

Chapter 3

Arithmetic for Computers

Reading: The corresponding chapter in the 2nd edition is Chapter 4, in the 3rd edition it is Chapter 3 and in the 4th edition it is Chapter 3.

3.1 Representing Numbers

To calculate the sum of, e.g., the number 7 and the number 4 with a MIPS R2000 program, an `add` instruction can be used:

```
add $6, $4, $5
```

This instructs the CPU to “add the two numbers stored in the registers `$4` and `$5` and store the result in `$6`”. This is more involved than it may seem. The registers `$4` and `$5` each contain 1 word, i.e., 32 bits, as discussed in the previous chapter. These 32 bits can represent 2^{32} different 32-bit patterns. Therefore, each 32-bit pattern can encode one out of 2^{32} possibilities (numbers, characters, etc.). So, first, we need to define which 32-bit patterns correspond to the non-negative integers 7 and 4 and store those in `$4` and `$5`. Second, the hardware for the adder needs to be designed such that, with 7 and 4 stored in their respective 32-bit representations, in `$4` and `$5`, it computes the 32-bit representation of the correct result, i.e., the number 11 and stores that 32-bit word in `$6`.

Therefore, in this chapter, we will discuss how numbers can be represented with 32-bit words, and how addition and multiplication can correctly be implemented in hardware. Issues that arise when using a finite representation (32 bits, allowing 2^{32} bit patterns) to represent infinite number systems (natural numbers, integers, etc.) will also be presented. For example, overflow will occur when the result of a calculation is a number that falls outside the range of numbers that can be represented using a given 32-bit representation.

3.1.1 Natural numbers

With 32 bits, we can represent 2^{32} different natural numbers. To relate each of the 2^{32} 32-bit patterns to one of 2^{32} different natural numbers, one can make the following associations,

following the natural counting order, starting at 0, all the way up to $2^{32} - 1$:

$$\begin{array}{l}
 0_{10} \iff \overbrace{000 \cdots 0000}_{{32\text{bits}}}_2 \\
 1_{10} \iff 000 \cdots 0001_2 \\
 2_{10} \iff 000 \cdots 0010_2 \\
 3_{10} \iff 000 \cdots 0011_2 \\
 4_{10} \iff 000 \cdots 0100_2 \\
 5_{10} \iff 000 \cdots 0101_2 \\
 6_{10} \iff 000 \cdots 0110_2 \\
 7_{10} \iff 000 \cdots 0111_2 \\
 \dots \\
 (2^{32} - 1)_{10} \iff 111 \cdots 1111_2
 \end{array}$$

Following this convention, the natural numbers 4 and 7 can be stored in two registers, as $000 \cdots 0100_2$ and $000 \cdots 0111_2$, respectively. The subscript indicates whether the representation is binary (subscript 2) or decimal (subscript 10). A leading “0x” indicates hexadecimal representation. For example, $16_{10} = 10000_2 = 0x10$. For natural numbers, the value of the number represented by a particular 32-bit pattern can be computed as:

$$10000_2 \rightarrow 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0,$$

or, in general, for a 32-bit pattern $a = a_{31}a_{30}\dots a_1a_0$, the value is given by:

$$\sum_{i=0}^{31} a_i 2^i.$$

Natural numbers will typically be used in MIPS R2000 to address memory (with 32 bits, we can address 4GB of memory).

To add two integer numbers, e.g., 7 and -4 , a representation for negative numbers is needed. To add real numbers, such as 3.141592 and $\sqrt{2}$, or fractions as $\frac{3}{7}$ and $\frac{2}{17}$, one first needs to define how to represent such numbers using 32-bit words. A good representation is one that allows simple hardware (electronic circuits manipulating bits, i.e., 0s and 1s) to compute the 32-bit representation of a result (sum, product, etc.) from the 32-bit representation of the arguments. The particular representation will determine the **range** of numbers that can be represented and how to detect **overflow**, i.e., that the result of a computation is in fact outside that range.

