Lecture #12 – February 25, 2004

Ugly Programming Tricks

In this lecture we will visit a number of "tricks" available in assembly language not normally seen in high-level languages. Some of the techniques are valid, legitimate approaches to solving problems, but others are of dubious value and are to be used (if at all) in rare, limited and extreme circumstances. In many cases these tricks are not necessary unless program memory is unusually tight, execution speed is critical, or both.

Mixing Code and Data

Up to this point, subroutine design has been presented in a style where any data variables local to a subroutine are located in primary memory very near to the subroutine's code. For example, both subroutines in the example below have their own local storage (for saving the contents of `R0`) directly following their corresponding return instructions:

```
;---------------------------------

Sub1            STR  R0,Sub1SaveR0
                …
                LDR  R0,Sub1SaveR0
                MOV  PC,LR

Sub1SaveR0      DCD  0

;---------------------------------

Sub2            STR  R0,Sub2SaveR0
                …
                LDR  R0,Sub2SaveR0
                MOV  PC,LR

Sub2SaveR0      DCD  0

;---------------------------------
```

The word in memory which is reserved for `Sub1SaveR0` directly follows the `MOV PC,LR` instruction of `Sub1` and directly precedes the `STR R0,Sub2SaveR0` instruction of `Sub2`. Storing a value into `Sub1SaveR0` changes that single word of primary memory located *between* the code blocks of the two subroutines. Note that the `DCD` directives need not initialize their corresponding data variables to zero, but instead could place in them arbitrary numeric values, or even leave those locations uninitialized. I find it convenient to start all variables at a known and consistent state.

One advantage of this approach is that the data values are located close to where they are used. The assembly language programmer does not have to hunt all over a large program to locate the data for a particular routine.

There is another, less obvious, advantage of keeping memory variables near their associated code. Memory reference instructions on many computers have a limited addressing range. For example, the `LDR` and `STR` instructions on the ARM have only 12 bits of the instruction available to contain an address value, which is treated as an offset relative to one of the registers (such as the program counter). One more bit of the instruction determines if the offset is added to or subtracted from the given ***base register***, and so no memory location can be addressed *directly*, without special techniques, if it is more than ±4095 bytes from the current value of the program counter. Keeping data near the associated code reduces the possibility of a memory reference being out of range of its instruction. We will explore techniques later on of using other base registers for referencing memory variables at arbitrary addresses.

A major disadvantage of this technique is that the code cannot be placed into ***ROM*** (***read only memory***). Small embedded systems most likely do not have disk drives, and must contain all executable program code in ROM. Data variables must be located in RAM. No system will interleave RAM and ROM at the fine detail level required by individual subroutines, but only at much larger scales instead (on the order of kilobytes or megabytes in each block).

Another disadvantage appears when storing arrays in the data areas between code blocks. Indexing beyond the end of an array is a very common problem, particularly in assembly language. If the next bytes of primary memory beyond the end of an array contain instructions, an array overrun error will destroy executable code (or worse, change the code into a different set of instructions with unpredictable behavior).

Keeping data separate from code reduces or eliminates this kind of problem. An array overrun error is still a serious problem and is to be avoided at all costs, but in such cases the code remains unchanged and has a remote chance of recovering. Operating systems have an easier time preventing memory reference errors when code and data are kept separate as well. This effectively enforces a "ROM/RAM" distinction for individual programs at the OS level instead of at the hardware level.

Self-Modifying Code

Occasionally, we *do* want to write over instructions in the code section of a program. This is rarely a good idea, and is to be used only in extreme circumstances, for it is nearly impossible to debug a ***self-modifying*** program.

For example, we might have a program structure that contains an IF-THEN-ELSE block inside a loop of some kind. In a program such as this, the IF test is performed during every iteration of the loop. For programs where the IF test *does not change* its result as a consequence of executing the loop code, the IF test is static and will always guide the flow of the program through the same branch of the IF-THEN-ELSE.

