll naturally wind up with its own starting value at the end of the computation. By explicitly saving and restoring its value you isolate any errors to that one subroutine only, and prevent those errors from propagating back to the calling routine.

The act of saving and restoring all registers in a subroutine that are modified (except for those needed to pass values back to the calling code) is called ***transparency***. A fully transparent subroutine saves and restores *everything* it changes. The calling code must not encounter any unexpected side effects from calling the subroutine. Any register not used by the subroutine need not be saved, but if you must use a temporary register to hold the result of a calculation, save it first and restore it afterwards.

As we will see later, the use of a stack greatly simplifies transparency in subroutines, but even without a stack transparency is still simple to implement by using primary memory (as long as recursion is not required).

Always, always, always write completely transparent assembly language subroutines!