3.1.2 Integers

In this subsection, we will study how to represent positive and negative integers using 32-bit words. The idea to guide our choice of representation is based on whether it allows simple,

universal hardware to implement a 32-bit **adder**. More precisely, to add 2 integers, e.g., -3 and 4 , we store the 32-bit representations of -3 and 4 in the registers **\$4** and **\$5** respectively and then execute the instruction

$$\text{add } \$6, \$4, \$5$$

This will feed the 32-bit words stored in **\$4** and **\$5** into the CPU's **adder** (hardware used to compute a sum). The adder will output a 32-bit word, which will be stored in **\$6**. The sum has been computed successfully if the 32-bit word in **\$6** indeed represents the sum of -3 and 4 , i.e., the number 1 . The complexity of the hardware that implements the adder will depend on how we decide to represent integers with 32-bit words. We are looking for a representation that will make the adder hardware as simple as possible. Historically, three different conventions to represent integers have been proposed:

1. Sign-magnitude representation
2. One's complement representation
3. Two's complement representation

Sign-magnitude representation

The sign-magnitude representation is very intuitive: the most significant bit (MSB) represents the sign of the integer, and the remaining bits represent the magnitude (i.e., absolute value) of the integer. For example, using a 4-bit sign-magnitude representation, we have:

$$\begin{aligned} 3_{10} &= 0011_2 \Rightarrow \text{MSB } 0 \text{ means "positive"} \\ -3_{10} &= 1011_2 \Rightarrow \text{MSB } 1 \text{ means "negative"} \end{aligned}$$

Note that 0 can be represented in two different ways:

$$\begin{aligned} 0_{10} &= 0000_2 \\ -0_{10} &= 1000_2 \end{aligned}$$

The sum of two positive integers can be computed as

$$\begin{array}{r} 3_{10} \\ + 4_{10} \\ \hline 7_{10} \end{array} \Rightarrow \begin{array}{r} 0011_2 \\ 0100_2 \\ \hline 0111_2 \end{array}$$

Given the 4-bit sign-magnitude representation of the numbers 3 and -4 , applying a bit-wise longhand addition for all except the magnitude bit and simply copying the magnitude bit (as the sum of two positive integers will be positive) results in the sign-magnitude representation of the correct result, i.e., the number $7 = 3 + 4$ since $0111_2 = 7_{10}$. The same method computes the sum of two negative integers:

$$\begin{array}{r} -3_{10} \\ + -4_{10} \\ \hline -7_{10} \end{array} \Rightarrow \begin{array}{r} 1011_2 \\ 1100_2 \\ \hline 1111_2 \end{array}$$

Again, copying the magnitude bit (as the sum of two negative numbers is negative) and applying a bit-wise longhand addition for all other bits renders the representation of the correct result, i.e., $1111_2 = -7_{10}$.

However, the same method cannot be used to add two numbers with different signs. To know the sign of the result, the adder would first have to compare the magnitude of both numbers, subtract the smallest magnitude from the largest magnitude and assign the sign of the argument with largest magnitude to the sum. Clearly, using the sign-magnitude representation will complicate the adder hardware.

For example, $3_{10} + (-4_{10})$ will result in a negative sum, so the sum's MSB should be 1. The other bits are computed by subtracting the magnitudes: $4 - 3 = 1 \Rightarrow 001_2$.

$$\begin{array}{r} 3_{10} \\ + -4_{10} \\ \hline -1_{10} \end{array} \Rightarrow \begin{array}{r} 0011_2 \\ 1100_2 \\ \hline 1001_2 \end{array}$$

One's complement representation

In one's complement representation, positive integers are represented as in sign-magnitude representation. Negative integers are represented by flipping all the bits of the representation of the corresponding positive integers. For example, in 4-bit one's complement representation, we have:

$$\begin{aligned} 3_{10} &= 0011_2 \\ -3_{10} &= 1100_2 \\ 7_{10} &= 0111_2 \\ -7_{10} &= 1000_2 \end{aligned}$$

Note that 0 can still be represented in two different ways:

$$\begin{aligned} 0_{10} &= 0000_2 \\ -0_{10} &= 1111_2 \end{aligned}$$

As for the sign-magnitude representation, we can easily check whether a number is positive or negative by examining the MSB. Adder hardware using the one's complement representation is still not really simple though.

