7 Computer Arithmetic

In Chapter 6, we described the basic circuits for logical operations and storage elements. In this chapter, we will use this knowledge to design hardware algorithms for arithmetic operations. This chapter also requires the knowledge of binary 2's complement numbers and floating point numbers that we gained in Chapter 2. The plan for this chapter is as follows.

In the first part, we describe algorithms for integer arithmetic. Initially, we describe the basic algorithms for adding two binary numbers. It turns out that there are many ways of doing these basic operations, and each method has its own set of pros and cons. Note that the problem of binary subtraction is conceptually the same as binary addition in the 2's complement system. Consequently, we do not need to treat it separately. Subsequently, we shall see that the problem of adding n numbers is intimately related to the problem of multiplication, and it is a fast operation in hardware. Sadly, very efficient methods do not exist for integer division. Nevertheless, we shall consider two popular algorithms for dividing positive binary numbers.

After integer arithmetic, we shall look at methods for floating point (numbers with a decimal point) arithmetic. Most of the algorithms for integer arithmetic can be ported to the realm of floating point numbers with minor modifications. As compared to integer division, floating point division can be done very efficiently.

7.1 Addition

7.1.1 Addition of Two 1-bit Numbers

Let us look at the problem of adding two 1-bit numbers, a and b. Both a and b can take two values – 0 or 1. Hence, there are four possible combinations of a and b. Their sum in binary can be either 00, 01, or 10. Their sum will be 10, when both a and b are 1. We should make an important observation here. The sum of two 1 bit numbers might potentially be two bits long. Let us call the LSB of the result as the *sum*, and the MSB as the *carry*. We can relate this concept to standard primary school addition of two 1 digit decimal numbers. If we are adding 8 and 9, then the result is 17. We say that the sum is 7, and the carry is 1. Similarly, if we add 3 and 4, then the result is 7. We say that the sum is 7, and the carry is 0.

We can extend the concept of sum and carry to adding three 1 bit numbers also. If we are adding three 1 bit numbers then the range of the result is between 00 and 11 in binary. In this case also, we call the LSB as the *sum*, and the MSB as the *carry*.

Definition 52

sum *The sum is the LSB of the result of adding two or three 1 bit numbers.*

carry *The carry is the MSB of the result of adding two or three 1 bit numbers.*

For an adder that can add two 1 bit numbers, there will be two output bits – a sum s and a carry c. An adder that adds two bits is known as a *half adder*. The truth table of a half adder is shown in Table 7.1.

Definition 53
A half adder *adds two bits to produce a sum and a carry.*

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 7.1: Truth table of a half adder

From the truth table, we can conclude that $s = a \oplus b = \overline{a}.b + a.\overline{b}$, where $\oplus$ stands for exclusive or, '.' stands for boolean AND, and '+' stands for boolean OR. Secondly, $c = a.b$. The circuit diagram of a half adder is shown in Figure 7.1. As we can see, a half adder is a very simple structure and we have constructed it using just six gates in Figure 7.1.

7.1.2 Addition of Three 1-bit Numbers

The aim is to be ultimately able to add 32-bit numbers. To add the two least significant bits, we can use a half adder. However, for adding the second bit pair, we cannot use a half adder because there might be an output carry from the first half adder. In this case, we need to add three 1-bit numbers. Hence, we need to implement a *full adder* that can add 3 bits. One of these bits is a carry out of another adder and we call it the *input carry*. We represent the input carry as c_{in}, and the two other input bits as a and b.

Definition 54 *An adder than can add 3 bits is known as a full adder.*

Table 7.2 shows the truth table for the full adder. We have three inputs – a, b, and c_{in}. There are two output bits – the sum (s), and the carry out (c_{out}).

From the truth table, we can deduce the following relationships:

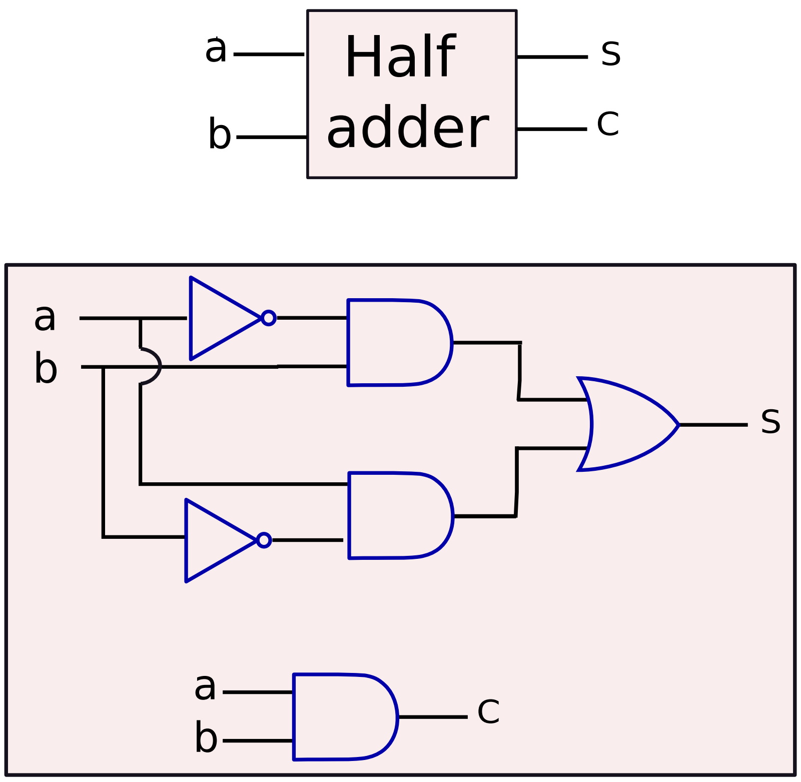


Figure 7.1: A half adder

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 7.2: Truth table of a full adder

$$\begin{aligned}
s &= a \oplus b \oplus c_{in} \\
&= (a.\overline{b} + \overline{a}.b) \oplus c_{in} \\
&= (a.\overline{b} + \overline{a}.b).\overline{c_{in}} + \overline{(a.\overline{b} + \overline{a}.b)}.c_{in} \\
&= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + \overline{(a.\overline{b})}.\overline{\overline{a}.b}.c_{in} \\
&= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + (\overline{a} + b).(a + \overline{b}).c_{in} \\
&= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + \overline{a}.\overline{b}.c_{in} + a.b.c_{in} \\
c_{out} &= a.b + a.c_{in} + b.c_{in}
\end{aligned}$$

The circuit diagram of a full adder is shown in Figure 7.2. This is far more complicated than the circuit of a half adder. We have used 12 logic gates to build this circuit. Furthermore, some of these logic gates use three inputs. However, this degree of complexity is required because all our practical adders will use full adders as their basic element. We face the need of adding 3 bits in all of our arithmetic algorithms.

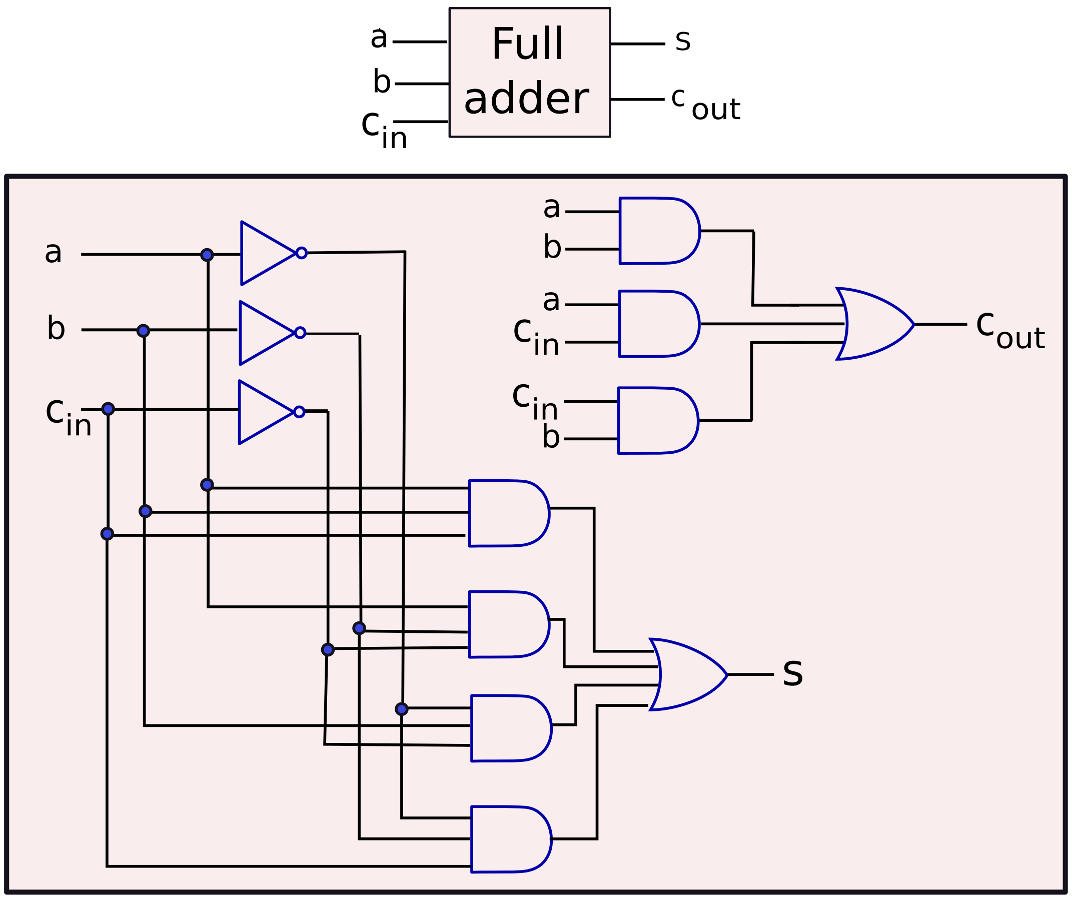


Figure 7.2: A full adder

7.1.3 Ripple Carry Adder

Let us now try to add two n bit numbers. Let us start with an example: $1011_2 + 0101_2$. The addition is shown in Figure 7.3. We have seen in Section 2.2.3 that binary numbers can be added the same way as decimal numbers. In the case of base 10 decimal numbers, we start at the unit's digit and proceed towards higher digits. In each step, a carry might be generated, which is then added to the immediately higher digits. In the case of binary numbers also we do the same. The only difference is that instead of base 10, we are using base 2.

For example, in Figure 7.3, we observe that when two binary bits are added a carry might be generated. The value of the carry is equal to 1. This carry needs to be added to the bits in the next position (more significant position). The computation is complete when we have finished the addition of the most significant bits. It is possible that a carry might propagate from one pair of bits to another pair of bits. This process of propagation of the carry from one bit pair to another is known as *rippling*.

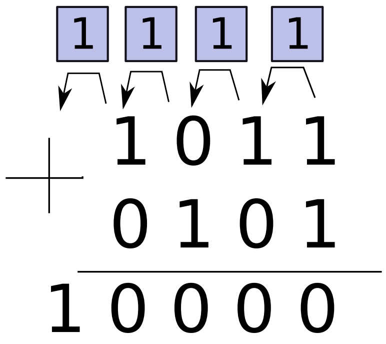


Figure 7.3: Addition of two binary numbers

Let us construct a simple adder to implement this procedure. Let us try to add two n bit binary numbers – A and B. We number the bits of A and B as $A_1 \ldots A_n$ and $B_1 \ldots B_n$ respectively. Let A_1 refer to A's LSB, and A_n refer to A's MSB. We can create an adder for adding A and B as follows. We use a half adder to add the LSBs. Then we use $n-1$ full adders to add the rest of the corresponding bits of A and B and their input carry values. This n bit adder is known as a *ripple carry adder*. Its design is shown in Figure 7.4. We observe that we add two n bit numbers to produce a $n+1$ bit result. The method of addition is exactly similar to the procedure we follow while adding two binary numbers manually. We start from the LSB and move towards the MSB. At every step we propagate the carry to the next pair of digits.

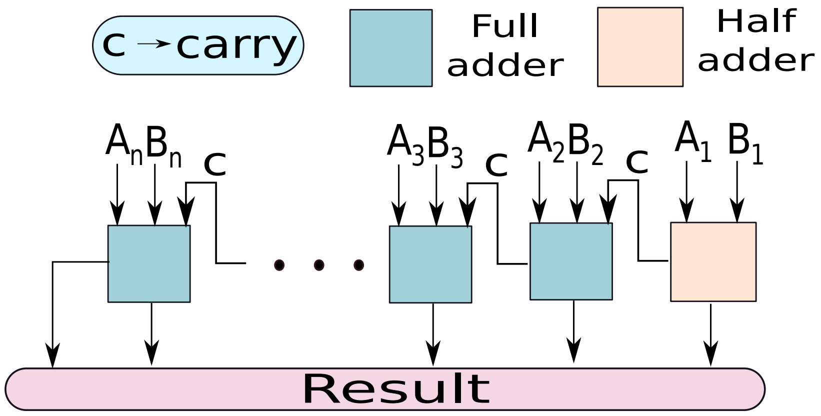


Figure 7.4: Addition of two binary numbers

Now, let us calculate the speed of this adder. Let us assume that it takes t_h units of time for a half adder to complete its operation, and t_f units of time for a full adder to complete its operation. If we assume that carries are propagated instantaneously across blocks, then the total time, $f(n)$, is equal to $t_h + (n-1)t_f$. Here, n is equal to the number of bits being added.

However, as we shall see this is a rather cryptic basis of comparison, especially for large values of n. We do not wish to have a lot of constants in our timing model. Secondly, the values of these constants are heavily dependent on the specific technology used. It is thus hard to derive algorithmic insights. Hence, we introduce the notion of *asymptotic time complexity* that can significantly simplify the timing models, yet retain their basic characteristics. For example, in the case of a ripple carry adder, we can say that the complexity is almost equal to n multiplied by some constant. We can further abstract away the constant, and say that the time complexity is the order of n. Let us now formally define this notion.

Asymptotic Time Complexity

Let us consider two functions $f(n) = 2n^2 + 3$, and $g(n) = 10n$. Here, n is the size of the input, and $f(n)$, and $g(n)$ represent the number of time units it takes for a certain circuit to complete its operation. We plot the time values for different values of n in Figure 7.5. As we can see, $g(n)$ is greater than $f(n)$ for small values of n. However, for larger values of n, $f(n)$ is larger, and it continues to be so. This is because it contains a square term, and $g(n)$ does not. We can extend this argument to observe that even if $g(n)$ would have been defined to be $100n$, $f(n)$ would have ultimately exceeded it. The gist of the argument lies in the fact that $f(n)$ contains a quadratic term (n^2) and $g(n)$ only contains linear terms. For large n, we can conclude that $f(n)$ is slower than $g(n)$. Consequently, we need to define a new notion of time that precisely captures this fact. We call this new notion of time as the **asymptotic time complexity**. The name comes from the fact that we are interested in finding an envelope or asymptote to the time function such that the function is contained within this envelope for practically large values of n.

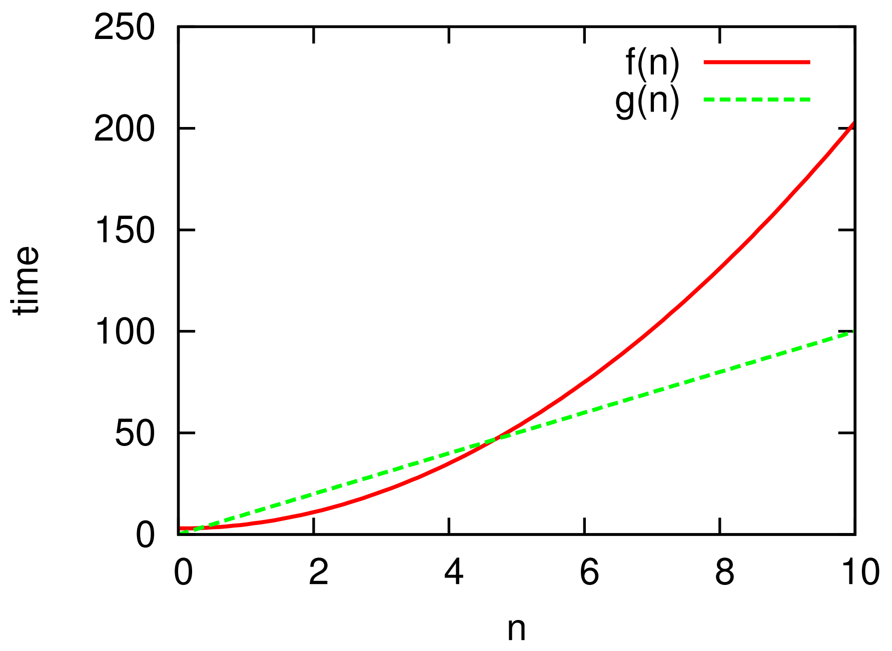


Figure 7.5: $f(n) = 2n^2 + 3$ and $g(n) = 10n$

For example, we can define the asymptotic time complexity of $f(n)$ to be n^2 and that of $g(n)$ to be n respectively. This notion of time is powerful enough to say that $f(n)$ is greater than $g(n)$ for values of n larger than some threshold. What if we consider: $f(n) = 2n^2 + 3$, and $f'(n) = 3n^2 + 10$. Needless to say, $f'(n) > f(n)$. However, we might not be interested in the difference. If we compare the asymptotic time complexity of $f(n)$ or $f'(n)$ with another function that has terms with different exponents (other than 2), then the results of the comparison will be the same. Consequently, for the sake of simplicity we can ignore the additive and multiplicative constants. We capture the definition of one form of asymptotic time in the big-O notation. It is precisely defined in Definition 55.

