Lecture #7 – February 11, 2004

A Worked Problem for the ARM

In this lecture we are going to look at a single problem in detail and from many angles. By doing so, we will explore basic concepts of programming in ARM assembly language. We will also explore methods of reworking code so that it requires fewer instructions, uses fewer registers, and/or runs faster. The sample problem is very simple to describe: given some positive integer `N`, we want to compute the sum of all integers from 1 up to and including `N`. In Pascal, this would be written as:

```
Total := 0 ;
For I := 1 To N Do
     Total := Total + I ;
```

An equivalent form which is more easily converted into assembly language is written as follows:

```
Total := 0 ;
I := 1 ;
While (I <= N) Do
     Begin
          Total := Total + I ;
          I := I + 1 ;
     End ;
```

If we act as a "stupid compiler" by mechanically translating each statement into assembly language independently of all other statements, we will get a working but very inefficient version of the program. The goal of this lecture is to demonstrate stupid, mediocre, and smart translations of this code into ARM assembly language.

For the first (stupid) translation we will assume that integer variables `Total`, `I`, and `N` have been allocated somewhere in memory, and that *somehow* a positive value got entered into `N`. (Remember that the `CMP` instruction automatically modifies the status bits in the ***program status register*** without requiring an `S` suffix on the instruction.) Examine the code below to see if you can determine where the code can be streamlined.

```
     MOV  R0,#0          Total := 0
     STR  R0,Total
--------------------
     MOV  R0,#1          I := 1
     STR  R0,I
--------------------
Loop LDR  R0,I           While I <= N Do
     LDR  R1,N
     CMP  R0,R1
     BGT  Done
--------------------
     LDR  R0,Total            Total := Total + I
     LDR  R1,I
     ADD  R0,R0,R1
     STR  R0,Total
--------------------
     LDR  R0,I                I := I + 1
     ADD  R0,R0,#1
     STR  R0,I
--------------------
     B    Loop           End_While
Done …
--------------------
```

The major problem with this code is that it takes no advantage from the fact that many values loaded from or stored into memory are already present in one or more of the registers. If, instead, we write the code so that values are left in the registers and stored into memory only after the final results are computed, then most of the unnecessary data movement will be eliminated. In this version, `Total` is kept in `R0`, `N` in `R1`, and `I` in `R2`:

```
     MOV  R0,#0
     MOV  R2,#1
     LDR  R1,N
Loop CMP  R2,R1
     BGT  Done
     ADD  R0,R0,R2
     ADD  R2,R2,#1
     B    Loop
Done STR  R0,Total
```

One problem with this code is that there are two branches in the loop; one conditional and the other unconditional. If we can guarantee that `N` is greater than zero as a precondition for running this code, then checking for the exit condition at the top of the loop is unnecessary during the first pass. By rewriting the code as a repeat-loop instead of a while-loop, the exit condition moves to the bottom of the loop.

The repeat-loop code expressed in high-level form is as follows:

```
Total := 0 ;
I := 1 ;
Repeat
     Total := Total + I ;
     I := I + 1 ;
Until I > N ;
```

In assembly language, this turns into:

```
     MOV  R0,#0
     MOV  R2,#1
     LDR  R1,N
Loop ADD  R0,R0,R2
     ADD  R2,R2,#1
     CMP  R2,R1
     BLE  Loop
     STR  R0,Total
```

This version has a single conditional branch and four instructions inside the loop instead of five, so it will run a little faster than the previous version. In general, writing repeat-loops in assembly code is more efficient and simpler than writing while-loops.

By looking at the task we are performing, we notice that it does not matter to the algorithm if we start counting at 1 and count up to `N`, or start at `N` and count down to 1. By counting down, we can exploit the program status register flags (the `Z` bit in particular) to detect when the loop is to terminate. In addition, we need not have a register reserved specifically for counting; we can initialize one register to the initial value of `N` and count it down to zero (i.e., there is no need for an "`I`" variable). The resulting code is as tight as it is possible to be:

```
     MOV  R0,#0
     LDR  R1,N
Loop ADD  R0,R0,R1
     SUBS R1,R1,#1
     BNE  Loop
     STR  R0,Total
```

Notice that there is an `S` suffix on the `SUB` instruction. The suffix causes the result of the subtraction to modify the status bits in the program status register. The last valid value in `R1` is 1; when `R1` is decremented to zero the `Z` bit is set and the `BNE` branch fails. Omitting the `S` suffix prevents the status bits from being ever modified, and the loop will run forever! (Adding an `S` suffix to the `ADD` instruction is unnecessary, but doing so does not "hurt" the code other than making it somewhat more difficult to read. Any flags modified by an `ADDS` instruction are completely undone by the `SUBS` instruction on the very next line.)

While the code that was generated in the previous step is as good as it can get for the problem *as it is stated*, the presence of the loop always guarantees that this is an O(N) algorithm. The time to solve the problem grows linearly as N increases. By going back to the original problem and examining it closely, we realize that we can recast the problem as an O(1) algorithm. Mathematically, adding all integers between 1 and N can be computed directly by the equation N×(N+1)÷2. In the equivalent code that follows, every instruction is executed exactly once:

```
MOV  R0,N               R0 := N
ADD  R1,R0,#1           R1 := N+1
MUL  R2,R0,R1           R2 := N*(N+1)
MOV  R2,R2,LSR #1       R2 := N*(N+1) Div 2
STR  R2,Total
```

There is no integer division instruction on the ARM, but most of the arithmetic instructions allow for the last operand to be shifted left or right before it is used. In the MOV instruction, the value in R2 is shifted right by one bit (i.e., divided by 2) before it is moved back into R2.

(As an aside, the shifter can be used to do quick multiplication by special constants without using the MUL instruction. For example ADD R0,R0,R0,LSL #2 multiplies R0 by 5 in place by adding R0 to R0×4, e.g., R0 shifted left 2 places. The shifter will be covered in more detail in a later lecture.)

In this section we've used a simple sample problem to show how to write basic ARM assembly language, optimize that code to reduce the number of registers, and reduce the number of instructions executed inside loops. By recognizing that a different computational method generates the same desired result we may be able to write even better assembly code.