Two's complement representation

Positive integers are represented as in sign-magnitude and one's complement representation. Generating the representation of a negative integer is done in two steps:

- Complement all bits of the representation of the corresponding positive integer

- Add 1

For example, in 4-bit two's complement representation, we have:

$$\begin{aligned} 3_{10} &= 0011_2 \\ -3_{10} &= 1100_2 + 1 = 1101_2 \\ 7_{10} &= 0111_2 \\ -7_{10} &= 1000_2 + 1 = 1001_2 \end{aligned}$$

The MSB still represents the sign bit and there is only one representation of the number 0:

$$\begin{aligned} 0_{10} &= 0000_2 \\ -0_{10} &= 1111_2 + 1 = 0000_2 \text{ as the 5th bit is discarded} \end{aligned}$$

There is one more negative number than there are positive numbers in 4-bit two's complement representation: $-8_{10} = 1000_2$. Most importantly, adding integers represented in two's complement representation can be done using a simple long hand scheme, regardless of the signs.

$$\begin{array}{rcl} & 3_{10} & 0011_2 \\ + & 4_{10} & \underline{0100_2} \\ \hline & 7_{10} & 0111_2 \end{array} \Rightarrow$$

$$\begin{array}{rcl} & -3_{10} & 1101_2 \\ + & -4_{10} & \underline{1100_2} \\ \hline & -7_{10} & 1001_2 \end{array} \Rightarrow$$

$$\begin{array}{rcl} & 3_{10} & 0011_2 \\ + & -4_{10} & \underline{1100_2} \\ \hline & -1_{10} & 1111_2 \end{array} \Rightarrow$$

$$\begin{array}{rcl} & -3_{10} & 1101_2 \\ + & 4_{10} & \underline{0100_2} \\ \hline & 1_{10} & 0001_2 \end{array} \Rightarrow$$

Indeed, even when the signs are different, the two's complement representation of the sum is correctly obtained by applying a bit-wise long hand addition scheme to the two's complement representation of the arguments. Therefore, as we will see, we can add two integers using simple adder hardware, when using two's complement representation. In what comes next we will mathematically prove this.

Below is a table showing the associations between all 2^4 4-bit patterns and the values represented, in 4-bit two's complement representation. To obtain the table, we first write down the 4-bit patterns for all positive numbers (0 through 7), than invert each one of those (using the two's complement rule) to obtain the 4-bit patterns of the corresponding negative numbers (-7 through 0) and then add the extra negative number the representation allows, i.e., 1000_2 , representing -8.

Integer	Two's complement
-8	1000 ₂
-7	1001 ₂
-6	1010 ₂
-5	1011 ₂
-4	1100 ₂
-3	1101 ₂
-2	1110 ₂
-1	1111 ₂
0	0000 ₂
1	0001 ₂
2	0010 ₂
3	0011 ₂
4	0100 ₂
5	0101 ₂
6	0110 ₂
7	0111 ₂

We note that, for example, the 4-bit two's complement representation of -5 is 1011₂. If 1011₂ were representing a natural number, it would be representing the number 11. We can see that $11 = 16 - 5 = 2^4 - 5$. This is not a coincidence. In fact, in the next section, we show that the 4-bit two's complement representation of $-X$ (with $X \geq 0$) can be obtained as the 4-bit natural number representation of $2^4 - X$. Similarly, the 4-bit two's complement representation of -7 can be obtained as the 4-bit natural number representation of $2^4 - 7 = 9$, which is 1001₂ and indeed the 4-bit two's complement representation of -7 according to the table above.

3.1.3 Properties of two's complement representation

In this section, we mention (and prove) some properties of the two's complement representation. The final goal is to prove that this representation indeed allows simple, universal adder hardware, to compute sums of both positive and/or negative numbers.

