0 to 9	0.80	0
0 to 99	1.46	0.83
0 to 999	1.58	1.44
0 to 9999	2.13	2.06
0 to $2^{32} - 1$	2.97	2.97

If we assume a divide time of 20 cycles and *x* ranging uniformly from 0 to 9999, then both algorithms execute in about 81 cycles.

Binary Search

Because the algorithms based on Newton's method start out with a sort of binary search to obtain the first guess, why not do the whole computation with a binary search? This method would start out with two bounds, perhaps initialized to 0 and 2^{16}. It would make a guess at the midpoint of the bounds. If the square of the midpoint is greater than the argument *x*, then the upper bound is changed to be equal to the midpoint. If the square of the midpoint is less than the argument *x*, then the lower bound is changed to be equal to the midpoint. The process ends when the upper and lower bounds differ by 1, and the result is the lower bound.

This avoids division, but requires quite a few multiplications—16 if 0 and 2^{16} are used as the initial bounds. (The method gets one more bit of precision with each iteration.) Figure 11-3 illustrates a variation of this procedure, which uses initial values for the bounds that are slight improvements over 0 and 2^{16}. The procedure shown in Figure 11-3 also saves a cycle in the loop, for most RISC machines, by altering *a* and *b* in such a way that the comparison is $b \geq a$ rather than $b - a \geq 1$.

Figure 11-3 Integer square root, simple binary search.

```
int isqrt(unsigned x) {
   unsigned a, b, m;            // Limits and midpoint.

   a = 1;
   b = (x >> 5) + 8;            // See text.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
```

```
    } while (b >= a);
    return a - 1;
}
```

The predicates that must be maintained at the beginning of each iteration are $a \le \lfloor \sqrt{x} \rfloor + 1$ and $b \ge \lfloor \sqrt{x} \rfloor$. The initial value of *b* should be something that's easy to compute and close to $\lfloor \sqrt{x} \rfloor$. Reasonable initial values are *x*, *x* ÷ 4 + 1, *x* ÷ 8 + 2, *x* ÷ 16 + 4, *x* ÷ 32 + 8, *x* ÷ 64 + 16, and so on. Expressions near the beginning of this list are better initial bounds for small *x*, and those near the end are better for larger *x*. (The value *x* ÷ 2 + 1 is acceptable, but probably not useful, because *x* ÷ 4 + 1 is everywhere a better or equal bound.)

Seven variations on the procedure shown in Figure 11-3 can be more or less mechanically generated by substituting *a* + 1 for *a*, or *b* - 1 for *b*, or by changing *m* = (*a* + *b*) ÷ 2 to *m* = (*a* + *b* + 1) ÷ 2, or some combination of these substitutions.

The execution time of the procedure shown in Figure 11-3 is about 6 + (*M* + 7.5)*n*, where *M* is the multiplication time in cycles and *n* is the number of times the loop is executed. The table below gives the average number of times the loop is executed, for *x* uniformly distributed in the indicated range.

x	**Average Number of Loop Iterations**
0 to 9	3.00
0 to 99	3.15
0 to 999	4.68
0 to 9999	7.04
0 to 2^{32} - 1	16.00

If we assume a multiplication time of 5 cycles and *x* ranging uniformly from 0 to 9999, the algorithm runs in about 94 cycles. The maximum execution time (*n* = 16) is about 206 cycles.

If *number of leading zeros* is available, the initial bounds can be set from

```
b = (1 << (33 - nlz(x))/2) - 1;
a = (b + 3)/2;
```

That is, $b = 2^{(33 - \mathrm{nlz}(x)) \div 2} - 1$. These are very good bounds for small values of x (one loop iteration for $0 \le x \le 15$), but only a moderate improvement, for large x, over the bounds calculated in Figure 11-3. For x in the range 0 to 9999, the average number of iterations is about 5.45, which gives an execution time of about 74 cycles, using the same assumptions as above.

A Hardware Algorithm

There is a shift-and-subtract algorithm for computing the square root that is quite similar to the hardware division algorithm described in Figure 9-2 on page 149. Embodied in hardware on a 32-bit machine, this algorithm employs a 64-bit register that is initialized to 32 0-bits followed by the argument x. On each iteration, the 64-bit register is shifted left two positions, and the current result y (initially 0) is shifted left one position. Then $2y + 1$ is subtracted from the left half of the 64-bit register. If the result of the subtraction is nonnegative, it replaces the left half of the 64-bit register, and 1 is added to y (this does not require an adder, because y ends in 0 at this point). If the result of the subtraction is negative, then the 64-bit register and y are left unaltered. The iteration is done 16 times.

This algorithm was described in 1945 [JVN].

Perhaps surprisingly, this process runs in about half the time of that of the 64 ÷ 32 ⟹ 32 hardware division algorithm cited, because it does half as many iterations and each iteration is about equally complex in the two algorithms.

To code this algorithm in software, it is probably best to avoid the use of a doubleword shift register, which requires about four instructions to shift. The algorithm in Figure 11-4 [GLS1] accomplishes this by shifting y and a mask bit m to the right. It executes in about 149 basic RISC instructions (average). The two expressions `y | m` could also be `y + m`.

Figure 11-4 Integer square root, hardware algorithm.

```
int isqrt(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while(m != 0) {              // Do 16 times.
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
```