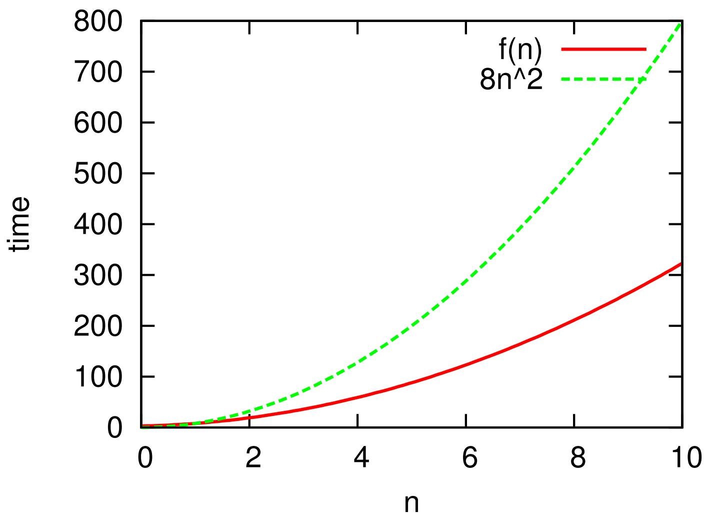
Definition 55
We say that: $f(n) = O(g(n))$
if $| f(n) | \leq c | g(n) |$ *for all* $n > n_0$. *Here,* c *is a positive constant.*

The big-O notation is actually a part of a set of asymptotic notations. For more details, the reader can refer to a standard text in computer algorithms [Cormen et al., 2009]. From our point of view, $g(n)$ gives a worst case time bound for $f(n)$ ignoring additive and multiplicative constants. We illustrate this fact with two examples: Examples 93 and 94. In this book, we will refer to asymptotic time complexity as *time complexity.*

Example 93
$f(n) = 3n^2 + 2n + 3$. *Find its asymptotic time complexity.*
Answer:

$$\begin{aligned} f(n) &= 3n^2 + 2n + 3 \\ &\leq 3n^2 + 2n^2 + 3n^2 \quad (n > 0) \\ &\leq 8(n^2) \end{aligned}$$

Hence, $f(n) = O(n^2)$.



$8n^2$ *is a strict upper bound on* $f(n)$ *as shown in the figure.*

Example 94
$f(n) = 0.00001n^{100} + 10000n^{99} + 234344$. *Find its asymptotic time complexity.*
Answer: $f(n) = O(n^{100})$

Time Complexity of a Ripple Carry Adder

The worst case delay happens when the carry propagates from the least significant bit to the most significant bit. In this case, each full adder waits for the input carry, performs the addition, and then propagates the carry out to the next full adder. Since, there are n 1 bit adders, the total time taken is $O(n)$.

7.1.4 Carry Select Adder

A ripple carry adder is extremely slow for large values of n such as 32 or 64. Consequently, we desire faster implementations. We observe that in hardware we can potentially do a lot of tasks in parallel. Unlike purely sequential C or Java programs where one statement executes after the next, hundreds or even thousands of actions can be performed in parallel in hardware. Let us use this insight to design a faster adder that runs in $O(\sqrt{n})$ time.

Let us consider the problem of adding two numbers A and B represented as: $A_{32} \ldots A_1$ and $B_{32} \ldots B_1$ respectively. Let us start out by dividing the set of bits into blocks of let us say 4 bits. The blocks are shown in Figure 7.6. Each block contains a fragment of A and a fragment of B. We need to add the two fragments by considering the input carry to the block, and generate a set of sum bits and a carry out. This carry out is an input carry for the subsequent block.

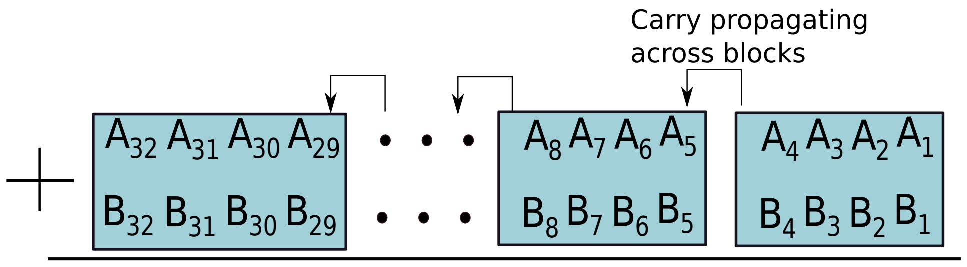


Figure 7.6: Dividing the numbers into blocks

In this case, a carry is propagated between blocks rather that between bit pairs. To add the pair of fragments within a block, we can use a simple ripple carry adder. For small values of n, ripple carry adders are not very inefficient. However, our basic problem of carry propagation has not been solved yet.

Let us now introduce the basic idea of the *carry select adder*. We divide the computation into two stages. In the first stage, we generate two results for each block. One result assumes that the input carry is 0, and the other result assumes that the input carry is 1. A result consists of 4 sum bits, and a carry out. We thus require two ripple carry adders per block. Note that each of these additions are independent of each other and thus can proceed in parallel.

Now, at the beginning of the second stage two sets of results for the n^{th} block are ready. If we know the value of the input carry, C_{in} produced by the $(n-1)^{th}$ block, then we can quickly calculate the value of the output carry, C_{out}, by using a simple multiplexer. We do not need to perform any extra additions. The inputs to the multiplexer are the values of C_{out} generated by the two ripple carry adders that assume C_{in} to be 0 and 1 respectively. When the correct value of C_{in} is available, it can be used to choose between the two values of C_{out}. This process is much faster than adding the two blocks. Simultaneously, we can also choose the right set of sum bits. Then we need to propagate the output carry, C_{out}, to the $(n+1)^{th}$ block.

Let us now evaluate the time complexity of the carry select adder. Let us generalise the problem and assume the block size to be k. The first stage takes $O(k)$ time because we add each pair of fragments within a block using a regular ripple carry adder, and all the pairs of fragments are added in parallel. The second phase takes time $O(n/k)$. This is because we have have $\lceil n/k \rceil$ blocks and we assume that it takes 1 time unit for the input carry of a block to choose the right output carry in the multiplexer. The total time is thus: $O(k + n/k)$. Note that we are making some simplistic assumptions regarding the constants. However, our final answer will not change if we make our model more complicated.

Let us now try to minimise the time taken. This can be done as follows:

$$\begin{aligned}&\frac{\partial(k + n/k)}{\partial k} = 0\\ \Rightarrow& 1 - \frac{n}{k^2} = 0\\ \Rightarrow& k = \sqrt{n}\end{aligned} \tag{7.1}$$

Thus, the optimal block size is equal to $\sqrt{n}$. The total time complexity is thus $O(\sqrt{n} + \sqrt{n})$, which is the same as $O(\sqrt{n})$.

7.1.5 Carry Lookahead Adder

We have improved the time complexity from $O(n)$ for a ripple carry adder to $O(\sqrt{n})$ for a carry select adder. The question is, "Can we do better?" In this section, we shall present the *carry lookahead adder* that can perform addition in $O(log(n))$ time. $O(log(n))$ has been proved as the theoretical lower bound for adding two n bit numbers. Note that the *log* operation in this book typically has a base equal to 2, unless explicitly mentioned otherwise. Secondly, since logarithms to different bases differ by constant multiplicative factors, the base is immaterial in the big-O notation.

Generate and Propagate Functions

Before introducing the adder, we need to introduce a little bit of theory and terminology. Let us again consider the addition of two numbers – A and B – represented as $A_{32} \ldots A_1$ and $B_{32} \ldots B_1$ respectively. Let us consider a bit pair – A_i and B_i. If it is equal to (0,0), then irrespective of the carry in, the carry out is 0. In this case, the carry is *absorbed.*

However, if the bit pair is equal to (0,1) or (1,0) then the value of the carry out is equal to the value of the carry in. If the carry in is 0, then the sum is 1, and the carry out is 0. If the carry in is 1, then the sum is 0, and the carry out is 1. In this case, the carry is *propagated.*

Lastly, if the bit pair is equal to (1,1), then the carry out is always equal to 1, irrespective of the carry in. In this case, a carry is *generated.*

We can thus define a *generate*(g_i) and *propagate*(p_i) function as follows:

$$g_i = A_i.B_i \tag{7.2}$$

$$p_i = A_i \oplus B_i \tag{7.3}$$

The generate function captures the fact that both the bits are 1. The propagate function captures the fact that only one of the bits is 1. We can now compute the carry out C_{out} in terms of the carry in C_{in}, g_i, and p_i. Note that by our case by case analysis, we can conclude that the carry out is equal to 1, only if a carry is either generated, or it is propagated. Thus, we have:

$$C_{out} = g_i + p_i.C_{in} \tag{7.4}$$

Example 95
$A_i = 0$, $B_i = 1$. Let the input carry be C_{in}. Compute g_i, p_i and C_{out}.

Answer:

$$\begin{aligned} g_i &= A_i.B_i = 0.1 = 0 \\ p_i &= A_i \oplus B_i = 0 \oplus 1 = 1 \\ C_{out} &= g_i + p_i.C_{in} = C_{in} \end{aligned} \tag{7.5}$$

Let us now try to generalise the notion of generate and propagate functions to multiple bits. Let us consider a two bit system that has an input carry, and an output carry. Let the bit pairs be numbered 1 and 2, where 2 represents the most significant bit. Let C_{out}^i represent the output carry obtained after adding the i^{th} bit pair. Likewise, C_{in}^i is the input carry for the i^{th} bit pair. The output carry of the two bit system is thus equal to C_{out}^2. We have:

$$\begin{aligned} C_{out}^2 &= g_2 + p_2.C_{out}^1 \\ &= g_2 + p_2.(g_1 + p_1.C_{in}^1) \\ &= (g_2 + p_2.g_1) + p_2.p_1.C_{in}^1 \end{aligned} \tag{7.6}$$

Similarly, for a 3 bit system, we have:

$$\begin{aligned} C_{out}^3 &= g_3 + p_3.C_{out}^2 \\ &= g_3 + p_3.((g_2 + p_2.g_1) + p_2.p_1.C_{in}^1) \\ &= (g_3 + p_3.g_2 + p_3.p_2.g_1) + p_3.p_2.p_1.C_{in}^1 \end{aligned} \tag{7.7}$$

For a 4-bit system, we have:

$$\begin{aligned} C_{out}^4 &= g_4 + p_4.C_{out}^3 \\ &= g_4 + p_4.((g_3 + p_3.g_2 + p_3.p_2.g_1) + p_3.p_2.p_1.C_{in}^1) \\ &= (g_4 + p_4.g_3 + p_4.p_3.g_2 + p_4.p_3.p_2.g_1) + p_4.p_3.p_2.p_1.C_{in}^1 \end{aligned} \tag{7.8}$$

Let us now try to derive a pattern, in these results (see Table 7.3).

We observe that for a system of n bits, it is possible to define a generate function (G_n) and a propagate function (P_n). If we are able to somehow precompute these functions, then we can generate C_{out} from C_{in} in a single step. However, as we can see from the example of the 4-bit system, the functions are fairly difficult to compute for large values of n. Let us now derive an interesting property of the generate and propagate functions.

Let us consider a sequence of n bits. Let us divide it into two parts $1 \ldots m$ and $(m+1) \ldots n$. Let the generate and propagate functions for both the parts be $(G_{1,m}, P_{1,m})$ and $(G_{m+1,n}, P_{m+1,n})$ respectively. Furthermore, let the generate and propagate functions for the entire block be $G_{1,n}$ and $P_{1,n}$. We wish to find a relationship between the generate and propagate functions for the whole block with n bits and the functions for the sub blocks.

1 bit	$C^1_{out} = \underbrace{g_1}_{G_1} + \underbrace{p_1}_{P_1}.C^1_{in}$
2 bit	$C^2_{out} = \underbrace{(g_2 + p_2.g_1)}_{G_2} + \underbrace{p_2.p_1}_{P_2}.C^1_{in}$
3 bit	$C^3_{out} = \underbrace{(g_3 + p_3.g_2 + p_3.p_2.g_1)}_{G_3} + \underbrace{p_3.p_2.p_1}_{P_3}.C^1_{in})$
4 bit	$C^4_{out} = \underbrace{(g_4 + p_4.g_3 + p_4.p_3.g_2 + p_4.p_3.p_2.g_1)}_{G_4} + \underbrace{p_4.p_3.p_2.p_1}_{P_4}.C^1_{in})$
n bit	$C^n_{out} = G_n + P_n.C^1_{in}$

Table 7.3: Generate and propagate functions for multi bit systems

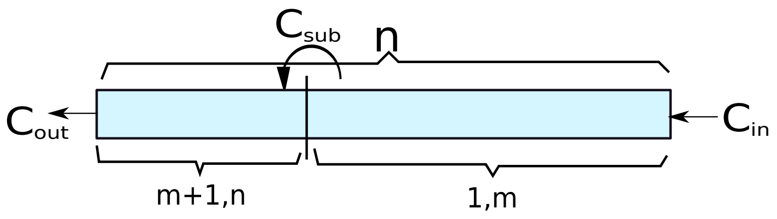


Figure 7.7: A block of n bits divided into two parts

Let the carry out and carry in of the n bit block be C_{out} and C_{in} respectively. Let the carry between the two sub-blocks be C_{sub}. See Figure 7.7. We have:

$$\begin{aligned} C_{out} &= G_{m+1,n} + P_{m+1,n}.C_{sub} \\ &= G_{m+1,n} + P_{m+1,n}.(G_{1,m} + P_{1,m}.C_{in}) \\ &= \underbrace{G_{m+1,n} + P_{m+1,n}.G_{1,m}}_{G_{1,n}} + \underbrace{P_{m+1,n}.P_{1,m}}_{P_{1,n}}.C_{in} \\ &= G_{1,n} + P_{1,n}.C_{in} \end{aligned} \tag{7.9}$$

Thus, for a block of n bits, we can easily compute $G_{1,n}$ and $P_{1,n}$ from the corresponding functions of its sub blocks. This is a very powerful property and is the basis of the carry lookahead adder.

Carry Lookahead Adder – Stage I

The carry lookahead adder's operation is divided into two stages. In the first stage, we compute the generate and propagate functions for different subsequences of bits. In the next stage, we use these functions to generate the result.

The diagram for the first stage is shown in Figure 7.8. Like the carry select adder, we divide bit pairs into blocks. In this diagram, we have considered a block size equal to 2. In the first level, we compute the generate and propagate functions for each block. We build a tree of (G,P) circuits(blocks) as follows. Each (G,P) block in level n takes as input the generate and propagate functions of two blocks in level $n-1$. Thus, at each level the number of (G,P) blocks decreases by a factor of 2. For example, the first (G,P) block in level 1 processes the bit pairs $(1, 2)$. Its parent processes the bit pairs $(1 \dots 4)$, and so on. The ranges are shown in Figure 7.8. We create a tree of (G,P) blocks in this fashion.

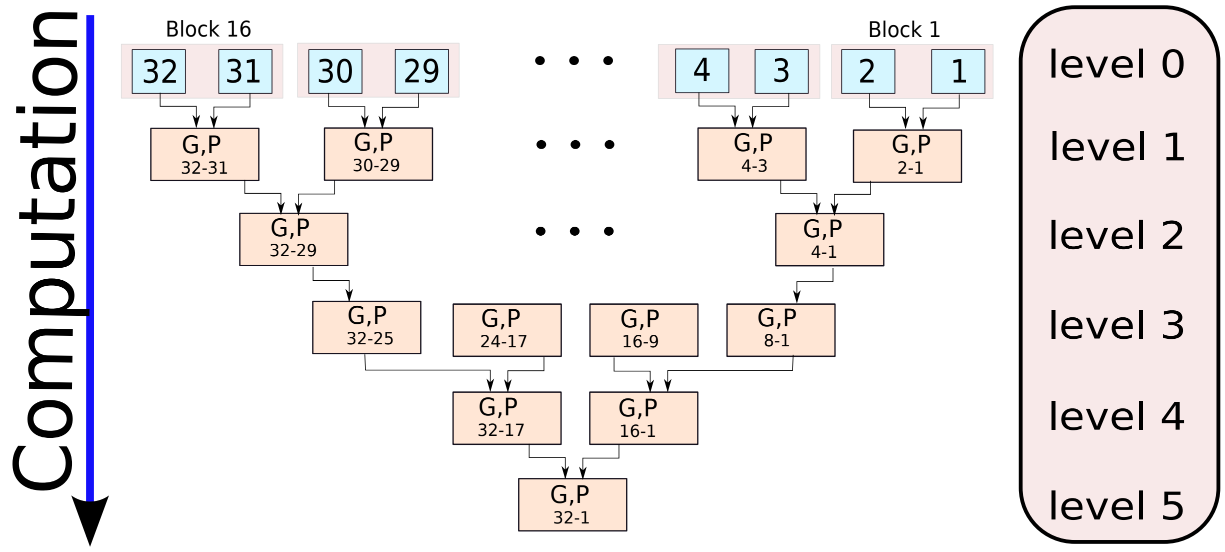


Figure 7.8: Carry Lookahead Adder – Stage I

For a n bit input, there are $O(log(n))$ levels. In each level, we are doing a constant amount of work since each (G,P) block is only processing the inputs from two other blocks. Hence, the time complexity of this stage is equal to $O(log(n))$.

Carry Lookahead Adder – Stage II

In this stage, we use the information generated in Stage I to compute the final sum bits, and the carry out. The block diagram for the second stage is shown in Figure 7.9.

Let us first focus at the rightmost (G,P) blocks in each level. The ranges for each of these blocks start at 1. They take the input carry, C_{in}^1, as input, and then calculate the output carry for the range of bit pairs that they represent as $C_{out} = G + P.C_{in}^1$. When we are adding two numbers, the input carry at the first bit is typically 0. However, some special instructions (ADC in ARM) can consider a non-zero value of C_{in}^1 also.

Each (G,P) block with a range $(r2, r1)$ $(r2 > r1)$, is connected to all (G,P) blocks that have a range of the form $(r3, r2 + 1)$. The output carry of the block is equal to the input carry of those blocks. To avoid excessive clutter in the diagram (Figure 7.9), we show the connections for only the (G,P) block with range (16-1) using solid lines. Each block is connected to the block to its left in the same level and to one (G,P) block in every lower level.

The arrangement of (G,P) blocks represents a tree like computation where the correct carry values propagate from different levels to the leaves. The leaves at level 0, contain a set of 2-bit ripple carry(RC) adders that compute the result bits by considering the correct value of the input carry. We show an example in Figure 7.9 of the correct carry in value propagating from the block with range (16-1) to the 2-bit adder representing the bits 31 and 32. The path is shown using dotted lines.

In a similar manner, carry values propagate to every single ripple carry adder at the zeroth level. The operation completes once all the result bits and the output carry have been computed.