First, we prove a property that was already mentioned at the end of the previous section, i.e., that the 4-bit two's complement representation of the negative number $-X$ ($X \geq 0$) can be obtained as the 4-bit natural number representation of $2^4 - X$. This is illustrated below for all negative numbers in 4-bit two's complement representation.

Integer	Two's complement	Corresponding natural number
-8	1000 ₂	$8 = 2^4 - 8$
-7	1001 ₂	$9 = 2^4 - 7$
-6	1010 ₂	$10 = 2^4 - 6$
-5	1011 ₂	$11 = 2^4 - 5$
-4	1100 ₂	$12 = 2^4 - 4$
-3	1101 ₂	$13 = 2^4 - 3$
-2	1110 ₂	$14 = 2^4 - 2$
-1	1111 ₂	$15 = 2^4 - 1$

This can be generalized to the following property of two's complement representation.

Property 1: The N -bit two's complement representation of the negative number $-X$ ($X \geq 0$) can be obtained as the N -bit natural number representation of $2^N - X$.

Proof. Let $X_{N-1}X_{N-2}\cdots X_1X_0$ be the N -bit (natural number) representation of the positive number X . We have $2^N - X = (2^N - 1) - X + 1$, where $2^N - 1 = \underbrace{111\cdots 11}_N$, so

$$2^N - X = \underbrace{111\cdots 11}_N - X + 1 = \underbrace{111\cdots 11}_N - X_{N-1}X_{N-2}\cdots X_1X_0 + 1.$$

Computing $\underbrace{111\cdots 11}_N - X_{N-1}X_{N-2}\cdots X_1X_0$ as a subtraction of natural numbers, is equivalent to *flipping* the bits of X . Thus, $2^N - X$ flips the bits of X and then adds 1, which is exactly what the two's complement representation of $-X$ achieves. \square

Example 3.1.1

For $X = 7$, the 4-bit two's complement representation of $-X = -7$ can be obtained as

$$2^4 - X = 1111_2 - X + 1 = 1111_2 - 0111_2 + 1 = 1000_2 + 1 = 1001_2$$

Note: If $X \leq 0$ such that $-X$ is a positive integer, the same result still holds. The two's complement representation of the positive number $-X$ is nothing but the N -bit natural number representation of $-X$. Adding $2^N = \underbrace{1000\cdots 00}_{N+1}$, to obtain $-X + 2^N = 2^N - X$, doesn't affect the representation: the MSB of 2^N cannot be represented with N bits, so its N -bit representation is $\underbrace{000\cdots 000}_N$, which results in adding zero. Therefore, the N -bit natural number representation of $2^N - X$ is the same as the N -bit natural number representation of $-X$, which is the N -bit two's complement representation of $-X$.

We just described, in a mathematical way, how the N -bit two's complement representation of any integer value can be obtained (value \rightarrow representation). Next, we want to study the opposite: how to determine which integer value is represented by a given N -bit pattern, assuming an N -bit two's complement representation is used (representation \rightarrow value).

Remember that, for natural numbers, the value of the number represented by an N -bit pattern $a = a_{N-1}a_{N-2}\cdots a_1a_0$ is given by:

$$\sum_{i=0}^{N-1} a_i 2^i.$$

For example, the natural number represented by the 5-bit pattern 10111_2 is given by $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 2 + 1 = 23$. We want to derive a similar expression to compute the value represented by an N -bit pattern in N -bit two's complement representation:

- For a positive number, e.g., 0101_2 in 4-bit two's complement representation, the corresponding *value* is given by the exact same formula as for natural numbers, applied to all except the MSB:

$$0101_2 \rightarrow 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 1 = 5$$

Note that adding a term $0 \cdot 2^3$ wouldn't change the result, so we could add in the MSB.

- For a negative number, e.g., 1010_2 in 4-bit two's complement representation, let's assume it represents the *value* $-a$, where $a \geq 0$. Property 1 tells us that the 4-bit two's complement representation for $-a$ can be obtained as the 4-bit natural number representation of $2^4 - a$. Since 1010_2 represents the natural number $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, we have that $2^4 - a = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Therefore, we can find $-a$ as

$$\begin{aligned} -a &= -2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \end{aligned}$$

The previous observations generalize to the following property, which we formally prove.

