Lecture #15 – March 3, 2004

A Worked Problem in Floating Point

In this lecture we will use the floating point capabilities of the ARM to compute the square root of a single-precision number in register `F0` and put the result into `F1`. A short list of floating point opcodes available on the ARM is as follows (there are a number of other such instructions, including sine, cosine, tangent, common and natural logarithms, arcsine, etc., but those aren't germane to the problem at hand):

```
ADF  Add                            MUF  Multiply
SUF  Subtract                       RSF  Reverse Subtract
DVS  Divide                         RDF  Reverse Divide
POW  Power                          RPW  Reverse Power
ABS  Absolute Value                 SQT  Square Root
CMF  Compare
FLT  Float an Integer Register
FIX  Truncate a Floating Register
```

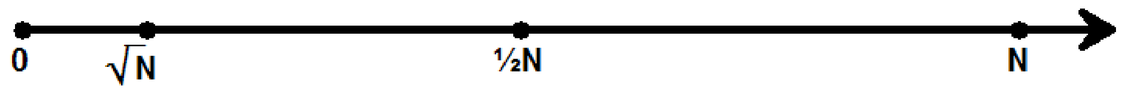
Remember that each of the instructions above requires a suffix of `S` or `D` to determine single or double precision arithmetic, except for `CMF`, `FLT`, and `FIX`. For all the examples here we will use single precision.

There are several approaches to computing a square root, from simple to complex. The simplest is to use a built-in square root instruction or a built-in power instruction, such as either of the following:

```
          SQTS F1,F0            F1 := SquareRoot(F0)
     -or-
          POWS F1,F0,#0.5       F1 := F0^0.5
```

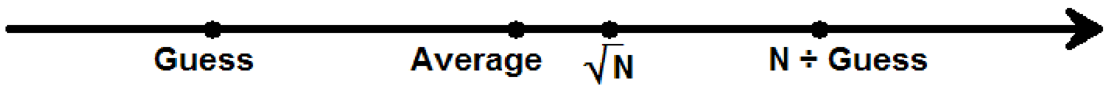
Newton-Raphson Square Root

While those approaches are generally preferred, they are not very instructive in how to compute a square root, or even how to use the floating point instructions effectively. Those special instructions may not be even present on some floating point processors. What we will do here is compute square roots using an iterative technique called ***Newton-Raphson iteration***. Examine the figure below, which shows a number line containing a sample value of N, ½N, and the square root of N.



For any value of N greater than 1, the square root of N is between 1 and N. (Square roots converge to 1; the square root of any value N between 0 and 1 is greater than N but less than 1. We're going to ignore that case here, but the arguments are pretty much the same.)

If we divide N by its square root we get exactly the square root as our quotient. (Obviously.) Thus, if we divide N by a number *smaller* than the square root the quotient is *larger*, and vice versa. Whatever number we "guess" as a divisor, both it and its quotient will *bracket* the true value of the square root. The average of those two numbers must therefore be closer to the true square root value than one of them, and that average can be used as a better "guess" value for the next iteration. This is shown in the following drawing, where it is obvious that the average is closer to the true square root than one of the two bracketing values (in this example it is closer than both bracket values, but the only guarantee is that it will be closer than one of them).



The initial value of the guess can be pretty critical to the running time of this algorithm, but for now we will assume that half of the initial value is "good enough" to get it going. Later on we will look at a technique for generating a better initial guess. In high level pseudo code, the technique is written as follows:

```
Guess := N / 2
Loop
     Next := N / Guess
     If |Guess - Next| < ε Then Exit
     Guess := (Guess + Next) / 2
EndLoop
```

The algorithm stops when the absolute value of the difference between the current guess (`Guess`) and the next guess (`Next`) is below some numerical threshold (ε). The selection of the proper value of ε determines the precision of the result, and is usually a tiny positive number such as 1.0×2^{-23} for single precision (which has 23 bits of mantissa).