The time complexity of this stage is also $O(log(n))$ because there are $O(log(n))$ levels in the diagram and there is a constant amount of work done per level. This work comprises of computing C_{out} and propagating it to (G,P) blocks at lower levels.

Hence, the total time complexity of the carry lookahead adder is $\boxed{O(log(n))}$.

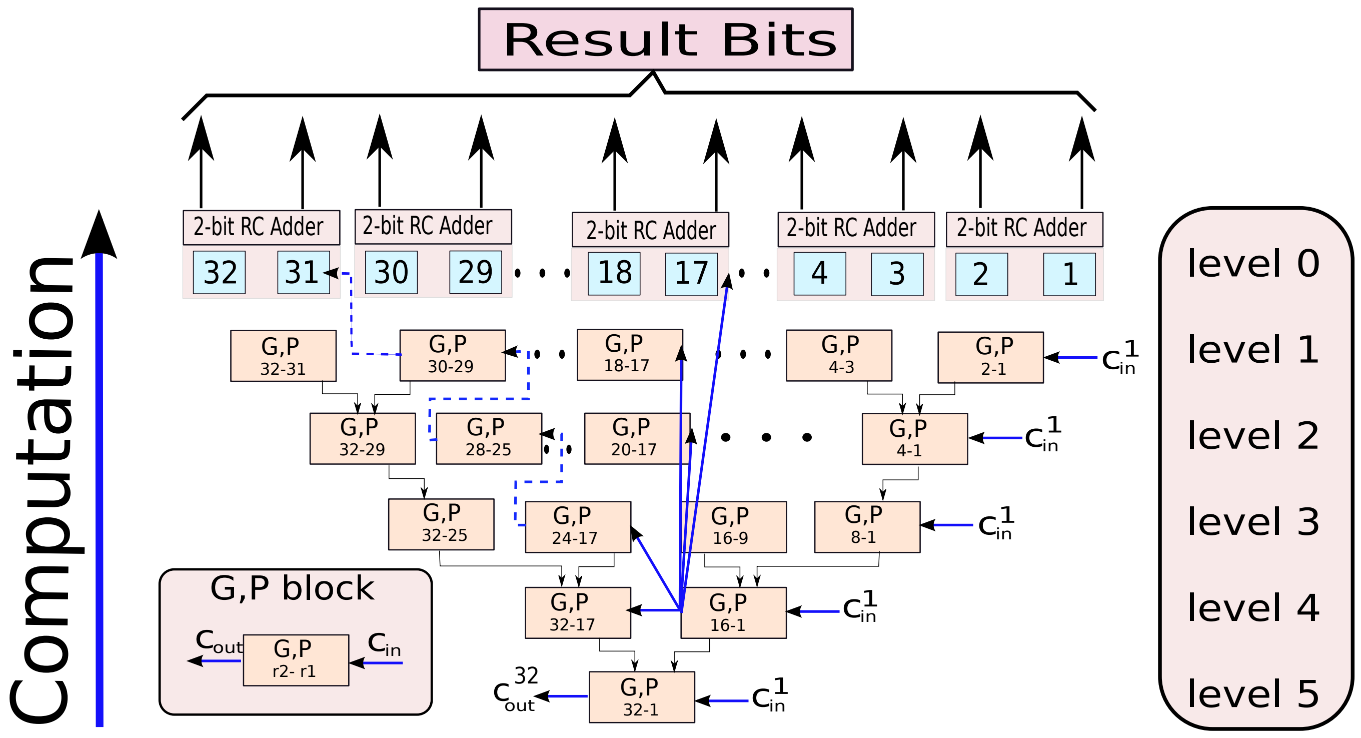


Figure 7.9: Carry Lookahead Adder – Stage II

Way Point 5
Time complexities of different adders:

- *Ripple Carry Adder:* $O(n)$
- *Carry Select Adder:* $O(\sqrt{n})$
- *Carry Lookahead Adder:* $O(log(n))$

7.2 Multiplication

7.2.1 Overview

Let us now consider the classic problem of binary multiplication. Similar to addition, let us first look at the most naive way of multiplying two decimal numbers. Let us try to multiply 13 times 9. In this case, 13 is known as the *multiplicand* and 9 is known as the *multiplier*, and 117 is the *product*.

Figure 7.10(a) shows the multiplication in the decimal number system, and Figure 7.10(b) shows the multiplication in binary. Note that multiplying two binary numbers can be done exactly the same way as decimal numbers. We need to consider each bit of the multiplier from the least significant position to the most significant position. If the bit is 1, then we write the value of the multiplicand below the line, otherwise

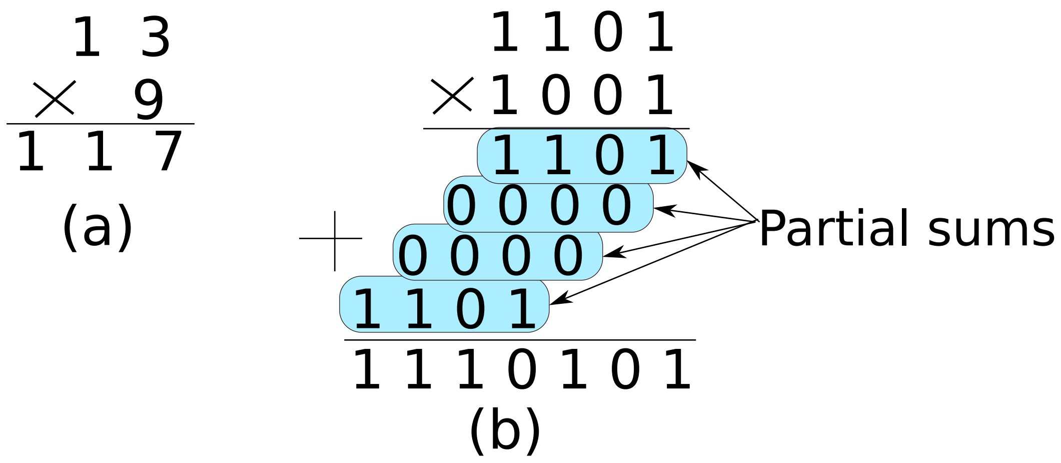


Figure 7.10: Multiplication in decimal and binary

we write 0. For each multiplier bit, we shift the multiplicand progressively one bit to the left. The reason for this is that each multiplier bit represents a higher power of two. We call each such value a *partial sum* (see Figure 7.10(b)). If the multiplier has m bits, then we need to add m partial sums to obtain the product. In this case the product is 117 in decimal and 1110101 in binary. The reader can verify that they actually represent the same number. Let us define another term called the *partial product* for ease of representation later. It is the sum of a contiguous sequence of partial sums.

Definition 56

Partial sum *It is equal to the value of the multiplicand left shifted by a certain number of bits, or it is equal to 0.*

Partial product *It is the sum of a set of partial sums.*

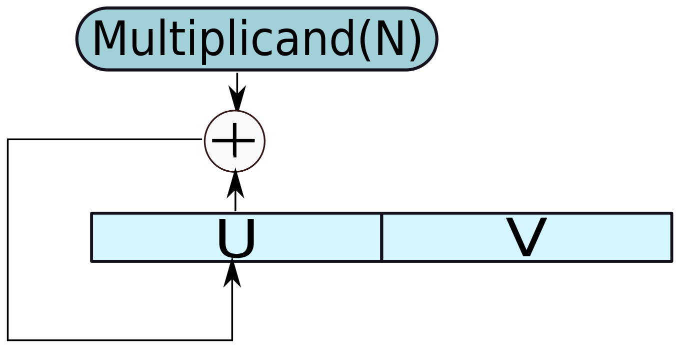


Figure 7.11: Iterative Multiplier

In this example, we have considered unsigned numbers. What about signed numbers? In Section 2.3.4, we proved that multiplying two 2's complement signed n bit binary numbers, and constraining the result to n bits without any concern for overflows, is not different from unsigned multiplication. We need to just multiply the 2's complement numbers without bothering about the sign. The result will be correct.

Let us now consider the issue of overflows in multiplication. If we are multiplying two signed 32-bit values, the product can be as large as $(2^{-31})^2 = 2^{-62}$. There will thus be an overflow if we try to save the result in 32 bits. We need to keep this in mind. If we desire precision, then it is best to allot 64 bits for storing the result of 32-bit multiplication. Let us now look at a naive approach for multiplying two 32-bit numbers by using an *iterative multiplier*.

7.2.2 Iterative Multiplier

In this section, we present the design of an iterative multiplier (see Figure 7.11) that multiplies two signed 32-bit numbers to produce a 64-bit result. We cannot treat the numbers as unsigned anymore and the algorithm thus gets slightly complicated. We use a 33-bit register U, and a 32-bit register V as shown in Figure 7.11. The multiplier is loaded into V at the beginning. The multiplicand is stored separately in register N. The size of the register N is equal to 33 bits, and we store the multiplicand in it by extending its sign by 1 position. The two registers U and V are treated as one large register for the purposes of shifting. If we perform a right shift on U and V, then the value shifted out of U, becomes the MSB of V. We have an adder that adds the multiplicand to the current value of U, and updates U. U is initialised to 0. Let us represent the multiplicand by N, the multiplier by M, and the product by P. We need to compute $P = MN$.

The algorithm used by the iterative multiplier is very similar to the multiplication algorithm that we learnt in elementary school. We need to consider each bit of the multiplier in turn and add a shifted version of the multiplicand to the partial product if the bit is 1. The algorithm is as follows:

Algorithm 1: Algorithm to multiply two 32-bit numbers and produce a 64-bit result

```
   Data: Multiplier in $V$, $U = 0$, Multiplicand in $N$
   Result: The lower 64 bits of $UV$ contains the product
 1 $i \leftarrow 0$
 2 for $i < 32$ do
 3     $i \leftarrow i + 1$
 4     if LSB of V is 1 then
 5         if $i < 32$ then
 6             $U \leftarrow U + N$
 7         end
 8         else
 9             $U \leftarrow U - N$
10         end
11     end
12     $UV \leftarrow UV \gg 1$ (arithmetic right shift)
13 end
```

Let us now try to understand how this algorithm works. We iterate for 32 times to consider each bit of the multiplier. The multiplier is initially loaded into register V.

Now, if the LSB of V is 1 (Line 4), then we add the multiplicand N to U and save the result in U. This basically means that if a bit in the multiplier is equal to 1, then we need to add the multiplicand to the already accumulated partial product. The *partial product* is a running sum of the shifted values of the multiplicands. It is initialised to 0. In the iterative algorithm, the part of UV that does not contain the multiplier, contains the partial product. We then shift UV one step to the right (Line 12). The reason for this is as follows. In each step we actually need to shift the multiplicand 1 bit to the left and add it to the partial product. This is the same as not shifting the multiplicand but shifting the partial product 1 bit to the right assuming that we do not lose any bits. The relative displacement between the multiplicand and the partial product remains the same.

If in any iteration of the algorithm, we find the LSB of V to be 0, then nothing needs to be done. We do not need to add the value of the multiplicand to the partial product. We simply need to shift UV one position to the right using an arithmetic right shift operation.

Note that till the last step we assume that the multiplier is positive. If in the last step we see that the multiplier is not positive (MSB is 1), then we need to subtract the multiplicand from U (Line 9). This follows directly from Theorem 2.3.4.2. The theorem states that the value of the multiplier (M) in the 2's complement notation is equal to $(-M_n 2^{n-1} + \sum_{i=1}^{n-1} M_i 2^{i-1})$. Here M_i is the i^{th} bit of the multiplier, M. In the first $n-1$ iterations, we effectively multiply the multiplicand with $\sum_{i=1}^{n-1} M_i 2^{i-1}$. In the last iteration, we take a look at the MSB of the multiplier, M_n. If it is 0, then we need not do anything. If it is 1, then we need to subtract $2^{n-1} \times N$ from the partial product. Since the partial product is shifted to the right by $n-1$ positions with respect to the multiplicand, the multiplicand is effectively shifted $n-1$ positions to the left with respect to the partial product. To subtract $2^{n-1} \times N$ to the partial product, we need to simply subtract N from register U, which is our last step.

Important Point 8

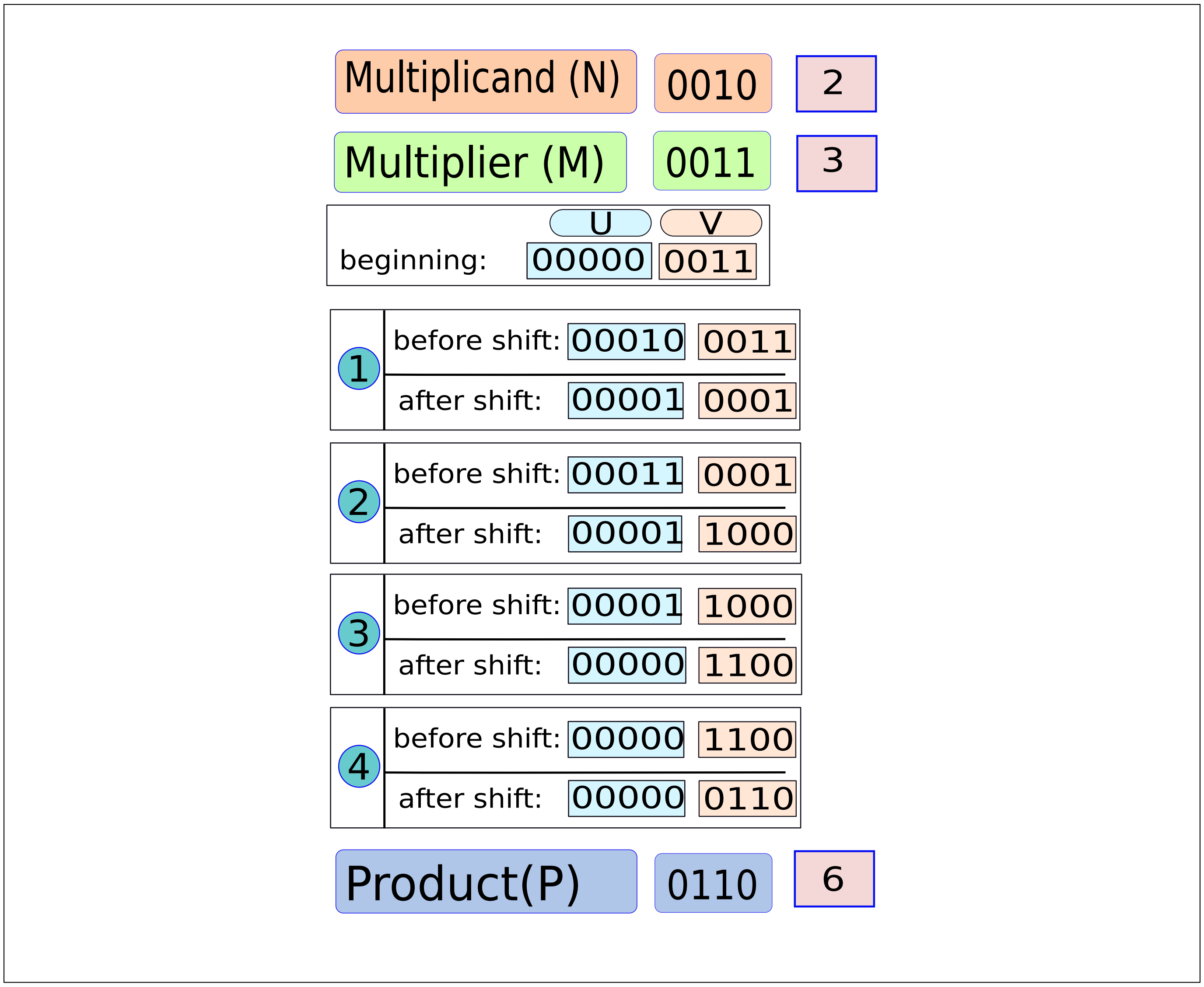
We assume that register U is 33 bits wide. We did this to avoid overflows while adding or subtracting N from U. Let us consider U and N again. $|N| \leq 2^{31}$ *because N is essentially a 32-bit number. For our induction hypothesis, let us assume that* $|U| \leq 2^{31}$ *(true for the base case,* $U = 0$*). Thus* $|U \pm N| \leq 2^{32}$*. Hence, if we store both the numbers and their sum in 33-bit registers, we will never have overflows while adding or subtracting them. Note that we could have had overflows, if we would have used just 32 bits. Now, after the shift operation, the value in U is divided by 2. Since* $U \pm N$ *is assigned to U, and we have established that* $|U \pm N| \leq 2^{32}$*, we can prove that* $|U| \leq 2^{31}$*. Thus, our induction hypothesis holds, and we can thus conclude that during the operation of our algorithm, we shall never have an overflow. The absolute value of the product can at the most be* $2^{31} \times 2^{31} = 2^{62}$*. Hence, the product can fit in 64 bits(proved in Section 7.2.1), and we thus need to only consider the lower 64 bits of the UV register.*