Property 2: The value represented by the N -bit pattern $a_{N-1}a_{N-2} \cdots a_1a_0$, in N -bit two's complement representation, is given by

$$-a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i2^i.$$

Proof. If $a_{N-1} = 0$, a is a positive integer, i.e., natural number, the value of which we can compute as:

$$\sum_{i=0}^{N-1} a_i2^i = \sum_{i=0}^{N-2} a_i2^i = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i2^i,$$

since $a_{N-1} = 0$. If $a_{N-1} = 1$, a is a negative integer and its value is given by $-|a|$ (“minus the absolute value of a ”). For negative a , we have that $|a| = -a$, which is positive, so the value of $(-a)$ is given by

$$\sum_{i=0}^{N-1} (1 - a_i)2^i + 1.$$

Indeed, since we're using two's complement representation, $-a$ can be obtained by complementing the bits of $a_{N-1}a_{N-2} \cdots a_1a_0$ and then adding 1. Therefore, the value of a is given

by

$$\begin{aligned}
-(-a) &= -\left(\sum_{i=0}^{N-1} (1 - a_i)2^i + 1\right) \\
&= -\left(\sum_{i=0}^{N-2} (1 - a_i)2^i + 1\right) \\
&= -\left(\sum_{i=0}^{N-2} 2^i - \sum_{i=0}^{N-2} a_i 2^i + 1\right) \\
&= -\left(\underbrace{\left(\sum_{i=0}^{N-2} 2^i + 1\right)}_{\substack{111 \cdots 11_2 \\ N-2 \text{ bits}}} - \sum_{i=0}^{N-2} a_i 2^i\right) \\
&= -\left(\underbrace{100 \cdots 00}_{N-1 \text{ bits}} - \sum_{i=0}^{N-2} a_i 2^i\right) \\
&= -2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i \\
&= -a_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i,
\end{aligned}$$

since $a_{N-1} = 1$. □

3.1.4 Detecting overflow

Overflow occurs when more *information* is being transmitted than the hardware can represent. For example, a natural number which is larger than $2^{32} - 1$ cannot be represented with standard 32-bit natural number representation. If we want to represent a real number $\pi = 3.141592\dots$, it is impossible to represent it *precisely* using a finite number of bits. In such cases, overflow occurs because the hardware of the computer (32-bit registers) cannot correctly represent the information that is being passed on to it.

When adding numbers, overflow can occur if the sum of 2 numbers falls outside the range of numbers that can be represented with 32 bits. Indeed, even if both numbers fall nicely inside the representable range and can be correctly represented using 32 bits, their sum might fall outside the range of numbers that can be represented with 32 bits, so the sum cannot be correctly represented by the hardware, using 32 bits.

Natural numbers

For natural numbers (i.e., unsigned integers), overflow can be easily detected by checking for a carry-out of the MSB. This is illustrated by the following example.

Example 3.1.2

The 4-bit natural number representations of 8 and 9 are 1000_2 and 1001_2 , respectively. We know that $8 + 9 = 17$, where 17 can be represented as 10001_2 . Note that representing 17, the sum, requires 5 bits. It cannot be represented with 4 bits since it is larger than $2^4 - 1 = 15$, so overflow occurs. When addition hardware computes the sum, by implementing the long hand method, we get the following result:

$$\begin{array}{r} 8_{10} \\ + 9_{10} \\ \hline 17_{10} \end{array} \Rightarrow \begin{array}{r} 1000_2 \\ 1001_2 \\ \hline \dot{1}0001_2 \end{array}$$

The dotted 1 is the carry-out of the MSB. Since it cannot be represented, it is lost and the result, computed by the hardware, becomes 0001_2 , which represents 1, not 17. Since there is no way to represent 17 with 4 bits, overflow occurred and the computed bit sequence for the sum does not represent the correct result.

