Computer Science 237
Assignment 6
Due before lab next Tuesday (no extensions)

This week we will investigate the efficient computation of square roots. Here, we develop an algorithm which we can experiment with, in C. Your task is to convert the C code into a assembly and to experiment with a hardware implementation.

The Theory. Let's begin with some basic observations. Suppose n is an integer, then $\log_2 n$ is roughly the number of bits it takes to represent n in binary. Since multiplying numbers effectively sums their logarithms, then $\log_2(\sqrt{n})$ takes half the number of bits. Computing each bit of the root will consume *two* bits of the original number.

With this in mind, let's suppose we have a number, n, with k bits: $n = n[1..k]$ (be prepared: bits are numbered from the left starting at 1 for ease of description). We may safely assume that k is even (if it isn't, then simply write n with a leading zero). The square root of n, $r = r[1..\frac{k}{2}]$, is represented by $\frac{k}{2}$ bits, or roughly one bit for every two in n. The root-finding process extracts the exact root of n one digit $r[i]$ at a time using a *greedy algorithm*. The "aggressiveness" of this algorithm—its need to consume two bits at each iteration—allows us to prove theorems about its maximum running time.

Consider a very simple related problem: calculating the square root of just the two most significant bits of n, $n_1 = n[1..2]$. Represented as an integer, this root consists of a (single bit) value, $r[1]$. Since there are only two choices for $r[1]$ (0 or 1), this computation is simply a matter of a single test: is $n_1 \geq 1$. There may, of course, be a remainder, m_1, which relates these values:

$$n_1 = r[1]^2 + m_1$$

If, for example, $n_1 = n[1..2] = 10$, then $2 = 1^2 + 1$.

Now, let's consider a slightly more complex problem: What is the root of $n_2 = n[1..4]$? Our first observation relates n_2 to the answer of the simpler problem, above. Because n_2 contains two more bits, $d = n[3..4]$, to the right:

$$\begin{aligned} n_2 &= 4n_1 + d \\ &= 4(r[1]^2 + m_1) + d \\ &= 4r[1]^2 + (4m_1 + d) \end{aligned}$$

Second, we relate n_2 to its root $r[1..2]$ and remainder of m_2:

$$\begin{aligned} n_2 &= r[1..2]^2 + m_2 \\ &= (2r[1] + r[2])^2 + m_2 \\ &= 4r[1]^2 + 4r[1]r[2] + r[2]^2 + m_2 \end{aligned}$$

These two observations can be used to discover the next bit of the root, $r[2]$:

$$\begin{aligned} 4r[1]^2 + (4m_1 + d) &= 4r[1]^2 + 4r[1]r[2] + r[2]^2 + m_2 \\ 4m_1 + n[3..4] &\geq r[2](4r[1] + r[2]) \end{aligned}$$

To minimize the remainder we've dropped from this last equation, m_2, we select bit $r[2]$ so that the inequality is as tight as possible without violating it. Since $r[2]$ is a single bit, the possible values of the right side are:

$$\begin{aligned} 1(4r[1] + 1) &= 4r[1] + 1 \quad \text{or} \\ 0(4r[1] + 0) &= 0 \end{aligned}$$

The Algorithm. We now have the basis for an iterative algorithm that computes bit $r[i]$. At each iteration, we seek the greatest integer representation of the square root of $n_i = n[1..2i]$. We write this root, $r_i = r[1..i]$, and the resulting computation leaves a remainder m_i. It is useful to assume that $r_0 = 0$ and $m_0 = 0$.

Computing step $i+1$ involves taking the previous remainder, m_i, shifting it two bits to the left (multiplying by 4), and bringing down and adding in two more bits from n, $d = n[2i+1..2i+2]$. Call this value $(4m_i + d)$, v_i. We perform one check: see if the v_i is at least $4r_i + 1$. If it is, then $r[i+1] = 1$ and $m_{i+1} = v_i - (4r_i + 1)$. Otherwise $r[i+1] = 0$ and $m_{i+1} = v_i$. If n takes k bits (k even) to represent, we repeat this process $\frac{k}{2}$ times.

An Example, worked. Here is how we would compute the square root of $n = 81 = 01010001_2$ (which was padded to an even number of bits). Since this n takes 8 bits, we compute the root in 4 steps, remembering $r_0 = 0$ and $m_0 = 0$:

Step 1. The most significant 2 bits of 81 are $n[1..2] = v_1 = 01_2$. Since $4r_0 + 1 = 1$ does not exceed 01_2, we have that $r[1] = 1$ (thus the best approximation to the root is $r_1 = 1_2$). We compute $m_1 = v_1 - (4r_0 + 1) = 0$.

Step 2. We shift m_1 to the left two bits, and bring in the next two bits of n, $n[3..4] = 01_2$, giving an intermediate value $v_2 = 1$. Since $4r_1 + 1 = 5$ is greater than v_2, we must not be so aggressive and we set $r[2] = 0$ ($r_2 = 10_2 = 2$). The remainder is $m_2 = v_2 - 0 = 1$.

Step 3. We shift m_2 to the left two bits, and bring in the next two bits of n, $n[5..6] = 00_2$, giving an intermediate value $v_3 = 100_2 = 4$. Since $4r_2 + 1 = 4r[1..2] + 1 = 1001_2 = 9$ is greater than v_3, we have the next bit, $r[3] = 0$, ($r_3 = 100_2 = 4$). We get $m_3 = v_3 = 4$.

Step 4. We shift m_3 to the left two bits, and bring in the next two bits of n, $n[7..8] = 01_2$, giving an intermediate value $v_4 = 10001_2 = 17$. Since $4r_3 + 1 = 4r[1..3] + 1 = 10001_2 = 17$ is not greater than v_4, we have the next bit, $r[4] = 1$ (so $r_4 = 1001_2$). We get $m_4 = 0$. The square root of $n = 81$ is $r_4 = 9$ with a remainder of $m_4 = 0$. Notice that a square root of 83 would yield a similar calculation, but we would be left with remainder $m_4 = 2$.

Note that if we stored 81 using 16 bits, the first 4 stages of the 8-step calculation would have values $v_i = 0$, and the corresponding bits $r[i] = 0$.

In general, this process can be used to calculate roots of fractional values. The decimal point is inserted in r_i when the last integral bits are pulled down from n.

What to do. On the next page is the C version of the `root` routine. This method computes the individual bits of the square root and reports the result in the character buffer, `bits`. I would like you to translate this code into assembly in the file, `root.s`. When you are finished turn in your code using `turnin`.

```
// Compute the square root of n as a string of digits.
void root(int n, char bits[])
{
  int m = 0;                  // the remainder,
  int r = 0;                  // the root,
  int v;                      // the intermediate value
  int i;                      // loop counter
  for (i=0; i<4; i++) {       // compute a 4 bit answer
    v = (m<<2) + (n>>6);      // drop down 2 highest bits
    n = (n<<2) & 0xff;        // shift n left, remove high bits
    m = v-((r<<2)+1);         // compute temporary remainder
    if (m < 0) {
      bits[i] = '0';
      m = v;                  // remainder was negative, r[i] = 0
      r <<= 1;
    } else {
      bits[i] = '1';
      r = (r<<1)+1;           // remainder not negative, r[i] = 1
    }
  }
  bits[i] = 0;                // terminate string
}
```