Examples

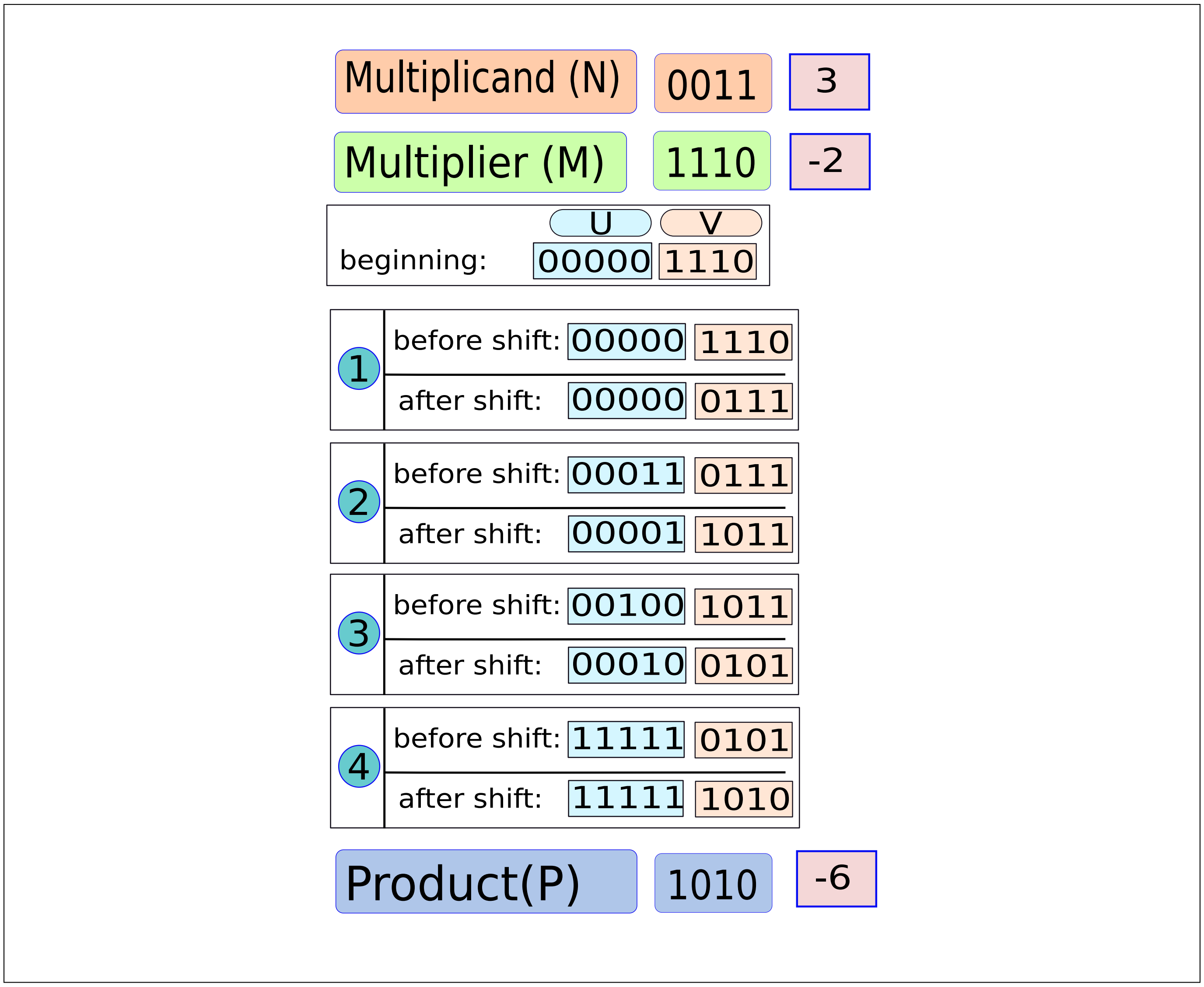
Example 96

Multiply 2×3 *using an iterative multiplier. Assume a 4-bit binary 2's complement number system. Let 2 (*0010_2*) be the multiplicand and let 3 (*0011_2*) be the multiplier. For each iteration show the values of U and V just before the right shift on Line 12, and just after the right shift.*

Answer:



Example 97
Multiply $3 \times (-2)$ *using an iterative multiplier. Assume a 4-bit binary 2's complement number system. Let 3* (0011_2) *be the multiplicand and let -2* (1110_2) *be the multiplier. For each iteration show the values of* U *and* V *just before the right shift on Line 12, and just after the right shift.*
Answer:



Time Complexity

If we are performing n bit multiplication, then there are n iterations of the loop, and each iteration performs one addition at the most. This takes $O(log(n))$ time. Hence, the total time required is $O(nlog(n))$.

7.2.3 Booth Multiplier

The iterative multiplier is a simple algorithm, yet it is slow. It is definitely not as fast as addition. Let us try to speed it up by making a simple alteration. This trick will not change the asymptotic time complexity of the algorithm. However, in practice the process of multiplication will become significantly faster. This algorithm is known as the *Booth* multiplication algorithm and has been used for designing fast multipliers in a lot of processors.

We observe that if a bit in the multiplier is 0, then nothing needs to be done other than a shift in the iterative multiplier. The complexity arises when a bit is 1. Let us assume that the multiplier contains a run of 1s. It is of the form - 0000111100. Let the run of 1s be from the i^{th} to the j^{th} position ($i \leq j$). The value of the multiplier M is thus:

$$M = \sum_{k=i}^{k=j} 2^{k-1} = 2^j - 2^{i-1} \tag{7.10}$$

Now, the iterative multiplier will perform $j-i+1$ additions. This is not required as we can see from Equation 7.10. We just need to do one subtraction when we are considering the i^{th} bit, and do one addition when we are considering the $(j+1)^{th}$ bit. We can thus replace $j-i+1$ additions with one addition and one subtraction. This insight allows us to reduce the number of additions if there are long runs of 1s in the 2's complement notation of the multiplier. If the multiplier is a small negative number, then it fits this pattern. It will have a long run of 1s especially in the most significant positions. Even otherwise, most of the numbers that we encounter will at least have some runs of 1s. The worst case arises, when we have a number of the form: 010101... . This is a very rare case.

If we consider our basic insight again, then we observe that we need to consider bit pairs consisting of the previous and the current multiplier bit. Depending on the bit pair we need to perform a certain action. Table 7.4 shows the actions that we need to perform.

(current value,previous value)	Action
0,0	-
1,0	subtract multiplicand from U
1,1	-
0,1	add multiplicand to U

Table 7.4: Actions in the Booth multiplier

If the current and previous bits are (0,0) respectively, then we do not need to do anything. We need to just shift UV and continue. Similarly, if the bits are (1,1), nothing needs to be done. However, if the current bit is 1, and the previous bit was 0, then a run of 1s is starting. We thus need to subtract the value of the multiplicand from U. Similarly, if the current bit is 0, and the previous bit was 1, then a run of 1s has just ended. In this case, we need to add the value of the multiplicand to U.

Algorithm 2: Booth's Algorithm to multiply two 32-bit numbers to produce a 64-bit result

```
Data: Multiplier in $V$, $U = 0$, Multiplicand in $N$
Result: The lower 64 bits of $UV$ contain the result
1  $i \leftarrow 0$
2  $prevBit \leftarrow 0$
3  for $i < 32$ do
4      $i \leftarrow i + 1$
5      $currBit \leftarrow$ LSB of $V$
6      if $(currBit, prevBit) = (1,0)$ then
7          $U \leftarrow U - N$
8      end
9      else if $(currBit, prevBit) = (0,1)$ then
10         $U \leftarrow U + N$
11     end
12     $prevBit \leftarrow currBit$
13     $UV \leftarrow UV \gg 1$ (arithmetic right shift)
14 end
```

The Booth's algorithm is shown in Algorithm 2. Here, also, we assume that U is 33 bits wide, and V is 32 bits wide. We iterate for 32 times, and consider bit pairs (current bit, previous bit). For (0,0) and (1,1), we do not need to perform any action, else we need to perform an addition and subtraction.

Proof of Correctness*

Let us try to prove that the Booth's algorithm produces the same result as the iterative algorithm for a positive multiplier.

There are two cases. The multiplier (M) can either be positive or negative. Let us consider the case of the positive multiplier first. The MSB of a positive multiplier is 0. Now, let us divide the multiplier into several sequences of contiguous 0s and 1s. For example, if the number is of the form: 000110010111. The sequences are: 000, 11, 00, 1, 0, and 111. For a run of 0s, both the multipliers (Booth's and iterative) produce the same result result because they simply shift the UV register 1 step to the right.

For a sequence of continuous 1s, both the multipliers also produce the same result because the Booth multiplier replaces a sequence of additions with an addition and a subtraction according to Equation 7.10. The only special case arises for the MSB bit, when the iterative multiplier may subtract the multiplicand. In this case, the MSB is 0, and thus no special cases arise. Each run of continuous 0s and 1s in the multiplier is accounted for in the partial product correctly. Therefore, we can conclude that the final result of the Booth multiplier is the same as that of a regular iterative multiplier.

Let us now consider a negative multiplier M. Its MSB is 1. According to Theorem 2.3.4.2, $M = -2^{n-1} + \sum_{i=1}^{n-1} M_i 2^{i-1}$. Let $M' = \sum_{i=1}^{n-1} M_i 2^{i-1}$. Hence, for a negative multiplier (M):

$$M = M' - 2^{n-1} \tag{7.11}$$

M' is a positive number (MSB is 0). Note that till we consider the MSB of the multiplier, the Booth's algorithm does not know if the multiplier is equal to M or M'.

Now, let us split our argument into two cases. Let us consider the MSB bit (n^{th} bit), and the $(n-1)^{th}$ bit. This bit pair can either be 10, or 11.

Case 10: Let us divide the multiplier M into two parts as shown in Equation in Equation 7.11. The first part is a positive number M', and the second part is -2^{n-1}, where $M = M' - 2^{n-1}$. Since the two MSB bits of the binary representation of M are 10, we can conclude that the binary representation of M' contains 00 as its two MSB bits. Recall that the binary representation of M and M' contain the same set of $n-1$ least significant bits, and the MSB of M' is always 0.

Since the Booth multiplier was proven to work correctly for positive multipliers, we can conclude that the Booth multiplier correctly computes the partial product as $N \times M'$ in the first $(n-1)$ iterations. The proof of this fact is as follows. Till the end of $(n-1)$ iterations, we are not sure if the MSB is 0 or 1. Hence, we do not know if we are multiplying N with M or M'. The partial product will be the same in both the cases. If we were multiplying N with M', then no action will be taken in the last step because the two MSB bits of M' are 00. This means that in the second last step ($(n-1)$ iterations), the partial product contains NM'. We can similarly prove that the partial product computed by the iterative multiplier after $(n-1)$ iterations is equal to NM' because the MSB of M' is 0.

Hence, till this point, both the algorithms compute the same partial product, or alternatively have the same contents in the U and V registers. In the last step, both the algorithms find out that the MSB is 1. The iterative algorithm subtracts the multiplicand(N) from U, or alternatively subtracts $N \times 2^{n-1}$ from the partial product. The reason that we treat the multiplicand as shifted by $n-1$ places is because the partial product in the last iteration spans the entire U register and $n-1$ bits of the V register. Now, when we add or subtract the multiplicand(N) to U, effectively, we are adding N shifted by $n-1$ places to the left. Hence, the iterative multiplier correctly computes the product as $M'N - 2^{n-1}N = MN$ (see Equation 7.11). The Booth multiplier also does the same in this case. It sees a $0 \rightarrow 1$ transition. It subtracts N from U, which is exactly the same step as taken by the iterative multiplier. Thus, the operation of the Booth multiplier is correct in this case (same result as the iterative multiplier).

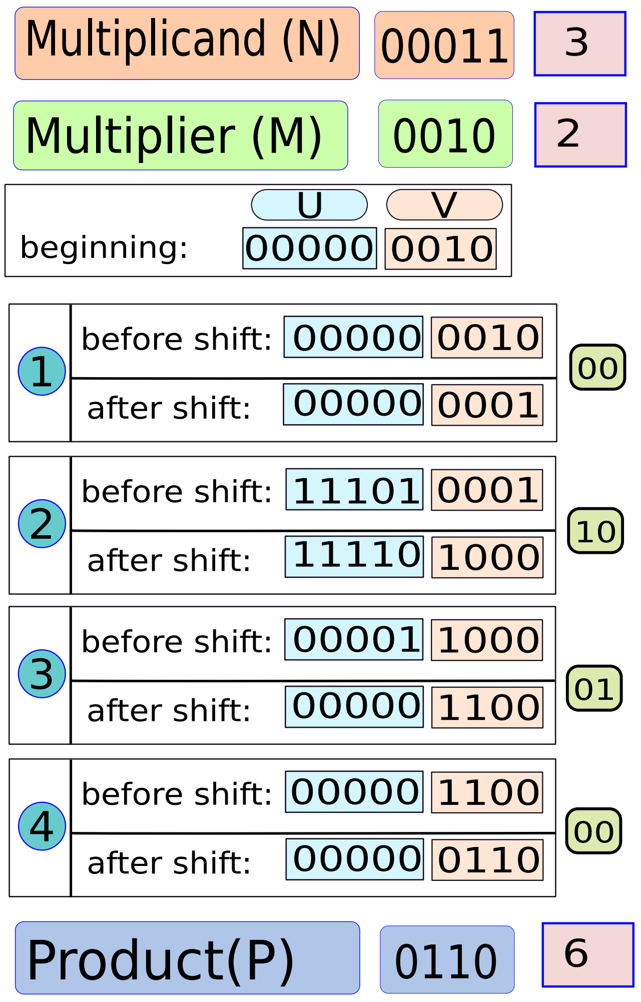
Case 11: Let us again consider the point at the beginning of the n^{th} iteration. At this point of time, the partial product computed by the iterative algorithm is $M'N$, whereas the partial product computed by the Booth multiplier is different because the two MSB bits of M' are 0 and 1, respectively. Let us assume that we were originally multiplying N with M', then the MSB would have been 0, and this fact would have been discovered in the last iteration. The Booth's algorithm would have then added $2^{n-1}N$ to obtain the final result in the last step because of a $1 \rightarrow 0$ transition. Hence, after the $(n-1)^{th}$ iteration, the partial product of the Booth multiplier is equal to $M'N - 2^{n-1}N$. Note that till the last iteration, the Booth multiplier does not know whether the multiplier is M or M'.

Now, let us take a look at the last iteration. In this iteration both the algorithms find out that the MSB is 1. The iterative multiplier subtracts $2^{n-1}N$ from the partial product, and correctly computes the final product as $MN = M'N - 2^{n-1}N$. The Booth multiplier finds the current and previous bit to be 11, and thus does not take any action. Hence, its final product is equal to the partial product computed at the end of the $(n-1)^{th}$ iteration, which is equal to $M'N - 2^{n-1}N$. Therefore, in this case also the outputs of both the multipliers match.

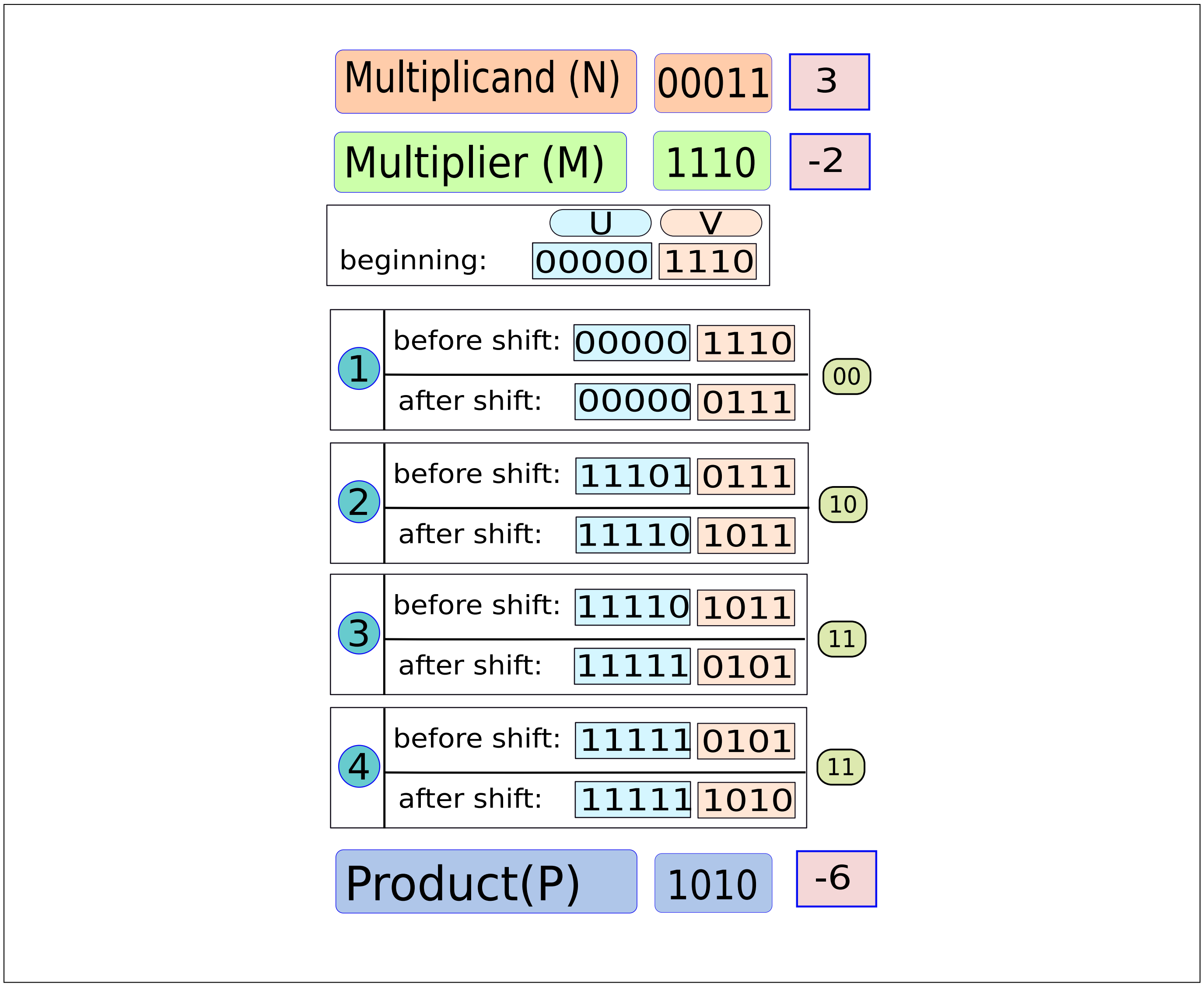
We have thus proved that the Booth multiplier works for both positive and negative multipliers.

Important Point 9
Here, we use a 33-bit U register to avoid overflows. Let us show an example of an overflow, if we would have used a 32-bit U register. Assume that we are trying to multiply -2^{31} *(multiplicand) with -1(multiplier). We will need to compute* $0 - N$ *in the first step. The value of U should be equal to* 2^{31}*; however, this number cannot be represented with 32 bits. Hence, we have an overflow. We do not have this issue when we use a 33-bit U register. Moreover, we can prove that with a 33-bit U register, the additions or subtractions in the algorithm will never lead to an overflow (similar to the proof for iterative multiplication).*

Example 98
Multiply 2×3 *using a Booth multiplier. Assume a 4-bit binary 2's complement number system. Let 3* (0011_2) *be the multiplicand and let 2* (0010_2) *be the multiplier. For each iteration show the values of U and V just before and after the right shift.*
Answer:



Example 99
Multiply $3 \times (-2)$ *using a Booth multiplier. Assume a 4-bit binary 2's complement number system. Let 3* (0011_2) *be the multiplicand and let -2* (1110_2) *be the multiplier. For each iteration show the values of U and V just before and after the right shift.*
Answer:



7.2.4 An $O(log(n)^2)$ Time Algorithm

Let us make our life slightly easier now. Let us multiply two n bit numbers, and save the product as also a n bit number. Let us ignore overflows, and concentrate only on performance. The issue of detecting overflows in a high performance multiplier is fairly complex, and is beyond the scope of this book. Using our results from Section 2.3.4, we use simple unsigned multiplication to compute the product of signed numbers. If there are no overflows then the result is correct.