Two's complement integers

Let's investigate some examples, to see when overflow could occur and, if it does, how it could be detected when adding two's complement integers. When adding two numbers with a different sign using the long hand method, for example,

$$\begin{array}{r} 7_{10} \\ + -5_{10} \\ \hline 2_{10} \end{array} \Rightarrow \begin{array}{r} 0111_2 \\ 1011_2 \\ \hline \dot{1}0010_2 \end{array}$$

or

$$\begin{array}{r} 5_{10} \\ + -7_{10} \\ \hline -2_{10} \end{array} \Rightarrow \begin{array}{r} 0101_2 \\ 1001_2 \\ \hline \dot{1}1110_2 \end{array}$$

overflow cannot occur, since the sum (2, respectively -2) cannot fall outside the range defined by the both operands ($[-5, 7]$, respectively $[-7, 5]$). Assuming both operands can be represented correctly with 4 bits, the sum will and there can be no overflow. Note the dotted 1 though, i.e., the carry-out of the MSB. Since the result is represented with 4 bits, the carry is lost, which is fine as the 4 bits represent the correct result. Therefore, simply detecting a MSB carry-out cannot detect overflow when adding two's complement integers.

Now, let's consider an addition of two positive two's complement numbers, using the long hand method:

$$\begin{array}{r} 5_{10} \\ + 7_{10} \\ \hline 12_{10} \end{array} \Rightarrow \begin{array}{r} 0101_2 \\ 0111_2 \\ \hline 1100_2 \end{array}$$

There is no carry-out of the MSB in this case. However, overflow did occur. Indeed, using a 4-bit two's complement representation, the correct result (the number 12) cannot be represented, as that would require 5 bits (i.e., 01100_2). The computed result, 1100_2 , represents the number -4 and not 12.

Similarly, adding two negative two's complement numbers, for example,

$$\begin{array}{r} -5_{10} \\ + -7_{10} \\ \hline -12_{10} \end{array} \Rightarrow \begin{array}{r} 1011_2 \\ 1001_2 \\ \hline \dot{1}0100_2 \end{array}$$

does create overflow (as well as a carry-out of the MSB, which is lost as only 4 bits are available to represent the result). Indeed, using a 4-bit two's complement representation, the correct result (the number -12) cannot be represented, as that would require 5 bits (i.e., 10100_2). The computed result, 0100_2 , represents the number 4 and not -12.

Overall, we can conclude that, when adding two's complement integers, detecting a carry out of the MSB is not the way to detect overflow. Comparing the examples with overflow with some examples without overflow, when adding numbers with the same sign, for example,

$$\begin{array}{r} 1_{10} \\ + 2_{10} \\ \hline 3_{10} \end{array} \Rightarrow \begin{array}{r} 0001_2 \\ 0010_2 \\ \hline 0011_2 \end{array}$$

and

$$\begin{array}{r} -1_{10} \\ + -2_{10} \\ \hline -3_{10} \end{array} \Rightarrow \begin{array}{r} 1111_2 \\ 1110_2 \\ \hline \dot{1}1101_2 \end{array}$$

we can conclude that overflow occurs when adding two positive numbers results in the representation of a negative number, or vice versa. This can be detected by checking whether the MSB of the result is different from the MSB of the arguments. The following tables summarize our discussion.

$A + B$		
A	B	overflow?
+	-	no
-	+	no
+	+	if $A + B < 0$
-	-	if $A + B > 0$

$A - B$		
A	B	overflow?
+	+	no
-	-	no
+	-	if $A - B < 0$
-	+	if $A - B > 0$

3.1.5 Two's complement representation to build a simple adder

As illustrated earlier, we can add two integers using simple adder hardware (implementing the long hand method for adding natural numbers) when using two's complement representation. So, this representation helps to minimize the hardware complexity. We now prove this result.

Property 3: The two's complement representation of the sum $X + Y$ is correctly obtained by applying the simple bit-wise long hand addition scheme (for natural numbers) to the two's complement representations of the arguments X and Y , if no overflow occurs.

