Introduction to Computer Arithmetic
Ivor Page[1]

Arithmetic operations are amongst the most fundamental of instructions in digital computers. Arithmetic units, such as adders and multipliers, are also fundamental to signal processing, image processing, and many other special purpose information processing systems. It is therefore important that these systems are fast and have high precision. From an engineering perspective, we also want them to be power-efficient, small, cheap to manufacture, easy to test, and easy to understand. Some requirements tend to be prerequisites for others. Small size and speed both suggest efficient layouts with minimized critical path lengths.

At first sight, testability appears to be a very significant problem. Consider a 64 bit multiplier. Each input has 2^{64} possible settings. If we tested a multiplier by applying every possible input vector, one each nanosecond, the time to test the multiplier would exceed the age of the universe (10 to 17 billion years). Production line testers take only a second or so to test such a component. The testing strategy must follow the structure of the component, testing sub-units independently, ensuring that every connection is present, and every gate output switches properly. In many modern designs, extra data paths and arithmetic units are added to arithmetic units in order that the components test themselves.

Speed implies an efficient layout, void of long poly-runs, and a minimum of unnecessary logic transitions at gate outputs. Minimizing logic transitions also helps lower power consumption, but requires equalization of signal delays through the sub-components. Low power designs often require extra gates to equalize delay paths and to mask "glitches" that can occur in minimal gate designs. In general, the component with the fewest gates has the lowest cost, but testability and low power requirements may cause designers to add gates to the design.

Speed has traditionally been the main goal of designers. It has only been tempered by practicalities. Speed comes from advances in three areas: Algorithms, Architecture, and Technology. New algorithms have provided huge advances since the early days of calculating machines.

Consider a ripple-carry adder for 64 bit binary values. If gate delay is normalized to D, (ignoring fan-in and fan-out effects on gate delay), we know that the worst case time to add two values will be approximately $2 \times 64 \times D$. Multiplying two values using one of these adders will take 63 times this value. A carry look-ahead adder with a block size of 4 reduces the add time to $D \times 2 \times log_4\ 64 = 12D$ and the multiply time is reduced to 63 times this quantity.

A redundant representation adder has constant add time, independent of the number of bits being added. It's add time would be less than $8D$. If we have 32 of these adders, we can multiply two 64 bit values in time less than $36D$. If we have 32+16+8+4+2+1=63 of these adders, we can pipeline them to provide a new multiply result every $6D$.

[1]University of Texas at Dallas

In this example we have gone from a multiply time of $8064D$ to $6D$. The number of gates has risen substantially (we will calculate this later), but the multiplier has an efficient layout and is easy to test. Multipliers based on redundant binary representation are widely used in signal processing applications.

The example also shows progress over time. Carry-look-ahead and the Manchester Carry Chain were used in the most powerful computers of the 1960's (the Atlas at Manchester University in England and the CDC 6600 in the USA). Although much of the math behind more modern systems was in place by the 1960's, the designers of these systems were already pushing the upper limit on the number of gates for such systems to be reliable. Some engineers had calculated that the CDC 6600 would have mean time between failures of only a couple of hours. It turned out to be much more reliable than predicted, but it generated a great deal of heat. The 7600 had a special Freon cooling system and the Cray 1 (of the late 70's) consumed over 120KW.

Modern VLSI technology has made it practical to build systems with large numbers of gates. TI had 64 bit IEEE compatible floating-point redundant binary multipliers with operating times of 35 nanoseconds by 1990.

Technology has made practical the more complex systems that use larger numbers of gates, while architecture has provided layouts and arrangements of adders, busses and pipeline registers that enabled a great deal of parallelism. Algorithms (or fundamental mathematics) provided a variety of ways forward that gave designers choices in balancing speed, size, ease of testing, power consumption (and therefore heat generated), and reliability. Advances are still being made in all three domains.

1.1 Limits of Arithmetic Speed

1.1.1 "Big O Notation"

Recall that, when we say that the computation time of a system is $O(f(k))$, we mean that, for values of $k > k_0$, the actual time, $T(k)$, is bounded above by $c \times f(k)$, where c and k_0 are constants. This means that we can choose constants c and k_0 such that, $T(k) \leq cf(k)$ for $k > k_0$. For our purposes, k is the number of digits in the numbers being added.

For example, if $T(k) = 200000 + 30 \times k + 1000000 \times k^2 + 5 \times k^5$, we would write $T(k) = O(k^5)$ since, for very large values of k, the term with the largest exponent dominates.

For positional number systems, integer addition, multiplication, and division have theoretical time limits of $O(1)$, $O(log\ k)$, and $O(log\ k)$, respectively.

The result for division is impractical, although we shall see later that floating point division has log time. Pipelining of a multiply unit enables us to begin new multiply operations (and obtain new results) P times faster than the $O(log\ k)$ time of a non-pipelined unit. Each multiply operation still takes $O(log\ k)$ time. Here P is the pipeline factor. It represents the

amount of parallelism or overlap obtained by pipelining. These results imply that addition times of a few gate delays are feasible, while multiply times of $O(log\ k)$ are also practical.

A Note of Caution

Big O notation is only of value in comparing the asymptotic behavior of algorithms or systems when the parameter (k in the above) becomes very large. In computer arithmetic we are almost always concerned with numbers that have no more than 256 bits. It is therefore important that we take into account the constants when comparing arithmetic systems. There are some practical examples of systems in which the Big O approach hides significant differences in the constants involved which lead to very significant differences in actual speed. For example, multiplier X and multiplier Y, may both have asymptotic run times of $O(log\ k)$, but if X is twice the speed of Y for the values of k for which we are designing our system, then X is easily the winner. Big O notation is mainly of value in Computer Science theory where the run times of algorithm are compared for very large values of the parameter.

Can we go even faster?

We shall see that residue number systems provide a way to reduce the time for multiply so that it is somewhat independent of the number of digits. There are some technical caveats that we need to discuss, including the time to convert to and from residue representation. Division is very difficult in the residue system.

1.2 Other Arithmetic Functions

So much for the four arithmetic operations on integers. What about floating point and other functions, such as square-root, log, sin, etc? We will study floating point systems, sticky bits, guard bits, and the IEEE standard in detail and we will also consider the effects of round-off errors and use some elementary numerical analysis to study accuracy in floating point calculations. We will also study methods for computing the transendental functions, both the digit-at-a-time system (Cordic) widely used in hand-held calculators, and the systems used in digital computers.

There is a lot to study. I will try to make it interesting and provide you will challenging projects and home works that help you understand and demonstrate your knowledge of the material.

Let's get started.