Let us take a look at the problem of multiplication again. We basically consider each bit of the multiplier in turn, and multiply it with a shifted version of the multiplicand. We obtain n such *partial sums*. The product is the sum of the n partial sums. Generating each partial sum is independent of the other. This process can be performed in parallel in hardware. To generate the i^{th} partial sum, we need to simply compute an AND operation between the i^{th} bit of the multiplier and each bit of the multiplicand. This takes $O(1)$ time.

Now, we can add the n partial sums($P^1 \dots P^n$) in parallel using a tree of adders as shown in Figure 7.12. There are $O(log(n))$ levels. In each level we are adding two $O(n)$ bit numbers; hence, each level takes

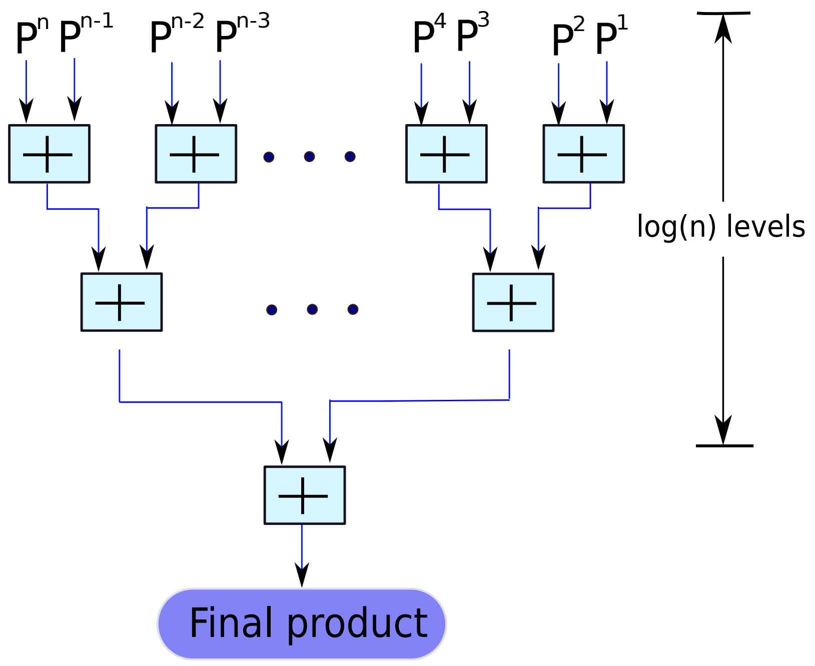


Figure 7.12: Tree of adders for adding partial sums

$O(log(n))$ time. The total time requirement is thus $O(log(n)^2)$. By exploiting the inherent parallelism, we have significantly improved the time from $O(nlog(n))$ to $O(log(n)^2)$. It turns out that we can do even better, and get an $O(log(n))$ time algorithm.

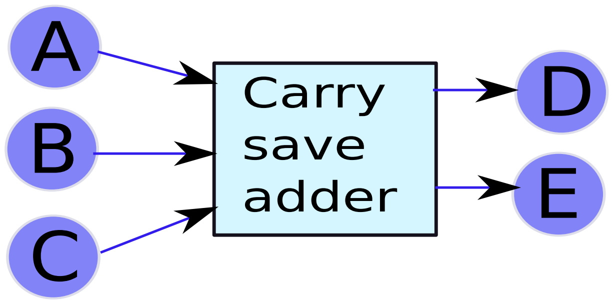


Figure 7.13: Carry Save Adder

7.2.5 Wallace Tree Multiplier

Before, we introduce the Wallace tree multiplier, let us introduce the carry save adder. A carry save adder adds three numbers, A, B, C, and produces two numbers D, and E such that: $A + B + C = D + E$(see Figure 7.13). We will extensively use carry save adders in constructing the Wallace tree multiplier that runs in $O(log(n))$ time.

Carry Save Adder

Let us consider the problem of adding three bits a, b, and c. The sum can range from 0 to 3. We can express all numbers between 0 to 3 in the form $2d + e$, where $(d, e) \in [0, 1]$. Using this relationship, we can express the sum of three numbers as the sum of two numbers as follows:

$$\begin{aligned} A + B + C &= \sum_{i=1}^{n} A_i 2^{i-1} + \sum_{i=1}^{n} B_i 2^{i-1} + \sum_{i=1}^{n} C_i 2^{i-1} \\ &= \sum_{i=1}^{n} (A_i + B_i + C_i) 2^{i-1} \\ &= \sum_{i=1}^{n} (2D_i + E_i) 2^{i-1} \\ &= \underbrace{\sum_{i=1}^{n} D_i 2^i}_{D} + \underbrace{\sum_{i=1}^{n} E_i 2^{i-1}}_{E} \\ &= D + E \end{aligned} \tag{7.12}$$

Thus, we have $A + B + C = D + E$. The question is how to compute the bits D_i, and E_i such that $A_i + B_i + C_i = 2D_i + E_i$. This is very simple. We note that if we add A_i, B_i, and C_i, we get a two bit result, where s is the sum bit and c is the carry bit. The result of the addition can be written as $2 \times c + s$. We thus have two equations as follows:

$$A_i + B_i + C_i = 2D_i + E_i \tag{7.13}$$

$$A_i + B_i + C_i = 2c + s \tag{7.14}$$

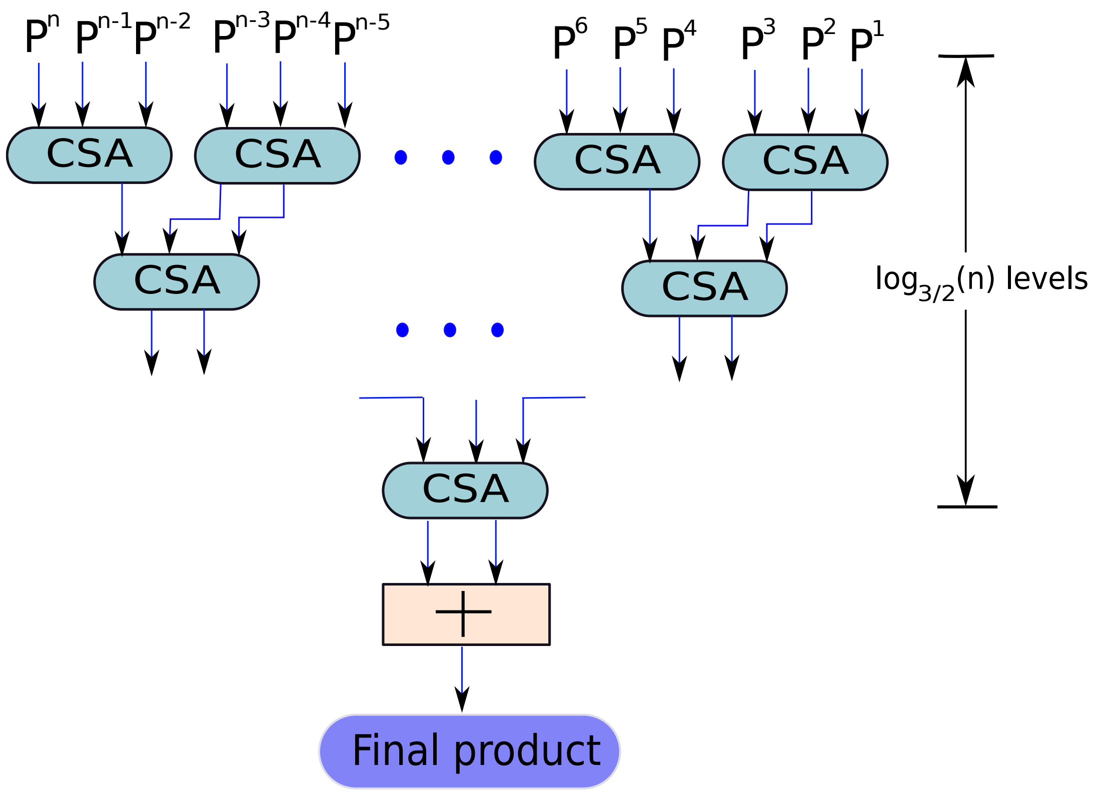
If we set D_i to the carry bit and E_i to the sum bit, then we are done! Now, E is equal to $\sum_{i=1}^{n} E_i 2^{i-1}$. We can thus obtain E by concatenating all the E_i bits. Similarly, D is equal to $\sum_{i=1}^{n} D_i 2^i$. D can be computed by concatenating all the D_i bits and shifting the number to the left by 1 position.

The hardware complexity of a carry save adder is not much. We need n full adders to compute all the sum and carry bits. Then, we need to route the wires appropriately to produce D and E. The asymptotic time complexity of a carry save adder is $O(1)$ (constant time).

Addition of n Numbers with Carry Save Adders

We can use carry save adders to add n partial sums (see Figure 7.14). In the first level, we can use a set of $n/3$ carry save adders to reduce the sum of n partial sums to a sum of $2n/3$ numbers in the second level. If we use $2n/9$ carry save adders in the second level, then we will have $4n/9$ numbers in the third level, and so on. In every level the set of numbers gets reduced by a factor of $2/3$. Thus, after $O(log_{3/2}(n))$ levels, there will only be two numbers left. Note that $O(log_{3/2}(n)$ is equivalent to $O(log(n))$. Since each stage takes $O(1)$ time because all the carry save adders are working in parallel, the total time taken up till now is $O(log(n))$.

In the last stage, when we have just two numbers left, we cannot use a carry save adder anymore. We can use a regular carry lookahead adder to add the two numbers. This will take $O(log(n))$ time. Hence, the total time taken by the Wallace tree multiplier is $O(log(n) + log(n)) = O(log(n))$. In terms of asymptotic time complexity, this is the fastest possible multiplier. It is possible to reduce the number of full adders by slightly modifying the design. This is known as the Dadda multiplier. The reader can refer to [Wikipedia,] for further information on this topic.



P^n P^{n-1} P^{n-2} P^{n-3} P^{n-4} P^{n-5}
P^6 P^5 P^4 P^3 P^2 P^1
CSA
$\log_{3/2}(n)$ levels
+
Final product

Figure 7.14: Wallace Tree Multiplier

7.3 Division

7.3.1 Overview

Let us now look at integer division. Unfortunately, unlike addition, subtraction, and multiplication, division is a significantly slower process. Any division operation can be represented as follows:

$$N = DQ + R \tag{7.15}$$

Here, N is the dividend, D is the divisor, Q is the quotient, and R is the remainder. Division algorithms assume that the divisor and dividend are positive. The process of division needs to satisfy the following properties.

Property 1 $R < D$, and $R \geq 0$.

Property 2 Q is the largest positive integer that satisfies Equation 7.15.

If we wish to divide negative numbers, then we need to first convert them to positive numbers, perform the division, and then adjust the sign of the quotient and remainder. Some architectures try to ensure that the remainder is always positive. In this case, it is necessary to decrement the quotient by 1, and add the divisor to the remainder to make it positive.

Let us focus on the core problem, which is to divide two n bit positive numbers. The MSB is the sign bit, which is 0. Now, $DQ = \sum_{i=1}^{n} DQ_i 2^{i-1}$. We can thus write:

$$\begin{aligned} N &= DQ + R \\ &= DQ_{1\ldots n-1} + DQ_n 2^{n-1} + R \\ \underbrace{(N - DQ_n 2^{n-1})}_{N'} &= D\underbrace{Q_{1\ldots n-1}}_{Q'} + R \\ N' &= DQ' + R \end{aligned} \quad (7.16)$$

We have thus reduced the original problem of division into a smaller problem. The original problem was to divide N by D. The reduced problem is to divide $N' = N - DQ_n 2^{n-1}$ by D. The remainder for both the problems is the same. The quotient, Q', for the reduced problem has the same least significant $n-1$ bits as the original quotient, Q. The n^{th} bit of Q' is 0.

To create the reduced problem it is necessary to find Q_n. We can try out both the choices – 0 and 1. We first try 1. If $N - D2^{n-1} \geq 0$, then $Q_n = 1$ (Property 1 and 2). Otherwise, it is 0.

Once, we have created the reduced problem, we can proceed to further reduce the problem till we have computed all the quotient bits. Ultimately, the divided will be equal to the remainder, R, and we will be done. Let us now illustrate an algorithm that precisely follows the procedure that we have just outlined.

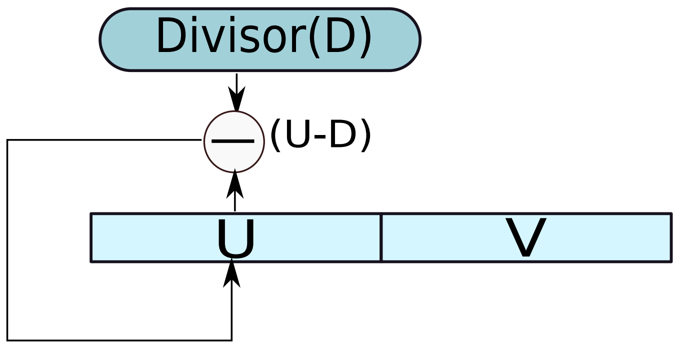


Figure 7.15: Iterative divider

7.3.2 Restoring Division

Let us consider a similar setup as the iterative multiplier to divide two positive 32-bit numbers. The divisor is stored in a 32-bit register called D. We have a 33 bit register U, and a 32-bit register V. If we left shift U and V, then the value shifted out of V, is shifted in to U. U is initialised to 0, and V is initialised to hold the dividend (see Figure 7.15). Note that the size of U is equal to 33 bits to avoid overflows (similar to the case of the iterative multiplier).

Algorithm 3: Restoring algorithm to divide two 32-bit numbers

```
   Data: Divisor in $D$, Dividend in $V$, $U = 0$
   Result: $U$ contains the remainder (lower 32 bits), and $V$ contains the quotient
1  $i \leftarrow 0$
2  for $i < 32$ do
3      $i \leftarrow i + 1$
       /* Left shift $UV$ by 1 position */
4      $UV \leftarrow UV << 1$
5      $U \leftarrow U - D$
6      if $U \geq 0$ then
7          $q \leftarrow 1$
8      end
9      else
10         $U \leftarrow U + D$
11         $q \leftarrow 0$
12     end
       /* Set the quotient bit */
13     LSB of $V \leftarrow q$
14 end
```

Algorithm 3 follows the discussion that we had in Section 7.3.1. We shall see that each iteration of the algorithm reduces the problem according to Equation 7.16. Let us prove its correctness.

Proof of Correctness*

To start out we iterate 32 times for each bit of the dividend (Lines 2 to 14). Let us consider the first iteration. At the beginning, the value in the combined register UV is equal to the value of the dividend N. The first step is to shift UV to the left by 1 position in Line 4. Since the dividend is originally loaded into register V, we are shifting the dividend to the left by 1 position. The next step is to subtract the divisor from register U in Line 5. If $U - D \geq 0$, then we set the MSB of the quotient to 1 (Line 7), otherwise we add D back to U in Line 10, and set the MSB of the quotient to 0.

We wish to use Equation 7.16 to reduce the problem in every iteration. Equation 7.16 states that the new dividend(N') is equal to:

$$N' = N - 2^{n-1}DQ_n \tag{7.17}$$

Q_n is the MSB of the quotient here. The divisor and remainder stay the same. The last $n-1$ bits of the new quotient match those of the old quotient.

We wish to prove that the value of UV at the end of the first iteration is equal to $(2N')$(ignoring quotient bits) such that we can reduce the problem according to Equation 7.16. Let us consider the value stored in UV. Just after executing Line 4, it contains twice the dividend – $2N$ – because we shifted UV by 1 position to the left. Now, we are subtracting D from the upper n bits of UV. In effect, we are subtracting 2^nD. Hence, after Line 5, UV contains $UV - 2^nD$. We have:

$$UV - 2^nD = 2N - 2^nD = 2 \times (N - 2^{n-1}D) \tag{7.18}$$

Subsequently, we test the sign of $U - D$ in Line 6. If $U - D$ is positive or zero, then it means that UV is greater than 2^nD because $V \geq 0$. If $U - D$ is negative, then let $U + \Delta = D$, where $\Delta \geq 1$. We have:

$$\begin{aligned} UV - 2^n D &= 2^n U + V - 2^n D \\ &= (U - D)2^n + V \\ &= V - \Delta \times 2^n \end{aligned} \tag{7.19}$$

Now, $V < 2^n$. Hence, $V < \Delta \times 2^n$, and thus $UV - 2^n D$ is negative. We thus observe that the sign of $U - D$ is the same as the sign of $UV - 2^n D$, which is same as the sign of $(N - 2^{n-1}D)$.

$$sign(U - D) = sign(N - 2^{n-1}D) \tag{7.20}$$

Now, for reducing the problem, if we observe that $U - D \geq 0$, then $N - 2^{n-1}D \geq 0$. Hence, we can set Q_n to 1, and set the new dividend to $N' = N - 2^{n-1}DQ_n$, and also conclude that at the end of the iteration UV contains $2N'$(Line 5 and 7). If $U - D < 0$, then we cannot set Q_n to 1. N' will become negative. Hence, the algorithm sets Q_n to 0 in Line 11 and adds D back to U. The value of UV is thus equal to $2N$. Since $Q_n = 0$, we have $N = N'$(Equation 7.17). In both the cases, the value of UV at the end of the iteration is $2N'$. We thus conclude that in the first iteration, the MSB of the quotient is computed correctly, and the value of UV ignoring the quotient bit is equal to $2N'$.