Proof. We distinguish different cases:

- $X \geq 0, Y \geq 0$. Representing X and Y in N -bit two's complement representation results in

$$\begin{aligned} X &= 0 X_{N-2} X_{N-3} \dots X_1 X_0 \\ Y &= 0 Y_{N-2} Y_{N-3} \dots Y_1 Y_0 \end{aligned}$$

Since $X \geq 0$ and $Y \geq 0$, executing a simple bit-wise long hand addition scheme for natural numbers will correctly compute the representation of the sum $X + Y$, assuming there is no overflow. There is no overflow if the MSB of the result is zero, representing a positive sum in two's complement representation. If the MSB of the sum is 1, overflow occurred.

- $X \geq 0, Y < 0$. Let $Y = -Z$ with $Z > 0$. The N -bit two's complement representation of Y is obtained as the N -bit natural number representation of $2^N - Z$ (see Property 2). Since $X \geq 0$, the N -bit two's complement representation of X is simply the N -bit natural number representation of X . Bit-wise long hand addition of the representations of X and Y , i.e., the N -bit natural number representations of X and $2^N - Z$ respectively, results in the N -bit natural number representation of $X + 2^N - Z$. We now consider two scenarios:

1. $X \geq Z$

$$\begin{aligned} X + 2^N - Z &= 2^N + X - Z \\ &= \underbrace{2^N}_{\substack{10 \dots 00_2 \\ N+1 \text{ bits}}} + \underbrace{(X - Z)}_{\text{positive}} \\ &= X - Z \\ &= X + Y \end{aligned}$$

So, the result correctly represents the positive number $X + Y$.

2. $X < Z$

$$\begin{aligned} X + 2^N - Z &= 2^N - Z + X \\ &= 2^N - \underbrace{(Z - X)}_{\text{positive}} \\ &= N\text{-bit two's complement representation of } -(Z - X) \\ &= N\text{-bit two's complement representation of } X - Z \\ &= N\text{-bit two's complement representation of } X + Y \end{aligned}$$

So, the result correctly represents the negative number $X + Y$.

- Other cases, like $X < 0, Y < 0$ or $X < 0, Y \geq 0$ can be proven similarly.

□

This proves that the two's complement representation allows to implement a simple long hand addition scheme, in hardware, to obtain the correct representation of the result, assuming no overflow happens.

3.1.6 Sign extension

To execute the instruction

```
addi $8,$9, constant ,
```

the two arguments, i.e., the content of `$9` and the binary representation of `constant` are fed into the adder hardware, which will compute the binary representation of the result, which then gets transferred into `$8`. Since registers contain 32-bit words, in MIPS R2000, the CPU provides a 32-bit adder, which takes 32-bit inputs and generates a 32-bit output, as illustrated in Figure 3.1. Since the second argument, `constant`, is only represented with 16 bits (since it is part of an I-type instruction), we need to determine the 32-bit representation of `constant` before we can use the adder, i.e., *extend* its 16-bit binary representation to a 32-bit binary representation that represents the same value.