Newton-Raphson in ARM Assembly Language

Dividing N in half to generate the initial guess warrants some discussion here. On the ARM, either the instruction `DVFS F1,F0,#2.0` (divide by 2) or the instruction `MUFS F1,F0,#0.5` (multiply by ½) will work. Remember that there are only eight constants that can be used directly in floating point instructions on the ARM: those constants are 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 10.0, and 0.5 (a rather good selection, I think). Given a choice between two mathematically equivalent instructions you should generally select the fastest one of the pair, and multiplication is nearly always faster than division.

When we convert the high-level pseudo code to ARM assembly language, we will keep N in register `F0`, the guess in `F1`, the next guess in `F2`, and the threshold ε in `F3`. The assembly language fragment follows.

```
          MUFS      F1,F0,#0.5     Guess := N / 2
Loop      DVFS      F2,F0,F1       Next  := N / Guess
          CMF       F1,F2          If Guess >= Next…
          SUFGES    F4,F1,F2        Then F4 := Guess - Next
          SUFLTS    F4,F2,F1        Else F4 := Next - Guess
          CMF       F3,F4
          BGT       Done           Exit If ε > F4
          ADFS      F1,F1,F2
          MUFS      F1,F1,#0.5     Guess := (Guess+Next)/2
          B         Loop
Done      …
```

Notice that in the construction of `F4`, which is to contain the absolute value of `Guess - Next`, we append conditional execution to the instructions so that we are always subtracting a smaller value from a larger value. This three-instruction block (the compare and the two subtracts) can be replaced by a shorter sequence if we use the special absolute value instruction on the ARM:

```
SUFS      F4,F1,F2
ABSS      F4,F4
```

A rewrite of our original pseudo code also results in a slight improvement in layout. By reordering the instructions to force one computation of the new guess value, the exit condition moves to the bottom of the loop. Repeat-loops are always the easiest and simplest conditional loops to write in assembly language:

```
Guess := N / 2
Repeat
     Next := N / Guess
     Guess := (Guess + Next) / 2
Until |Guess - Next| < ε
```

The equivalent assembly language, using the absolute value opcode, is now:

```
          MUFS      F1,F0,#0.5     Guess := N / 2
Loop      DVFS      F2,F0,F1       Next  := N / Guess
          ADFS      F1,F1,F2
          MUFS      F1,F1,#0.5     Guess := (Guess+Next)/2
          SUFS      F4,F1,F2
          ABSS      F4,F4          F4 := Abs(Guess - Next)
          CMF       F3,F4
          BLE       Loop           Repeat If F4 > ε
```

The Initial Guess

The final topic concerns the proper selection of the initial guess. We've been using *half* of the original number N as the first guess, but for larger and larger numbers this becomes a weaker and weaker choice. Ideally, we would like the initial guess to be the exact value of the square root, so the iterative algorithm makes only a single pass through its loop before exiting. The closer we get to the correct value at the beginning, the fewer passes through the loop will be required.

If you know about how single-precision floating-point numbers are constructed, we can do some "bit bashing" to generate a pretty good first guess. For any binary fraction of the form $1.xxxx \times 2^Y$, the exponent of its square root will be $\lfloor Y \div 2 \rfloor$ (that is, the largest integer less than or equal to $Y \div 2$, discarding any fractions when the exponent is odd). For example, the square root of 2^{16} is 2^8, and the square root of 2^{-8} is 2^{-4}. In cases where the exponent is odd, the mantissa of the square root should be scaled up or down by an extra square root of 2, or about 1.414…, but we can ignore that for now. Thus, an appropriate first guess is $1.0 \times 2^{\lfloor Y \div 2 \rfloor}$, and this number can be constructed as follows:

(1) Extract the biased exponent portion of N.
(2) Remove the bias.
(3) Divide the exponent by 2.
(4) Re-add the bias.
(5) Put the new biased exponent back where it belongs.
(6) Set the fraction to 1.0 by clearing all bits in the mantissa (since the rule for storing floating point numbers says to omit the leading 1 bit to the left of the binary point, the rest of the mantissa will be zero).