In the next iteration, we can use exactly the same procedure to prove that the value of UV(ignoring quotient bits) is equal to $4N''$. Ultimately, after 32 iterations, V will contain the entire quotient. The value of UV(ignoring quotient bits) at that point will be $2^n \times N^{32}$. Here N^i is the reduced dividend after the i^{th} iteration. We have the following relation according to Equation 7.17:

$$\begin{aligned} N^{31} &= DQ_1 + R \\ \Rightarrow \underbrace{N^{31} - DQ_1}_{N^{32}} &= R \end{aligned} \tag{7.21}$$

Hence, U will contain the value of the remainder and V will contain the quotient.

Important Point 10

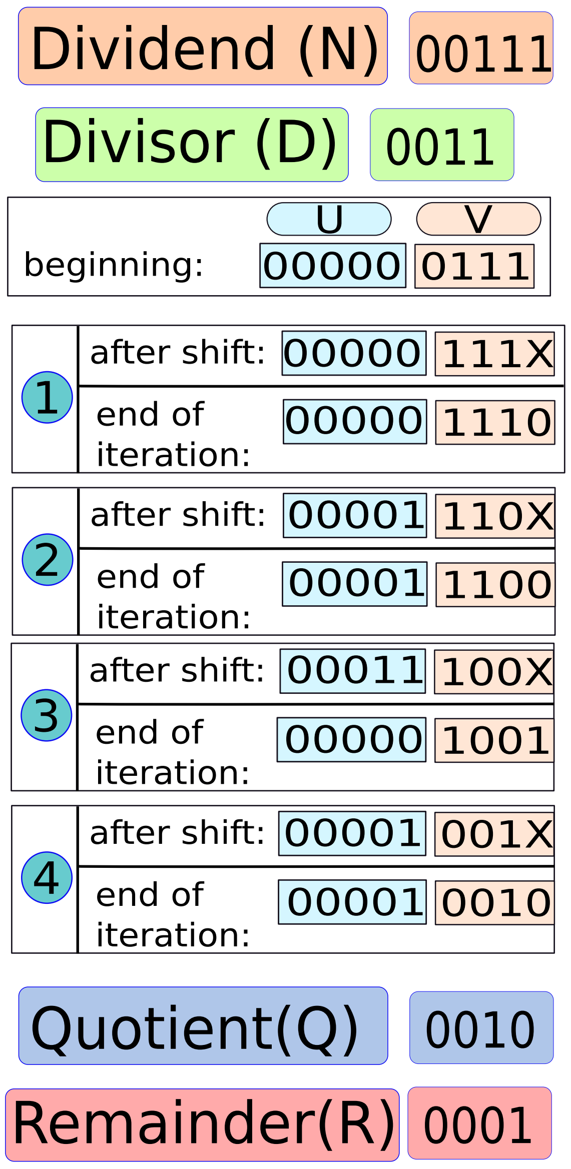
Let us now try to prove that the restoring algorithm does not suffer from overflows while performing a left shift, and adding or subtracting the divisor. Let us first prove that just before the shift operation in Line 4, $U < D$. Let us assume positive divisors ($D > 0$) and non-negative dividends ($N \geq 0$) for division. For the base case, ($U = 0$), the proposition holds. Let us consider the n^{th} iteration. Let the value of U before the shift operation be $\hat{U}$. From the induction hypothesis, we can conclude that $\hat{U} < D$, or alternatively, $\hat{U} \leq D - 1$ After the shift operation, we have $U \leq 2\hat{U} + 1$ because we are performing a left shift by 1 position, and shifting in the MSB of V. If $U < D$, then the induction hypothesis holds for the $(n+1)^{th}$ iteration. Otherwise, we subtract D from U. We have, $U = U - D \leq 2\hat{U} + 1 - D \leq 2(D-1) + 1 - D = D - 1$. Therefore, $U < D$. Thus, for the $(n+1)^{th}$ iteration, the induction hypothesis holds. Now that we have proved that $U < D$, let us prove that the largest value contained in register U is less than or equal to $2D - 1$.

After the shift operation, the largest value that U can contain is $(2(D-1)+1) = 2D - 1$. Henceforth, the value of U can only decrease. Since D is a 32-bit number, we require at the most 33 bits to store $(2D - 1)$. Consequently, having a 33-bit U register avoids overflows.

Time Complexity

There are n iterations of the *for* loop. Each iteration does one subtraction in Line 5 and maybe one addition in Line 10. The rest of the operations can be done in $O(1)$ time. Thus, per iteration it takes $O(log(n))$ time. Hence, the total time complexity is $O(nlog(n))$.

Example 100
Divide two 4-bit numbers: 7 (0111) / 3(0011) using restoring division. ***Answer:***



7.3.3 Non-Restoring Division

We observe that there can be a maximum of two add/subtract operations per iteration. It is possible to circumvent it by using another temporary register to store the result of the subtract operation $U - D$. We can move it to U only if $U - D \geq 0$. However, this also involves additional circuitry. The U register will get complicated and slower too.

The non-restoring algorithm does either one addition or one subtraction per iteration. Hence, it is more efficient even though the asymptotic time complexity is the same. The hardware setup (U and V registers, dividend (N), divisor (D)) is the same as that for the restoring algorithm.

Algorithm 4: Non-restoring algorithm to divide two 32-bit numbers

```
Data: Divisor in $D$, Dividend in $V$, $U = 0$
Result: $U$ contains the remainder, and $V$ contains the quotient
$i \leftarrow 0$
for $i < 32$ do
    $i \leftarrow i + 1$
    /* Left shift $UV$ by 1 position */
    $UV \leftarrow UV << 1$
    if $U \geq 0$ then
        $U \leftarrow U - D$
    end
    else
        $U \leftarrow U + D$
    end
    if $U \geq 0$ then
        $q \leftarrow 1$
    end
    else
        $q \leftarrow 0$
    end
    /* Set the quotient bit */
    LSB of $V \leftarrow q$
end
if $U < 0$ then
    $U \leftarrow U + D$
end
```

We see that the non-restoring algorithm is very similar to the restoring algorithm with some minor differences. The non-restoring algorithm shifts UV as the first step in an iteration. Then, if the value of U is negative, it adds D to U. Otherwise, it subtracts D from U. If the addition or subtraction has resulted in a value that is greater than or equal to zero, the non-restoring algorithm sets the appropriate quotient bit to 1, else it sets it to 0.

Finally, at the end V contains the entire quotient. If U is negative, then we need to add the divisor (D) to U. U will contain the remainder.

Proof of Correctness*

Like the restoring algorithm, let us assume that when we refer to the value of UV, we assume that all the quotient bits are equal to 0. As long as U remains positive or 0, the state of U and V is equal to that produced by the restoring algorithm. Let us assume that in the j^{th} iteration, U becomes negative for the first time. Let us consider the value represented by the register UV just after it is shifted to the left by 1 position, and call it UV_j (j stands for the iteration number).

At the end of the j^{th} iteration, $UV = UV_j - D'$, where $D' = D \times 2^n$. We shift D by n places to the left because we always add or subtract D from U, which is the upper half of UV. According to our assumption UV_j is negative. In this case the restoring algorithm would not have subtracted D', and it would have written 0 to the quotient. The non-restoring algorithm sets the quotient bit correctly since it finds UV to be negative (Line 15). Let us now move to the $(j+1)^{th}$ iteration.

$UV_{j+1} = 2UV_j - 2D'$. At the end of the $(j+1)^{th}$ iteration, $UV = 2UV_j - 2D' + D' = 2UV_j - D'$. If UV is not negative, then the non-restoring algorithm will save 1 in the quotient. Let us now see at this point what the restoring algorithm would have done (assuming UV is non-negative). In the $(j+1)^{th}$ iteration, the restoring algorithm would have started the iteration with $UV = UV_j$. It would have then performed a shift and subtracted D' to set $UV = 2UV_j - D'$, and written 1 to the quotient. We thus observe that at this point the state of the registers U and V matches exactly for both the algorithms.

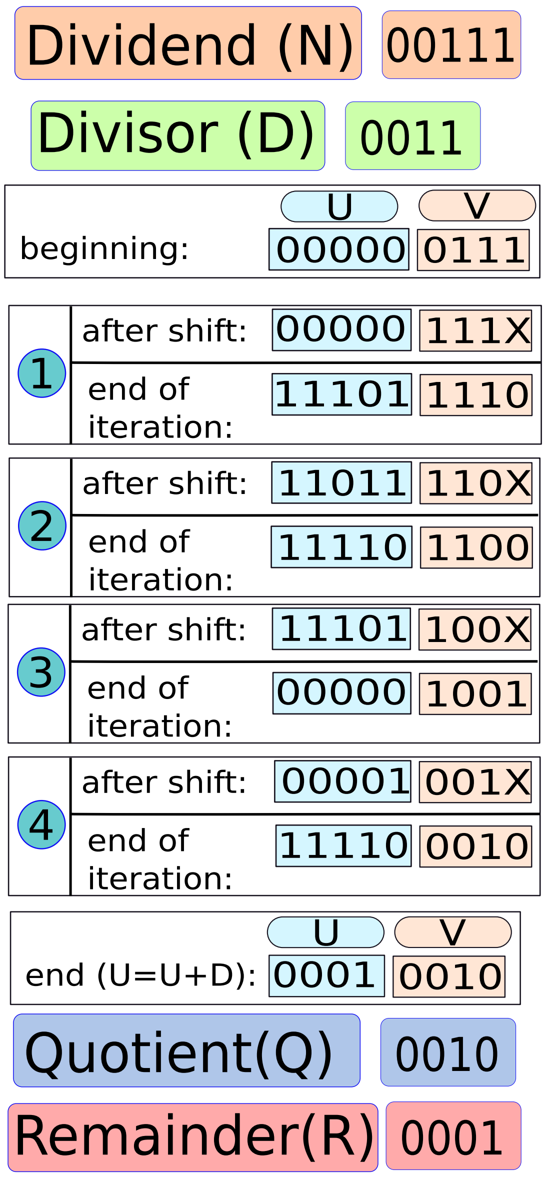
However, if UV is negative then the restoring and non-restoring algorithms will have a different state. Nonetheless the quotient bits will be set correctly. $UV_{j+2} = 4UV_j - 2D'$. Since a negative number multiplied by 2 (left shifted by 1 position) is still negative, the non-restoring algorithm will add D' to U. Hence, the value of UV at the end of the $(j+2)^{th}$ iteration will be $4UV_j - D'$. If this is non-negative, then the restoring algorithm would also have exactly the same state at this point.

We can continue this argument to observe that the quotient bits are always set correctly and the state of U and V exactly matches that of the restoring algorithm when $UV \geq 0$ at the end of an iteration. Consequently, for dividing the same pair of numbers the states of the restoring and non-restoring algorithms will start as the same, then diverge and converge several times. If the last iteration leads to a non-negative UV then the algorithm is correct because the state exactly matches that produced by the restoring algorithm.

However, if the last iteration leaves us with UV as negative, then we observe that $UV = 2^{n-k}UV_k - D'$, where k is the iteration number at which the states had converged for the last time. If we add D' to UV, then the states of both the algorithms match, and thus the results are correct (achieved in Line 20).

Important Point 11
Let us now try to prove that the non-restoring algorithm does not suffer from overflows while performing a left shift, and adding or subtracting the divisor. Similar to the proof for the restoring algorithm, let us first prove that just before the shift operation, $|U| < D$. For the base case, ($U = 0$), the proposition holds. Let us consider the n^{th} iteration, and let the value of U just before the shift operation be $\hat{U}$. Let us first assume that $\hat{U} \geq 0$. In this case, we can use the same logic as the restoring algorithm, and prove that $|U| < D$ at the beginning of the $(n+1)^{th}$ iteration. Let us now assume that $\hat{U} < 0$. From the induction hypothesis, $|\hat{U}| < D \Rightarrow \hat{U} \geq -(D-1)$. Now, if we shift $\hat{U}$ by 1 position, and shift in a 0 or 1, we compute $U \geq 2\hat{U}$ (for $\hat{U} < 0$, shifting in a 1 reduces the absolute value). After the shift operation, we add D to U. We thus have, $U = U + D \geq 2\hat{U} + D \geq 2 \times (1 - D) + D = 2 - D$. Thus, in this case also $|U| < D$, just before the shift, and after the shift we have $|U| \leq 2D - 1$. We thus need to allocate an extra bit to register U to correctly store all the possible intermediate values of U. Hence, the U register is 33 bits wide. We are thus guaranteed to not have overflows during the operation of the non-restoring algorithm.

Example 101
Divide two 4-bit numbers: 7 (0111) / 3(0011) using non-restoring division. ***Answer:***



7.4 Floating Point Addition and Subtraction

The problems of floating point addition and subtraction are actually different faces of the same problem. $A - B$ can be interpreted in two ways. We can say that we are subtracting B from A, or we can say that we are adding $-B$ to A. Hence, instead of looking at subtraction separately, let us look at it as a special case of addition. We shall first look at the problem of adding two numbers with the same sign in Section 7.4.1, with opposite signs in Section 7.4.4 and then look at the generic problem of adding two numbers in Section 7.4.5.

Before going further, let us quickly recapitulate our knowledge of floating point numbers (see Table 7.5).

Normalised form of a 32-bit (normal) floating point number.

$$A = (-1)^S \times P \times 2^{E-bias}, \quad (1 \leq P < 2, E \in \mathbf{Z}, 1 \leq E \leq 254) \qquad (7.22)$$

Normalised form of a 32-bit (denormal) floating point number.

$$A = (-1)^S \times P \times 2^{-126}, \quad (0 \leq P < 1) \qquad (7.23)$$

Symbol	Meaning
S	Sign bit (0(+ve), 1(-ve))
P	Significand (form: 1.xxx(normal) or 0.xxx(denormal))
M	Mantissa (fractional part of significand)
E	(exponent + 127(bias))
$\mathbf{Z}$	Set of integers

Table 7.5: IEEE 754 format

7.4.1 Simple Addition with Same Signs

The problem is to add two floating point numbers A and B with the same sign. We want to compute a new floating point number $C = A + B$. In this case, the sign of the result is known in advance (sign of A or B). All of our subsequent discussion assumes the IEEE 32-bit format. However, the techniques that we develop can be extended to other formats, especially double-precision arithmetic.

First, the floating point unit needs to unpack different fields from the floating point representations of A and B. Let the E fields (exponent + bias) be E_A and E_B for A and B respectively. Let the E field of the result, C, be E_C. In hardware, let us use a register called E to save the exponent (in the bias notation). The final value of E needs to be equal to E_C.

Unpacking the significand is slightly more elaborate. We shall represent the significands as unsigned integers and ignore the decimal point. Moreover, we shall add a leading most significant bit that can act as the sign bit. It is initially 0. For example, if a floating point number is of the form: 1.0111×2^{10}, the significand is 1.0111, and we shall represent it as 010111. Note that we have added a leading 0 bit. Figure 7.16 shows an example of how the significand is unpacked, and placed in a register for a normal floating point number. In the 32-bit IEEE 754 format, there are 23 bits for the mantissa, and there is either a 0 or 1 before the decimal point. The significand thus requires 24 bits, and if we wish to add a leading sign bit(0), then we need 25 bits of storage. Let us save this number in a register, and call it W.

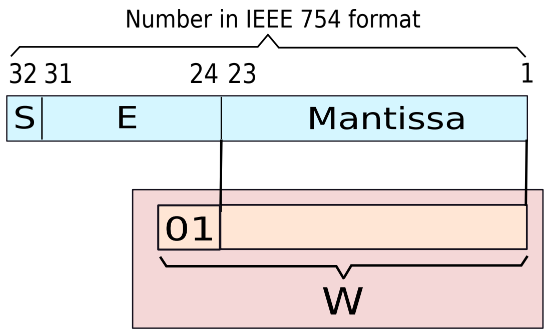


Figure 7.16: Expanding the significand and placing it in a register

Let us start out by observing that we cannot add A and B the way we have added integers, because the exponents might be different. The first task is to ensure that both the exponents are the same. Without no loss of generality, let us assume that $E_A \geq E_B$. This can be effected with a simple compare and swap in hardware. We can thus initialise the register E to E_A.

Let the significands of A and B be P_A and P_B respectively. Let us initially set W equal to the significand of $B(P_B)$ with a leading 0 bit as shown in Figure 7.16.

To make the exponent of A and B equal, we need to right shift W by $(E_A - E_B)$ positions. Now, we can proceed to add the significand of A termed as P_A to W.

$$W = W >> (E_A - E_B) \tag{7.24}$$
$$W = W + P_A \tag{7.25}$$

Let the significand represented by W be P_W. There is a possibility that P_W might be greater than or equal to 2. In this case, the significand of the result is not in normalised form. We will thus need to right shift W by 1 position and increment E by 1. This process is called *normalisation*. There is a possibility that incrementing E by 1 might make it equal to 255, which is not allowed. We can signal an overflow in this case. The final result can be obtained by constructing a floating point number out of the E, W, and the sign of the result (sign of either A or B).

Example 102
Add the numbers: $1.01_2 \times 2^3$ *+* $1.11_2 \times 2^1$. *Assume that the bias is 0.*
Answer:

1. $A = 1.01 \times 2^3$ *and* $B = 1.11 \times 2^1$
2. $W = 01.11$ *(significand of B)*
3. *E = 3*
4. *W = 01.11 >> (3-1) = 00.0111*
5. *W +* P_A *= 00.0111 + 01.0100 = 01.1011*
6. *Result:* $C = 1.1011 \times 2^3$

Example 103
Add 1.01×2^3 *+* 1.11×2^2. *Assume that the bias is 0.*
Answer:

1. $A = 1.01 \times 2^3$ *and* $B = 1.11 \times 2^2$
2. *W = 01.11 (significand of B)*
3. *E = 3*
4. *W = 01.11 >> (3-2) = 00.111*

5. *$W + P_A$ = 00.111 + 01.010 = 10.001*
6. *Normalisation: $W = 10.001 >> 1 = 1.0001$, $E = 4$*
7. *Result: $C = 1.0001 \times 2^4$*

7.4.2 Rounding

In Example 103, let us assume that we were allowed only two mantissa bits. Then, there would have been a need to discard all the mantissa bits other than the two most significant ones. The result would have been 1.00. To incorporate the effect of the discarded bits, it might have been necessary to round the result. For example, let us consider decimal numbers. If we wish to round 9.99 to the nearest integer, then we should round it to 10. Similarly, if we wish to round 9.05 to the nearest integer, then we should round it to 9. Likewise, it is necessary to introduce rounding schemes while doing floating point operations such that the final result can properly reflect the value contained in the discarded bits.