Executing the same test over and over again when we already know the answer is very inefficient. Normally we would optimize the code for that case by *inverting* the order of the components and placing a copy of the loop code inside each branch of the IF-THEN-ELSE block. This is shown in the following high-level pseudo code:

Original Code

```
Loop
     block #1

     If Flag Then
          block #2
     Else
          block #3
     EndIf

     block #4
EndLoop
```

Speed Optimized Code

```
If Flag Then
     Loop
          block #1
          block #2
          block #4
     EndLoop
Else
     Loop
          block #1
          block #3
          block #4
     EndLoop
EndIf
```

Notice that we have traded space for speed. The version on the right takes up more memory space than the version on the left because it contains duplicate copies of the code in blocks #1 and #4, but it runs much faster because the IF test was performed only once instead of in every pass through the loop. (None of the code in blocks #1 through #4 may change the value of `Flag`; if they do then this optimization technique cannot be used.)

How can we have both small program size and fast execution speed? The self-modifying approach is to set up the loop with a “slot” of bytes as a placeholder region big enough to contain the larger of blocks #2 or #3, then on the result of the IF test copy the appropriate section of code into the slot before executing the loop. This is shown below:

Self Modifying Code

```
If Flag Then
     Copy block #2 code into block slot
Else
     Copy block #3 code into block slot
EndIf

Loop
     block #1
     block slot               placeholder region
     block #4
EndLoop
```

Each block of code occurs only once, the IF test is performed only once, and the loop code runs as fast as possible. Compared to the earlier examples, our new result is essentially no larger than the original code (on the left), and no slower than the speed optimized code (on the right). This approach can be attractive when bytes are tight and shaving microseconds is crucial.

Here is an example of how that might be done in ARM assembly language. Say that inside our loop we will need to perform either an `ADD` or a `SUB` of 1 with the `R0` register, but we don't know which one ahead of time. Templates for those instructions are defined at labels `ThenCode` and `ElseCode`; the templates are never actually executed at those locations, but instead are treated as data by the two `LDR` instructions at the top of the code block. Whichever, then, is the appropriate instruction (determined by a comparison which sets or clears one of the status flags, `Z` in this case) is stored as data into the `BlockSlot` placeholder location in the middle of the instruction stream! When the program flow reaches the `BlockSlot` location the processor will simply execute whatever instruction it finds there.

```
          <perform some comparison here that
           affects the Z status bit, such as
           asking about the variable "Flag">

          LDREQ R0,ThenCode   Either get this template
          LDRNE R0,ElseCode   or this template, then
          STR   R0,BlockSlot  modify the code below.

Loop1     …
          …
          …
BlockSlot DCD   0             placeholder region
          …
          …
          B     Loop1

ThenCode  ADD   R0,R0,#1      instruction template #1
ElseCode  SUB   R0,R0,#1      instruction template #2
```

Obviously, this technique cannot be used at all if the executable code is placed into ROM. The code *must* be executing from RAM so that the self-modifying portion can overwrite one instruction with another.

Code templates may contain multiple instructions; the placeholder region must be large enough to accommodate the largest possible template. Any template smaller than the reserved placeholder region must be padded out with `NOP` (no-operation) instructions to insure all bytes in the placeholder are properly defined. If the memory reserved for the placeholder region is accidentally smaller than the size of the templates, then copying any template into the placeholder will overwrite instructions *after* the end of the region. Looking at the source code for clues to the program's behavior will then prove fruitless, as the executing code bears no resemblance to what the programmer actually wrote! This problem is exacerbated on processors with variable length instructions: not only are the instructions different from what the programmer expects, but what was a *data field* of an old instruction might become the *opcode* for a new instruction, depending on how bytes are overwritten!

It is a rare circumstance that requires self-modifying code, and it is to be avoided at (nearly) all costs. Misbehaving self-modifying code is incredibly difficult to debug. The benefits gained by using it are rarely counterbalanced by the frustration encountered in getting it to work correctly. Know about it, keep the technique in your bag of tricks, but never, ever use it casually! If you *must* use self-modifying code, document the hell out of it! As the bumper sticker says, "Meddle not in the affairs of dragons, for you are crunchy and good with ketchup!" Here there be dragons!

