Lecture #10 – February 20, 2004

ASCII and a "PrintHex" Subroutine

In this lecture we will examine a programming problem that appears frequently in assembly language projects. The task can be stated very simply: a numeric value between 0 and 15 is passed in to an ARM subroutine through register `R0`, and the subroutine must print out the corresponding hexadecimal character. In addition, any changes to register `R0` must be "undone" before exit so that the calling routine sees no ***side effects*** from calling the subroutine.

In the widely used ***ASCII*** character set (***American Standard Code for Information Interchange***) there are 128 defined characters, with values from 0 through 127. Values 0 through 31 represent ***control characters***, which historically cause actions to be performed instead of actual characters to be printed. For example, character 7 rings the bell. On old mechanical teleprinters character 13 caused the print carriage to physically move to the beginning of the current line; hence it was called the ***carriage return*** or ***CR*** character. Similarly, character 10 caused the paper to be rolled up to the next line, which is why it was called the ***line feed*** or ***LF*** character. With today's electronic displays the terms carriage return and line feed are obsolete, but those terms persist in how lines of text are separated in a file. On the PC, lines are separated by both a CR and an LF, while on the Mac lines are separated by just a CR, and on UNIX systems lines are separated by just a LF. (This is why FTP programs have an ASCII transfer mode; the FTP program is given permission to change the line breaks of a text file to match the requirements of the target computer system.)

ASCII values 32 through 63 represent "special" characters (32 is the space, 33 the exclamation point, and so on). Values 64 through 95 contain the upper case Roman alphabet, and values 96 through 127 contain the equivalent lower case letters (each group of letters also includes as a small number of special characters). All digit characters are in an unbroken sequence from character `0` through character `9`: the `0` is at index 48, `1` is at index 49, and so on. Similarly, the Roman alphabets are also in unbroken sequences. Capital letter `A` is at index 65, `B` is at index 66, up through `Z` at index 90. Lower case letter `a` is at index 97, `b` is at index 98, and on up through `z` at index 122. We can depend on these sequences for figuring out how to write our hexadecimal print subroutine. (Not all character sets have these nice properties; the EBCDIC character set used on old IBM mainframe computers did not have the letters in nice linear sequential order.)

In our routine, to be called `PrintHex`, input values from 0 through 9 must map onto characters `0` through `9`, and values from 10 through 15 must map onto characters `A` through `F`. One solution is to create an array of 16 characters, and index into that array with the input value to find the correct corresponding character. In non-ASCII character sets this approach may be the most appropriate technique to implement. For ASCII, all we need do is determine if the input value is in the range 0 to 9 or in the range 10 to 15, and *compute* the correct character based on the starting index of the appropriate sequence.

In a high-level language pseudo code, this process can be roughed out as follows:

```
Procedure PrintHex (C:Nybble)
     If C > 9 Then
          CH := C + 'A' - 10
     Else
          CH := C + '0'
     Print (CH)
```

In this pseudo code, the data type `Nybble` represents a number between 0 and 15 (we are not going to check for errors in the input value at this time). Notice that we are mixing numeric and character data types in the computations, but in assembly language this is a very natural thing to do.

Our first attempt at converting this into ARM code is to directly create an If-Then-Else structure that matches the pseudo code, and do the appropriate character conversion in each branch. In the following solution, the numeric value in `R0` is converted into the equivalent character, and then the character is printed by software interrupt `0` (which in some ARM software prints the character passed to it though `R0`).

```
PrintHex  CMP   R0,#9              If R0 > 9 …
          BGT   DoLetter
          ADD   R0,R0,#'0'         Else block
          B     PrintIt
DoLetter  ADD   R0,R0,#'A'-10      Then block
PrintIt   SWI   &0                 Print (R0)
          MOV   PC,LR              Return
```

This code will work, but it is overly complicated and it trashes the contents of register `R0` (the value passed back to the calling routine is not the same as the value passed in). Since both the Then-block and the Else-block contain only one instruction each, we can take advantage of the ARM's ability to embed the conditional test directly into those instructions. In the following version, the branch instructions are gone, and by controlling the two `ADD` instructions with *opposite* condition tests only one of the two instructions will ever be executed.

```
PrintHex  CMP   R0,#9
          ADDLE R0,R0,#'0'
          ADDGT R0,R0,#'A'-10
          SWI   &0
          MOV   PC,LR
```

This code is very fast, particularly since the flow of control is in a straight line. As each instruction is being executed the processor is simultaneously fetching and decoding subsequent instructions in the sequence; taking a branch requires that any such preliminary work be discarded and restarted at the destination of the branch. If there are no branches, there can be no hiccups in the execution sequence.

Finally, we must save and restore the contents of `R0`. Since we are not using the stack (yet), we must declare a location in primary memory to hold the value of `R0` while we are in the subroutine. We must also explicitly save the contents of `R0` into that location at the start of the subroutine, and restore `R0` from memory just before exiting the subroutine. The last, complete version of the `PrintHex` subroutine is as follows:

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

SaveR0    DCD   0
```

Every register modified by the subroutine is restored to its initial value before the subroutine exits, the code contains no branches, and without using an array of characters the code is as small and as fast as possible.

The `SaveR0 DCD 0` command is a directive to the assembler to reserve a full 32-bit word in primary memory, referenced by the symbol `SaveR0`, and with zero as its initial value.

One final comment about the notation used in these programs is necessary. The instruction `ADDGT R0,R0,#'A'-10` contains the interesting expression `'A'-10`. This expression is evaluated by the assembler at *translation time*, not at execution time. Since the character `'A'` in ASCII has the value 65, the instruction is converted by the assembler into the equivalent instruction `ADDGT R0,R0,#55`. While this last form is completely legal and will perform the correct computation, the `'A'-10` form is preferred by most assembly language programmers because it most closely indicates the meaning of the conversion task than does the "magic number" 55.