Let us first introduce some terminology. Let us consider the sum of the significands after we have normalised the result. Let us divide the sum into two parts: $(\widehat{P} + R) \times 2^{-23}(R < 1)$. Here, $\widehat{P}$ is the significand of the temporary result in W multiplied by 2^{23}. It is an integer and it might need to be further rounded. R is a residue (beyond 23 bits) that will be discarded. It is less than 1. The aim is to modify $\widehat{P}$ appropriately to take into account the value of R. Now, there are two ways in which $\widehat{P}$ can be modified because of rounding. Either we can leave $\widehat{P}$ as it is, or we can increment $\widehat{P}$ by 1. Leaving $\widehat{P}$ as it is is also known as *truncation*. This is because we are truncating or discarding the residue.

The IEEE 754 format supports four rounding modes as shown in Table 7.6. An empty entry corresponds to truncating the result. We only show the conditions in which we need to increment $\widehat{P}$.

Rounding Mode	Condition for incrementing the significand	
	Sign of the result (+ve)	Sign of the result (-ve)
Truncation		
Round to $+\infty$	$R > 0$	
Round to $-\infty$		$R > 0$
Round to Nearest	$(R > 0.5)\|\|(R = 0.5 \wedge LSB(\widehat{P}) = 1)$	$(R > 0.5)\|\|(R = 0.5 \wedge LSB(\widehat{P}) = 1)$
$\widehat{P}$ (significand of the temporary result multiplied by 2^{23}), $\wedge$ (logical AND), R (residue)		

Table 7.6: IEEE 754 rounding modes

We give examples in decimal (base 10) in the next few subsections for the ease of understanding. Exactly the same operations can be performed on binary numbers. Our aim is to round $\widehat{P} + R$ to an integer. There are four possible ways of doing this in the IEEE 754 format.

Truncation

This is the simplest rounding mode. This rounding mode simply truncates the residue. For example, in truncation based rounding, if $\widehat{P} + R = 1.5$, then we will discard 0.5, and we are left with 1. Likewise, truncating -1.5 will give us -1. This is the easiest to implement in hardware, and is the least accurate out of the four methods.

Round to $+\infty$

In this rounding mode, we always round a number to the larger integer. For example, if $\widehat{P} + R = 1.2$, we round it to 2. If $\widehat{P} + R = -1.2$, we round it to -1. The idea here is to check the sign bit and the residue. If the number is positive, and the residue is non-zero, then we need to increment $\widehat{P}$, or alternatively the LSB of the significand. Otherwise, in all the other cases (either $R = 0$ or the number is negative), it is sufficient to truncate the residue.

Round to $-\infty$

This is the reverse of rounding to $+\infty$. In this case, we round 1.2 to 1, and -1.2 to -2.

Round to Nearest

This rounding mode is the most complicated, and is also the most common. Most processors use this rounding mode as the default. In this case, we try to minimise the error by rounding $\widehat{P}$ to the nearest possible value. If $R > 0.5$, then the nearest integer is $\widehat{P}+1$. For example, we need to round 3.6 to 4, and -3.6 to -4. Similarly, if $R < 0.5$, then we need to truncate the residue. For example, we need to round 3.2 to 3, and -3.2 to -3.

The special case arises when $R = 0.5$. In this case, we would like to round $\widehat{P}$ to the nearest even integer. For example, we will round 3.5 to 4, and 4.5 to also 4. This is more of a convention than a profound mathematical concept. To translate this requirement in our terms, we need to take a look at the LSB of $\widehat{P}$. If it is 0, then $\widehat{P}$ is even, and we do not need to do anything more. However, if $LSB(\widehat{P}) = 1$, then $\widehat{P}$ is odd, and we need to increment it by 1.

7.4.3 Implementing Rounding

From our discussion on rounding, it is clear that we need to maintain some state regarding the discarded bits and $\widehat{P}$ such that we can make the proper rounding decision. In specific, we need four pieces of information – $LSB(\widehat{P})$, is $R = 0.5$, is $R > 0$, and is $R > 0.5$. The last three requirements can be captured with two bits – round and sticky.

The *round* bit is the MSB of the residue, R. The sticky bit is a logical OR of the rest of the bits of the residue. We can thus express the different conditions on the residue as shown in Table 7.7.

Condition on Residue	Implementation
$R > 0$	$r \vee s = 1$
$R = 0.5$	$r \wedge \overline{s} = 1$
$R > 0.5$	$r \wedge s = 1$
r (round bit), s(sticky bit)	

Table 7.7: Evaluating properties of the residue using round and sticky bits

Implementing rounding is thus as simple as maintaining the round bit, and sticky bit, and then using Table 7.6 to round the result. Maintaining the round and sticky bits requires us to simply update them on every single action of the algorithm. We can initialise these bits to 0. They need to be updated when B is shifted to the right. Then, they need to be further updated when we normalise the result. Now, it is possible that after rounding, the result is not in normalised form. For example, if $\widehat{P}$ contains all 1s, then incrementing it will produce 1 followed by 23 0s, which is not in the normalised form.

Renormalisation after Rounding

In case, the process of rounding brings the result to a state that is not in the normalised form, then we need to re-normalise the result. Note that in this case, we need to increment the exponent by 1, and set the mantissa to all 0s. Incrementing the exponent can make it invalid (if $E = 255$). We need to explicitly check for this case.

7.4.4 Addition of Numbers with Opposite Signs

Now let us look at the problem of adding two floating point numbers, A and B, to produce C. They have opposite signs. Again let us make the assumption that $E_A \geq E_B$.

The first step is to load the register W with the significand of $B(P_B)$ along with a leading 0. Since the signs are different, in effect we are subtracting the significand of B (shifted by some places) from the significand of A. Hence, we can take the 2's complement of W that contains P_B with a leading 0 bit, and then shift it to the right by $E_A - E_B$ places. This value is written back to the register W. Note that the shift needs to be an arithmetic right shift here such that the value is preserved. Secondly, the order of operations (shift and 2's complement) is not important.

We can now add the significand of A (P_A) to W. If the resulting value is negative, then we need to take its 2's complement, and set the sign of the result accordingly.

Next, we need to normalise the result. It is possible that $P_W < 1$. In this case, we need to shift W to the left till $1 \leq P_W < 2$. Most implementations of the floating point standard, use an extra bit called the *guard bit.* along with the round and sticky bits. They set the MSB of the residue to the guard bit, the next bit to the round bit, and the OR of the rest of the bits to the sticky bits. During the process of shifting a number left, they shift in the guard bit first, and then shift in 0s. At the end of the algorithm, it is necessary to set the round bit equal to the guard bit, and OR the sticky bit with the round bit such that our original semantics is maintained. This added complexity is to optimise for the case of a left shift by 1 position. If we did not have the guard bit, then we needed to shift the round bit into W, and we would thus lose the round bit forever.

Once W is normalised and the exponent(E) is updated, we need to round the result as per Table 7.6. This process might lead to another round of normalisation.

7.4.5 Generic Algorithm for Adding Floating Point Numbers

Note that we have not considered special values such as 0 in our analysis. The flowchart in Figure 7.17 shows the algorithm for adding two floating point numbers. This algorithm considers 0 values also.

7.5 Multiplication of Floating Point Numbers

The algorithm for multiplying floating point numbers is of exactly the same form as the algorithm for generic addition without a few steps. Let us again try to multiply $A \times B$ to obtain the product C. Again, let us assume without loss of generality that $E_A \geq E_B$.

The flowchart for multiplication is shown in Figure 7.18. We do not have to align the exponents in the case of multiplication. We initialise the algorithm as follows. We load the significand of B into register W. In this case, the width of W is equal to double the size of the operands such that the product can be accommodated. The E register is initialised to $E_A + E_B - bias$. This is because in the case of multiplication, the exponents are added together. We subtract the bias to avoid double counting. Computing the sign of the result is trivial.

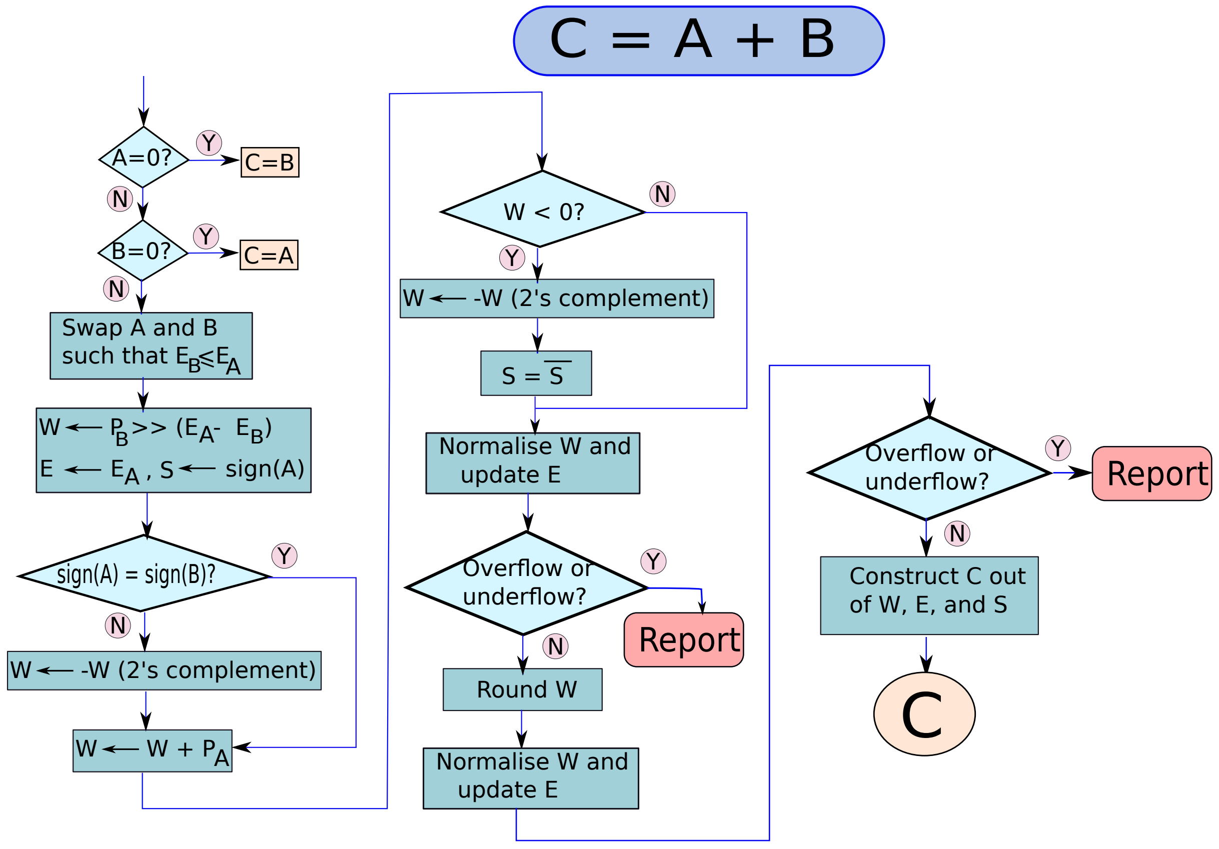


Figure 7.17: Flowchart for adding two floating point values

After initialisation, we multiply the significand of A with W and save the product in W. The product contains 46 bits after the floating point. We might need to discard some of the bits to ensure that the final mantissa is 23 bits long. Hence, it might be necessary to normalise the result by shifting it to the right (normal numbers), or shifting it to the left (denormal numbers).

As with the case of addition, we can then proceed to round the result to contain 23 bits in the mantissa, and renormalise if necessary. Since there are a constant number of add operations, the time complexity is equal to the sum of the time complexity of normal multiplication and addition. Both of them are $O(log(n))$ operations. The total time taken is thus $O(log(n))$.

7.6 Division of Floating Point Numbers

7.6.1 Simple Division

The major difference between integer and floating point division is that floating point division does not have a remainder. It only has a quotient. Let us try to divide A by B to obtain C.

We initialise the algorithm by setting the W register to contain the significand(P_A) of A. The E register is initialised as $E_A - E_B + bias$. This is done because in division, we subtract the exponents. Hence, in their biased representation we need to subtract E_B from E_A, and we need to add the value of the bias back. Computing the sign of the result is also trivial in this case.

We start out by dividing P_A by P_B. The rest of the steps are the same as that of multiplication (see

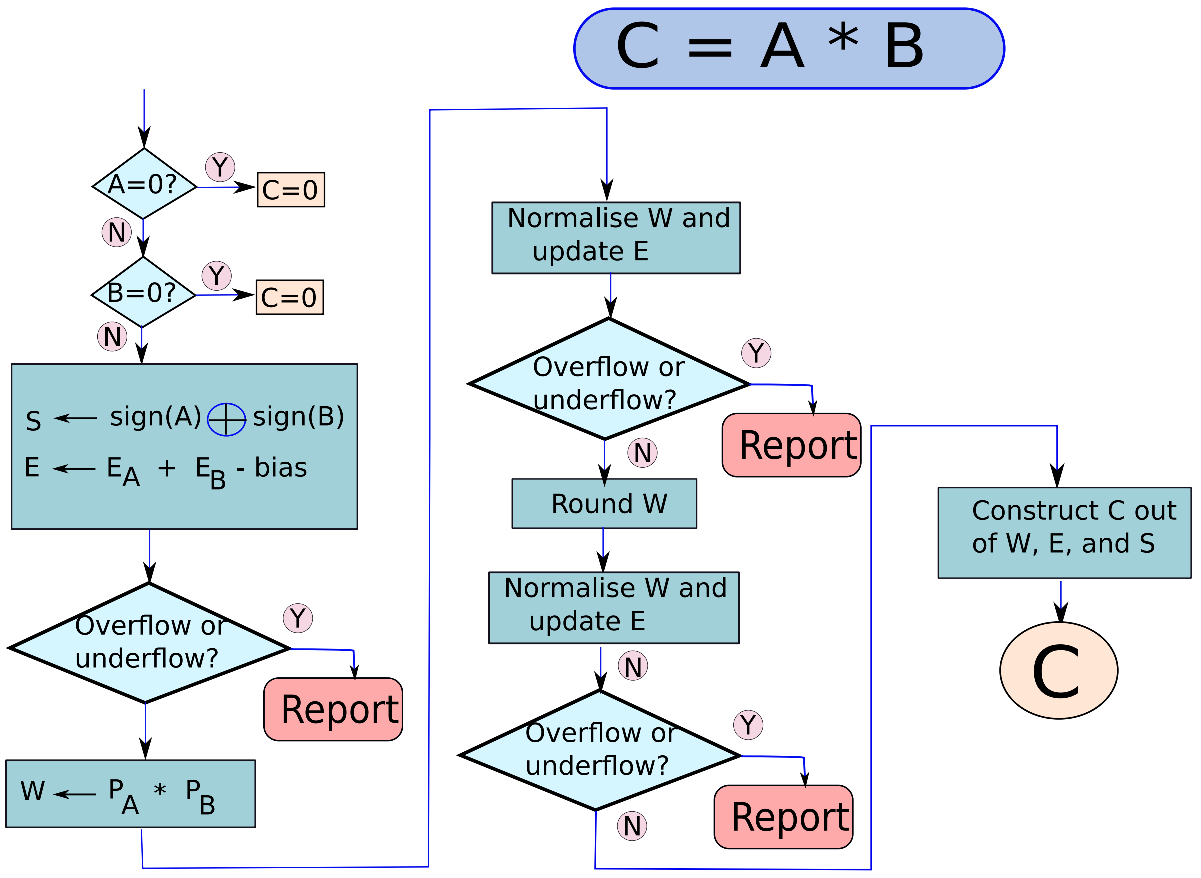


Figure 7.18: Flowchart for multiplying two floating point values

Section 7.5). We normalise the result, round it, and then renormalise if necessary.

The time complexity of this operation is the same as the time complexity of the restoring or non-restoring algorithms. It is equal to $O(nlog(n))$. It turns out that for the case of floating point division, we can do much better.

7.6.2 Goldschmidt Division

Let us try to simplify the process of division by dividing it into two stages. In the first stage, we compute the reciprocal of the divisor $(1/B)$. In the next stage, we multiply the obtained reciprocal with the dividend A. The product is equal to A/B. Floating point multiplication is an $O(log(n))$ time operation. Hence, let us focus on trying to compute the reciprocal of B.

Let us also ignore exponents in our discussion because, we just need to flip the sign of the exponent in the reciprocal. Let us only focus on the significand, P_B, and to keep matters simple, let us further assume that B is a normal floating point number. Thus, $1 \leq P_B < 2$. We can represent $P_B = 1 + X (X < 1)$. We have:

$$\begin{aligned}
\frac{1}{P_B} &= \frac{1}{1+X} \quad (P_B = 1+X, X < 1) \\
&= \frac{1}{1+1-X'} \quad (X' = 1-X, X' < 1) \\
&= \frac{1}{2-X'} \\
&= \frac{1}{2} \times \frac{1}{1-X'/2} \\
&= \frac{1}{2} \times \frac{1}{1-Y} \quad (Y = X'/2, Y < 1/2)
\end{aligned} \tag{7.26}$$

Let us thus focus on evaluating $1/(1-Y)$. We have:

$$\begin{aligned}
\frac{1}{1-Y} &= \frac{1+Y}{1-Y^2} \\
&= \frac{(1+Y)(1+Y^2)}{1-Y^4} \\
&= \ldots \\
&= \frac{(1+Y)(1+Y^2)\ldots(1+Y^{16})}{1-Y^{32}} \\
&\approx (1+Y)(1+Y^2)\ldots(1+Y^{16})
\end{aligned} \tag{7.27}$$

We need not proceed anymore. The reason for this is as follows. Since $Y < 1/2$, Y^n is less than $1/2^n$. The smallest mantissa that we can represent in the IEEE 32-bit floating point notation is $1/2^{23}$. Hence, there is no point in having terms that have an exponent greater than 23. Given the approximate nature of floating point mathematics, the product $(1+Y)(1+Y^2)\ldots(1+Y^{16})$ is as close to the real value of $1/(1-Y)$ as we can get.

