Computing Integer Square Roots

James Ulery
phasorheal@yahoo.com

Assume that one has an arbitrary unsigned 32-bit value \( v \) for which one wishes to find the largest unsigned 16-bit number \( n \) such that \( n^2 \leq v \). One can write:

\[ n^2 = (a_{15}2^{15} + a_{14}2^{14} + \ldots + a_0)^2 \leq v \]

where the \( a_i \) are either 0 or 1. How does one find the \( a_i \)? Following is the derivation of an iterative algorithm, requiring no multiplications, for doing so. It is a variation of the traditional “subtract and shift” division algorithm. First, for convenience, define

\[ r_i = a_i2^i + a_{i-1}2^{i-1} + \ldots + a_0. \]

Equation 1 can then be written as

\[ n^2 = r_{15}^2 \leq v. \]

Say that the current best guess for \( n \), \( \hat{n}_0 \), is 0. Since \( \hat{n}_0 \) is 0, one can add it to anything without changing things, so

\[ (\hat{n}_0 + r_{15})^2 = \hat{n}_0^2 + 2\hat{n}_0r_{15} + r_{15}^2 \leq v \]

which can be rewritten as

\[ 2\hat{n}_0r_{15} + r_{15}^2 = 2\hat{n}_0(a_{15}2^{15} + r_{14}) + (a_{15}2^{15} + r_{14})^2 \leq v - \hat{n}_0^2. \]

This can be partitioned as

\[ a_{15}[2\hat{n}_02^{15} + (2^{15})^2] + [2\hat{n}_0r_{14} + 2a_{15}2^{15}r_{14} + r_{14}^2] \leq v - \hat{n}_0^2. \]

This expression represents the key to the bit iteration test that forms the core of this algorithm, as follows. First, note that both bracketed terms are positive. Now, say that one wishes to know whether \( a_{15} \) is 0 or 1. Compute the left bracketed term (LBT). If the LBT exceeds \( v - \hat{n}_0^2 \), Equation 6 would be violated if \( a_{15} \) were 1, so \( a_{15} \) must therefore be 0. However, does this mean that if the LBT does not exceed \( v - \hat{n}_0^2 \) \( a_{15} = 1 \)? Note that the LBT is a “step” that “gets you closer to” \( v - \hat{n}_0^2 \), and that if you choose not to use it, the value of the right bracketed term (RBT) must at least equal the LBT to “keep up with” \( v - \hat{n}_0^2 \). However, with \( a_{15} = 0 \), the RBT becomes \( 2\hat{n}_0r_{14} + r_{14}^2 \), which is always less than the LBT, so one would never choose \( a_{15} = 0 \) if the LBT did not exceed \( v - \hat{n}_0^2 \). The test for \( a_{15} = 1 \) then becomes simply

\[ 1 \text{ Though the derivation here uses specific input and output word sizes to make it concrete, it can be generalized to any required input or output word length.} \]
Equation 7
\[ 2\hat{n}_0(2^{15}) + (2^{15})^2 \leq v - \hat{n}_0^2. \]

If this test is not true, \( a_{15} \) must be 0. Having determined \( a_{15} \), one then updates the best guess:

Equation 8
\[ \hat{n}_1 = \hat{n}_0 + a_{15}2^{15} \]

Squaring,

Equation 9
\[ \hat{n}_1^2 = \hat{n}_0^2 + 2\hat{n}_0 a_{15} 2^{15} + a_{15} (2^{15})^2. \]

Returning to Equation 5,

Equation 10
\[
2\hat{n}_0(a_{15}2^{15} + r_{14}) + (a_{15}2^{15} + r_{14})^2 \leq v - \hat{n}_0^2 \\
2\hat{n}_0 a_{15} 2^{15} + 2\hat{n}_0 r_{14} + (a_{15} 2^{15})^2 + 2a_{15} 2^{15} r_{14} + r_{14}^2 \leq v - \hat{n}_0^2 \\
2(\hat{n}_0 + a_{15} 2^{15}) r_{14} + r_{14}^2 \leq v - \left( \hat{n}_0^2 + 2\hat{n}_0 a_{15} 2^{15} + a_{15} (2^{15})^2 \right)
\]

Substituting Equation 8 and Equation 9 into the last line of Equation 10 one gets

Equation 11
\[ 2\hat{n}_1 r_{14} + r_{14}^2 \leq v - \hat{n}_1^2. \]

But this is in exactly the same form as Equation 5, from which \( a_{15} \) was determined. The test for \( a_{14} \) would therefore be

Equation 12
\[ 2\hat{n}_1(2^{14}) + (2^{14})^2 \leq v - \hat{n}_1^2. \]

Algorithm Specification

The previous discussion suggests an iterative procedure. For \( i=0,1,2,...,15 \):

Equation 13
\[
a_{15-i} = \begin{cases} 
1 \text{ iff } 2\hat{n}_i(2^{15-i}) + (2^{15-i})^2 \leq v_i \\
0, \text{ else} 
\end{cases} \\
\nu_{i+1} = v_i - [2\hat{n}_i a_{15-i} 2^{15-i} + a_{15-i} (2^{15-i})^2] \\
\hat{n}_{i+1} = \hat{n}_i + a_{15-i} 2^{15-i}
\]

where
\[ \hat{n}_0 = 0 \]
\[ \nu_0 = v \]

These computations can be carried out by shifts and adds only.
C Implementation

The previous algorithm results in a very simple C implementation:

```c
unsigned isqrt(unsigned long v) {
    unsigned long temp, nHat=0, b = 0x8000, bshft = 15;
    do {
        if (v >= (temp = (((nHat<<1)+b)<<bshft--))) {
            nHat += b;
            v -= temp;
        }
    } while (b >>= 1);
    return nHat;
}
```
Here, I use the form:

\[ 2 \hat{n}_i \left( 2^{15-i} \right) + \left( 2^{15-i} \right)^2 = \left( 2 \hat{n}_i + \left( 2^{15-i} \right) \right) 2^{15-i} \rightarrow ((nHat << 1) + b) << bshft -- \]

when computing the \( a_i \).

ARM Implementation

Because of the ARM’s barrel shifter, this algorithm maps very efficiently to ARM assembly code:

```assembly
; Define register aliases to make the code easier to read.
v    RN    r0
nHat RN    r1
b    RN    r2
bShft RN    r3
temp RN    r12

isqrt mov    b, #0x8000
           mov    bShft, #15
           mov    nHat, #0

loop add    temp, b, nHat, lsl #1
           subs    temp, v, temp, lsl bShft
           addge   nHat, nHat, b
           movge   v, temp
           sub    bShft, bShft, #1
movs    b, b, lsr #1
bne    loop
mov    r0, nHat
mov    pc, lr
```