Patching Binary Programs

A friend of mine once worked on a large assembly language project for a Z-80 microprocessor chip. The Z-80 is a Zilog-made, enhanced version of the Intel 8080, which was a predecessor of the 8088. As the deadline for project completion drew near, she realized that at one point in her code she had used the wrong instruction opcode; she had either used `ADD` instead of `SUB`, or `ADC` instead of `ADD`, or some similar mistake.

Rather than changing the source code, then reassembling, relinking, and reloading the program, she simply *edited the executable code* in a hexadecimal byte editor, replacing the errant opcode with the hexadecimal value of the correct opcode. The program then worked perfectly. When she told her computer-scientist husband of this, he simply replied, "Now you have known sin!"

The advantage of this approach is quickly overwhelmed by its disadvantages. It is very easy to change a byte at the wrong address, or to change the correct byte to an incorrect value. If the hexadecimal byte editor allows for the insertion or deletion of bytes in the code stream instead of simple replacement, many relative offsets (such as those in branch instructions) will now point to the wrong address.

There is also a mismatch between the source code and the executable; the first no longer assembles into an exact copy of the second. To be completely intellectually honest, any patches of this kind must be carefully documented, and then the source code must be changed correspondingly, reassembled, and the result compared byte-for-byte with the byte-edited-executable version to verify that they are identical. The more binary patches that are made they harder this verification step becomes.

Siamese Subroutines

This is a technique that I actually use in my assembly language programs, but there are limited circumstances where it is actually useful. If you have several subroutines that are identical except for a very small portion of code (such as the loading of subroutine-specific constants), then the total amount of code can be reduced by creating ***Siamese subroutines*** that share code bodies. Siamese subroutines have multiple entry points, but only one exit point. Consider the following four subroutines, which are identical after the `MOV R0,#const` instructions:

```
Sub1 STR  R0,SaveR0       Sub2 STR  R0,SaveR0
     MOV  R0,#const1           MOV  R0,#const2
     …                         …
     …                         …
     …                         …
     LDR  R0,SaveR0            LDR  R0,SaveR0
     MOV  PC,LR                MOV  PC,LR


Sub3 STR  R0,SaveR0       Sub4 STR  R0,SaveR0
     MOV  R0,#const3           MOV  R0,#const4
     …                         …
     …                         …
     …                         …
     LDR  R0,SaveR0            LDR  R0,SaveR0
     MOV  PC,LR                MOV  PC,LR
```

In a set of Siamese subroutines, the ***prolog code*** from each entry point up through the last unique section is written explicitly, and after each prolog the code branches to the section common to all. This is shown as follows:

```
Sub1 STR  R0,SaveR0      prolog for Sub1
     MOV  R0,#const1     prolog for Sub1
     B    SubGo          goto common section

Sub2 STR  R0,SaveR0      prolog for Sub2
     MOV  R0,#const2     prolog for Sub2
     B    SubGo          goto common section

Sub3 STR  R0,SaveR0      prolog for Sub3
     MOV  R0,#const3     prolog for Sub3
     B    SubGo          goto common section

Sub4 STR  R0,SaveR0      prolog for Sub4
     MOV  R0,#const4     prolog for Sub4
;    B    SubGo          commented out

SubGo                    branch target
     …                   code common to all
     …                   code common to all
     …                   code common to all
     LDR  R0,SaveR0      code common to all
     MOV  PC,LR          code common to all
```

Notice that in the last prolog the `B` (unconditional branch) instruction occurs in the code one instruction before the common target of all such branches. Executing the branch wouldn't actually change the program counter, so the branch isn't necessary. To keep the last prolog consistent with all others the branch instruction is written into the source code, but is then commented out so it will not take up space in the executable or waste any execution time.

For Siamese subroutines to be effective the prolog code must be short and simple, and each must be carefully crafted to operate identically to all other prologs. When done correctly, Siamese subroutines can save a considerable amount of memory space by avoiding the duplication of common code fragments, and execute faster than a series of small parameterized drivers for a general purpose subroutine.

Conclusions

Sometimes knowing how to do things the wrong way helps us do things correctly! The techniques discussed here are very important for the average assembly language programmer to know, and have been used in many successful (and unsuccessful) assembly language projects in the past, but they may be of limited utility to "modern" programmers.