Let us now consider the value – $(1+Y)(1+Y^2)\ldots(1+Y^{16})$. It has 5 add operations that can be done in parallel. To obtain $Y \ldots Y^{16}$, we can repeatedly multiply each term with itself. For example, to get Y^8, we can multiply Y^4 with Y^4 and so on. Thus, generating the powers of Y takes 4 multiply operations. Lastly, we need to multiply the terms in brackets – $(1+Y)$, $(1+Y^2)$,$(1+Y^4)$,$(1+Y^8)$, and $(1+Y^{16})$. This will required 4 multiplications. We thus require a total of 8 multiplications and 5 additions.

Let us now compute the time complexity. For an n-bit floating point number, let us assume that a fixed fraction of bits represent the mantissa. Thus, the number of bits required to represent the mantissa is $O(n)$. Consequently, the number of terms of the form $(1+Y^{2k})$ that we need to consider is $O(log(n))$. The total number of additions, and multiplications for finding the reciprocal is also $O(log(n))$. Since each addition or multiplication takes $O(log(n))$ time, the time complexity of finding the reciprocal of B is equal to $O(log(n)^2)$. Since the rest of the operations such as adjusting the exponents and multiplying the reciprocal with A take $O(log(n))$ time, the total complexity is equal to $O(log(n)^2)$.

We observe that floating point division is asymptotically much faster than normal integer division. This is primarily because floating point mathematics is approximate, whereas integer mathematics is exact.

7.6.3 Division Using the Newton-Raphson Method

We detail another algorithm that also takes $O(log(n)^2)$ time. We assume that we are trying to divide A by B. Let us only consider normal numbers. Akin to Goldschmidt division, the key point of the algorithm is to find the reciprocal of the significand of B. Adjusting the exponents, computing the sign bit, and multiplying the reciprocal with A are fast operations ($O(log(n))$).

For readability, let us designate P_B as b ($1 \leq b < 2$). We wish to compute $1/b$. Let us create a function: $f(x) = 1/x - b$. $f(x) = 0$ when $x = 1/b$. The problem of computing the reciprocal of b is thus the same as computing the root of f(x). Let us use the Newton Raphson method [Kreyszig, 2000].

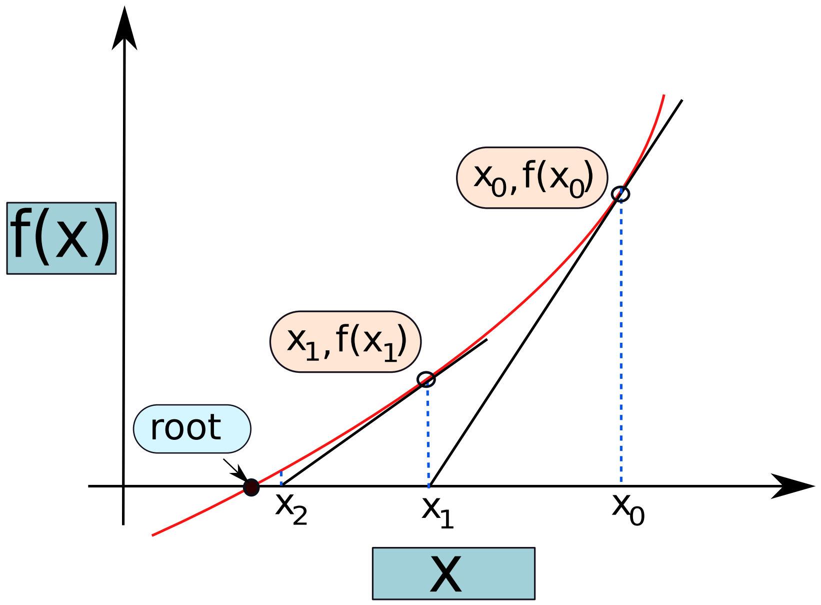


Figure 7.19: The Newton-Raphson method

The gist of this method is shown in Figure 7.19. We start with an arbitrary value of x such as x_0. We then locate the point on $f(x)$ that has x_0 as its x co-ordinate and then draw a tangent to $f(x)$ at $(x_0, f(x_0))$. Let the tangent intersect the x axis at x_1. We again follow the same procedure, and draw another tangent at $(x_1, f(x_1))$. This tangent will intersect the x axis at x_2. We continue this process. As we can observe in the figure, we gradually get closer to the root of f(x). We can terminate after a finite number of steps with an arbitrarily small error. Let us analyse this procedure mathematically.

The derivative of $f(x)$ at x_0 is $df(x)/dx = -1/x_0^2$. Let the equation of the tangent be $y = mx + c$. The slope is equal to $-1/x_0^2$. The equation is thus: $y = -x/x_0^2 + c$. Now, we know that at x_0, the value of y is $1/x_0 - b$. We thus have:

$$\begin{aligned}
&\frac{1}{x_0} - b = -\frac{x_0}{x_0^2} + c \\
\Rightarrow &\frac{1}{x_0} - b = -\frac{1}{x_0} + c \\
\Rightarrow &c = \frac{2}{x_0} - b
\end{aligned} \tag{7.28}$$

The equation of the tangent is $y = -x/x_0^2 + 2/x_0 - b$. This line intersects the x axis when $y = 0$, and $x = x_1$. We thus have:

$$\begin{aligned} 0 &= -\frac{x_1}{x_0^2} + \frac{2}{x_0} - b \\ \Rightarrow x_1 &= 2x_0 - bx_0^2 \end{aligned} \tag{7.29}$$

Let us now define an error function of the form: $\mathcal{E}(x) = bx - 1$. Note that $\mathcal{E}(x)$ is 0, when x is equal to $1/b$. Let us compute the error: $\mathcal{E}(x_0)$ and $\mathcal{E}(x_1)$.

$$\mathcal{E}(x_0) = bx_0 - 1 \tag{7.30}$$

$$\begin{aligned} \mathcal{E}(x_1) &= bx_1 - 1 \\ &= b\left(2x_0 - bx_0^2\right) - 1 \\ &= 2bx_0 - b^2x_0^2 - 1 \\ &= -(bx_0 - 1)^2 \\ &= -\mathcal{E}(x_0)^2 \\ \mid \mathcal{E}(x_1) \mid &= \mid \mathcal{E}(x_0) \mid^2 \end{aligned} \tag{7.31}$$

Thus, the error gets squared every iteration, and if the starting value of the error is less than 1, then it will ultimately get arbitrarily close to 0. If we can place bounds on the error, then we can compute the number of iterations required.

We start out by observing that $1 \leq b < 2$ since we only consider normal floating point numbers. Let x_0 be $1/2$. The range of $bx_0 - 1$ is thus $[-1/2, 0]$. We can thus bound the error($\mathcal{E}(x_0)$) as $-1/2 \leq \mathcal{E}(x_0) < 0$. Therefore, we can say that $\mid \mathcal{E}(x_0) \mid \leq 1/2$. Let us now take a look at the maximum value of the error as a function of the iteration in Table 7.8.

Iteration	max($\mathcal{E}(x)$)
0	$\frac{1}{2}$
1	$\frac{1}{2^2}$
2	$\frac{1}{2^4}$
3	$\frac{1}{2^8}$
4	$\frac{1}{2^{16}}$
5	$\frac{1}{2^{32}}$

Table 7.8: Maximum error vs iteration count

Since we only have 23 mantissa bits, we need not go beyond the fifth iteration. Thus, in this case also, the number of iterations is small, and bounded by a small constant. In every step we have a multiply and subtract operation. These are $O(log(n))$ time operations.

Let us compute the time complexity for n bit floating point numbers. Here, also we assume that a fixed fraction of bits are used to represent the mantissa. Like the case of Goldschmidt division, we need $O(log(n))$ iterations, and each iteration takes $O(log(n))$ time. Thus, the total complexity is $O(log(n)^2)$.

7.7 Summary and Further Reading

7.7.1 Summary

Summary 7

1. *Adding two 1 bit numbers (a and b) produces a sum bit(s) and a carry bit(c)*
   (a) $s = a \oplus b$
   (b) $c = a.b$
   (c) *We can add them using a circuit called a* half adder.
2. *Adding three 1 bit numbers (a, b, and c_{in}) also produces a sum bit(s) and a carry bit(c_{out})*
   (a) $s = a \oplus b \oplus c_{in}$
   (b) $c_{out} = a.b + a.c_{in} + b.c_{in}$
   (c) *We can add them using a circuit called a* full adder.
   (d)
3. *We can create a n bit adder known as a* ripple carry adder *by chaining together $n - 1$ full adders, and a half adder.*
4. *We typically use the notion of asymptotic time complexity to express the time taken by an arithmetic unit such as an adder.*
   (a) $f(n) = O(g(n))$ *if* $|f(n)| \leq c|g(n)|$ *for all* $n > n_0$, *where c is a positive constant.*
   (b) *For example, if the time taken by an adder is given by* $f(n) = 2n^3 + 1000n^2 + n$, *we can say that* $f(n) = O(n^3)$
5. *We discussed the following types of adders along with their time complexities:*
   (a) *Ripple carry adder –* $O(n)$
   (b) *Carry select adder –* $O(\sqrt{n})$
   (c) *Carry lookahead adder –* $O(log(n))$
6. *Multiplication can be done iteratively in* $O(n\,log(n))$ *time using an iterative multiplier. The algorithm is similar to the one we learned in elementary school.*
7. *We can speed it up by using a Booth multiplier that takes advantage of a continuous run of 1s in the multiplier.*
8. *The Wallace tree multiplier runs in* $O(log(n))$ *time. It uses a tree of* carry save adders *that express a sum of three numbers, as a sum of two numbers.*
9. *We introduced two algorithms for division:*
   (a) *Restoring algorithm*
   (b) *Non-restoring algorithm*

10. *Floating point addition and subtraction need not be considered separately. We can have one algorithm that takes care of the generic case.*
11. *Floating point addition requires us to perform the following steps:*
    (a) *Align the significand of the smaller number with the significand of the larger number.*
    (b) *If the signs are different then take a 2's complement of the smaller significand.*
    (c) *Add the significands.*
    (d) *Compute the sign bit of the result.*
    (e) *Normalise and round the result using one of four rounding modes.*
    (f) *Renormalise the result again if required.*
12. *We can follow the same steps for floating point multiplication and division. The only difference is that in this case the exponents get added or subtracted respectively.*
13. *Floating point division is fundamentally a faster operation than integer division because of the approximate nature of floating point mathematics. The basic operation is to compute the reciprocal of the denominator. It can be done in two ways:*
    (a) *Use the Newton-Raphson method to find the root of the equation $f(x) = 1/x - b$. The solution is the reciprocal of b.*
    (b) *Repeatedly multiply the numerator and denominator of a fraction derived from 1/b such that the denominator becomes 1 and the reciprocal is the numerator.*

7.7.2 Further Reading

For more details on the different algorithms for computer arithmetic, the reader can refer to classic texts such as the books by Israel Koren [Koren, 2001], Behrooz Parhami [Parhami, 2009], and Brent and Zimmermann [Brent and Zimmermann, 2010]. We have not covered the SRT division algorithm. It is used in a lot of modern processors. The reader can find good descriptions of this algorithm in the references. The reader is also advised to look at algorithms for multiplying large integers. The Karatsuba and Scönhage-Strassen algorithms are the most popular algorithms in this area. The area of approximate adders is gaining in prominence. These adders add two numbers by assuming certain properties such as a bound on the maximum number of positions a carry propagates. It is possible that they can occasionally make a mistake. Hence, they have additional circuitry to detect and correct errors. With high probability such adders can operate in $O(log(log(n))$ time. Verma et. al. [Verma et al., 2008] describe one such scheme.

Exercises

Addition

Ex. 1 — Design a circuit to find the 1's complement of a number using half adders only.

Ex. 2 — Design a circuit to find the 2's complement of a number using half adders and logic gates.

Ex. 3 — Assume that the latency of a full adder is 2ns, and that of a half adder is 1ns. What is the latency of a 32-bit ripple carry adder?

* **Ex. 4** — Design a carry-select adder to add two n-bit numbers in $O(\sqrt{n})$ time, where the sizes of the blocks are $1, 2, ..., m$ respectively.

Ex. 5 — Explain the operation of a carry lookahead adder.

* **Ex. 6** — Suppose there is an architecture which supports numbers in base 3 instead of base 2. Design a Carry Lookahead Adder for this system. Assume that you have a simple full-adder which adds numbers in base 3.

* **Ex. 7** — Most of the time, a carry does not propagate till the end. In such cases, the correct output is available much before the worst case delay. Modify a ripple carry adder to consider such cases and set an output line to high as soon as the correct output is available.

* **Ex. 8** — Design a fast adder, which uses only the propagate function, and simple logic operations. It should NOT use the generate function. What is its time and space complexity?

Ex. 9 — Design a hardware structure to compute the sum of m, n bit numbers. Make it run as fast as possible. Show the design of the structure. Compute a tight bound on its asymptotic time complexity. [NOTE: Computing the time complexity is not as simple as it seems].

** **Ex. 10** — You are given a probabilistic adder, which adds two numbers and yields the output ensuring that each bit is correct with probability, a. In other words, a bit in the output may be wrong with probability, $(1-a)$, and this event is independent of other bits being incorrect. How will you add two numbers using probabilistic adders ensuring that each output bit is correct with at least a probability of b, where $b > a$?

*** **Ex. 11** — How frequently does the carry propagate to the end for most numbers? Answer: Very infrequently. In most cases, the carry does not propagate beyond a couple of bits. Let us design an approximately correct adder. The insight is that a carry does not propagate by more than k positions most of the time. Formally, we have:
Assumption 1: While adding two numbers, the largest length of a chain of propagates is at most k.

Design an optimal adder in this case that has time complexity $O(log\, k)$ assuming that Assumption 1 holds all the time. Now design a circuit to check if assumption 1 has ever been violated. Verma et. al. [Verma et al., 2008] proved that k is equal to $O(log(n))$ with very high probability. Voila, we have an exactly correct adder, which runs most of the time in $O(log(log(n)))$ time.!!!

*** **Ex. 12** — Let us consider two n-bit binary numbers, A, and B. Further assume that the probability of a bit being equal to 1 is p in A, and q in B. Let us consider $(A + B)$ as one large chunk(block).

(a) What are the expected values of generate and propagate functions of this block as n tends to ∞?

(b) If $p = q = \frac{1}{2}$, what are the values of these functions?

(c) What can we infer from the answer to part (b) regarding the fundamental limits of binary addition?

Multiplication

Ex. 13 — Write a program in assembly language (any variant) to multiply two unsigned 32-bit numbers given in registers $r0$ and $r1$ and store the product in registers $r2$ (LSB) and $r3$ (MSB). Instead of using the multiply instruction, simulate the algorithm of the iterative multiplier.

Ex. 14 — Extend the solution to Exercise 13 for 32-bit signed integers.

* **Ex. 15** — Normally, in the Booth's algorithm, we consider the current bit, and the previous bit. Based on these two values, we decide whether we need to add or subtract a shifted version of the multiplicand. This is known as the radix-2 Booth's algorithm, because we are considering two bits at one time. There is a variation of Booth's algorithm, called radix-4 Booth's algorithm in which we consider 3 bits at a time. Is this algorithm faster than the original radix-2 Booth's algorithm? How will you implement this algorithm ?

* **Ex. 16** — Assume that in the sizes of the U and V registers are 32 bits in a 32-bit Booth multiplier. Is it possible to have an overflow? Answer the question with an example. [HINT: Can we have an overflow in the first iteration itself?]

* **Ex. 17** — Prove the correctness of the Booth multiplier in your own words.

Ex. 18 — Explain the design of the Wallace tree multiplier. What is its asymptotic time complexity?

** **Ex. 19** — Design a Wallace tree multiplier to multiply two signed 32-bit numbers, and save the result in a 32-bit register. How do we detect overflows in this case?

Division

Ex. 20 — Implementation of division using an assembly program.

i) Write an assembly program for restoring division.

ii) Write an assembly program for non-restoring division.

* **Ex. 21** — Design an $O(log(n)^k)$ time algorithm to find out if a number is divisible by 3. Try to minimise k.

* **Ex. 22** — Design an $O(log(n)^k)$ time algorithm to find out if a number is divisible by 5. Try to minimise k.

** **Ex. 23** — Design a fast algorithm to compute the remainder of the division of an unsigned number by a number of the form $(2^m + 1)$. What is its asymptotic time complexity?

** **Ex. 24** — Design a fast algorithm to compute the remainder of the division of an unsigned number by a number of the form $(2^m - 1)$. What is its asymptotic time complexity?

** **Ex. 25** — Design an $O(log(uv)^2)$ algorithm to find the greatest common divisor of two binary numbers u and v. [HINT: The gcd of two even numbers u and v is $2 * gcd(u/2, v/2)$]

Floating Point Arithmetic

Ex. 26 — Give the simplest possible algorithm to compare two 32-bit IEEE 754 floating point numbers. Do not consider $\pm\infty$, NAN, and (negative 0). Prove that your algorithm is correct. What is its time complexity ?

Ex. 27 — Design a circuit to compute $\lceil log_2(n) \rceil$. What is its asymptotic time complexity? Assume n is an integer. How can we use this circuit to convert n to a floating point number?

Ex. 28 — A and B, are saved in the computer as A' and B'. Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors of the factors.

Ex. 29 — Explain floating point addition with a flowchart.

Ex. 30 — Explain floating point multiplication with a flowchart.

Ex. 31 — Can we use regular floating point division for dividing integers also? If not, then how can we modify the algorithm for performing integer division?

Ex. 32 — Describe in detail how the "round to nearest" rounding mode is implemented.

*** **Ex. 33** — We wish to compute the square root of a floating point number in hardware using the Newton-Raphson method. Outline the details of an algorithm, prove it, and compute its computational complexity. Follow the following sequence of steps.

1. Find an appropriate objective function.
2. Find the equation of the tangent, and the point at which it intersects the x-axis.
3. Find an error function.
4. Calculate an appropriate initial guess for x.
5. Prove that the magnitude of the error is less than 1.
6. Prove that the error decreases at least by a constant factor per iteration.
7. Evaluate the asymptotic complexity of the algorithm.

Design Problems

Ex. 34 — Implement an adder and a multiplier in a hardware description language such as VHDL or Verilog.

Ex. 35 — Extend your design for implementing floating point addition and multiplication.

Ex. 36 — Read about the SRT division algorithm, comment on its computational complexity, and try to implement it in VHDL/Verilog.