All of these steps can be done on the *integer* side of the processor, so we can take advantage of its bit-shifting capabilities. Somewhere in our program we have the value of N stored in memory, and that memory is declared to be a single-precision floating-point number with the directive:

```
N     DCFS  value
```

To generate the initial guess, we first load N into *integer* register `R0`, and take things from there. N is assumed to be positive, since negative numbers are illegal arguments to square root; by the time we get this far it is fair to assume that negative numbers have been filtered out. Thus, we can assume that the sign bit is 0.

By shifting the number in `R0` to the right by 23 bits (the size of the single-precision mantissa) we kill two birds with one stone: we discard the existing mantissa, and we align the biased exponent with the right-hand bits of the register. At this point the only thing remaining in `R0` is the biased exponent as a simple 8-bit value between 0 and 255.

We can't divide the biased exponent by two at this point. We must subtract off the bias, divide the now *signed* integer (between -128 and +127) by two, and then add back the bias. Dividing a signed integer by two is a little more complicated than simply shifting it to the right by one bit. This process will be discussed in detail later.

Finally, shifting the result back to the left by 23 bits restores the biased exponent to its proper position in the number and keeps the mantissa bits equal to zero. Here is the final assembly language code to build the initial guess into integer register R0.

```
LDR   R0,N           Load 32 bit floating pt into integer R0
MOV   R0,R0,LSR #23  Discard mantissa, shift exponent down
SUB   R0,R0,#127     Remove the bias, R0 now in [-128..+127]
MOVS  R0,R0,ASR #1   Divide R0 by 2 sign extend, C=remainder
ADCMI R0,R0,#0       Correct for division of odd negatives
ADD   R0,R0,#127     Re-add the bias
MOV   R0,R0,LSL #23  Shift exponent back where it belongs
```

Register R0 now contains the initial guess, which needs to be transferred via memory or the stack to a floating point register such as F1. Since we have the case where the bit pattern in R0 already *looks like* a floating point number, we cannot use the FLT (float) instruction to move the number into a floating point register. If we did, the FLT instruction it will treat the pattern in R0 as a large integer and will attempt to convert it to floating point, not knowing that it is already in the correct format.

The MOVS and the ADCMI instructions in the code above must look very strange. Together they properly divide the (no longer biased) signed exponent by two. The instruction MOVS R0,R0,ASR #1 performs a right-shift on R0, dividing it by 2, but it preserves the sign bit through ***sign-extension***. Normally, LSR #1 (***logical shift right***) pumps in a 0-bit at the left as it shifts, but ASR #1 (***arithmetic shift right***) pumps in 0 if the sign bit is 0 (positive) and pumps in 1 if the sign bit is 1 (negative). This keeps positive numbers positive and negative numbers negative. By using MOVS instead of MOV, the N status bit will reflect the sign of that number and the C (carry) bit will hold the bit shifted out of the right end of the number.

The process works fine for positive numbers, but *odd negative integers* are too small by 1 after a simple right-shift. For example, the even number -4 has (in eight bits) binary representation 11111100, which when shifted right with sign extension becomes 11111110, or -2. This works OK. However, the odd number -3 has the binary representation 11111101, which when shifted right generates 11111110, also equal to -2. The result *should be* 11111111 in binary, or -1 in decimal. The detectable difference between the two shift examples is in the carry bit, which will become 0 after the shift for even numbers (no correction needed) and will become 1 for odd numbers (which are too low by 1). The shift result can be corrected by simply adding to it the value of the carry bit. The add-with-carry instruction ADCMI R0,R0,#0 adds back just the carry bit, but the condition allows it to do so only if the original number was negative. The addition doesn't happen for positive numbers, and negative even numbers aren't affected.