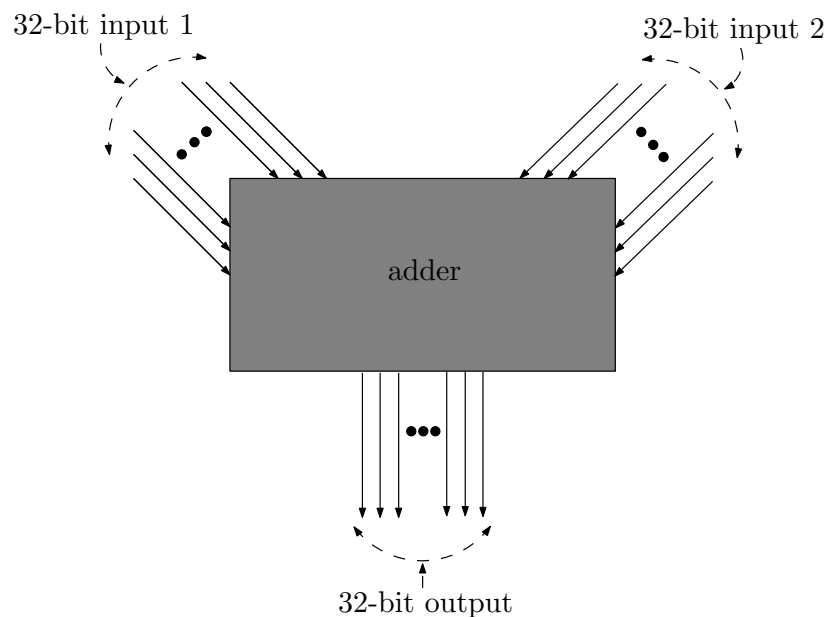


Figure 3.1: Inputs and output of the adder are 32-bit signals.

Let's look at some examples when using two's complement representation.

Example 3.1.3

Representing the integer 5 in 4-bit and 8-bit two's complement representation results in:

$$\begin{aligned} 5_{10} &= 0101_2 \\ 5_{10} &= \underline{0000}0101_2 \end{aligned}$$

So, for a positive integer, 0000 is added to extend a 4-bit two's complement representation to an 8-bit two's complement representation. Representing the negative number -6 in 4-bit and 8-bit two's complement representation results in:

$$\begin{aligned} -6_{10} &= 1010_2 \\ -6_{10} &= \underline{1111}1010_2 \end{aligned}$$

So, for a negative integer, 1111 is added to extend a 4-bit two's complement representation to an 8-bit two's complement representation.

From the example, we can conclude that extending the sign bit (i.e., copying the MSB to the left) is the way to go, to extend an N -bit two's complement representation to an M -bit two's complement representation with $M > N$. This procedure is called **sign-extension**. We now prove this formally.

Property 4: If

$$a = a_{N-1}a_{N-2}\dots a_1a_0$$

is the N -bit two's complement representation of X and

$$b = b_{M-1}b_{M-2}\dots b_{N+1}b_Nb_{N-1}b_{N-2}\dots b_1b_0$$

is the M -bit two's complement representation of X (so, a and b represent the same integer X) with $M > N$, then we have that

$$\begin{aligned} a_{N-1} &= b_{N-1} = b_N = \dots = b_{M-2} = b_{M-1} \\ a_{N-2} &= b_{N-2} \\ a_{N-3} &= b_{N-3} \\ &\dots \\ a_1 &= b_1 \\ a_0 &= b_0, \end{aligned}$$

or, b is obtained by sign-extending a .

Proof. Using Property 2 of two's complement representation, we have:

$$\begin{aligned} \text{value represented by } a &= -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i2^i, \\ \text{value represented by } b &= -b_{M-1}2^{M-1} + \sum_{i=0}^{M-2} b_i2^i. \end{aligned}$$

We now prove that the value represented by b is equal to the value represented by a .

$$\begin{aligned}
\text{value represented by } b &= -\underbrace{b_{M-1}}_{=a_{N-1}} 2^{M-1} + \sum_{i=0}^{M-2} b_i 2^i \\
&= -a_{N-1} 2^{M-1} + \sum_{i=N-1}^{M-2} b_i 2^i + \sum_{i=0}^{N-2} b_i 2^i \\
&= -a_{N-1} 2^{M-1} + \sum_{i=N-1}^{M-2} a_{N-1} 2^i + \sum_{i=0}^{N-2} a_i 2^i \\
&= a_{N-1} \left(-2^{M-1} + \sum_{i=N-1}^{M-2} 2^i \right) + \sum_{i=0}^{N-2} a_i 2^i \\
&= a_{N-1} \left(\underbrace{-2^{M-1} + 2^{M-2}}_{-2^{M-2}} + 2^{M-3} + \dots + 2^N + 2^{N-1} \right) + \sum_{i=0}^{N-2} a_i 2^i \\
&= a_{N-1} \left(\underbrace{-2^{M-2} + 2^{M-3}}_{-2^{M-3}} + \dots + 2^N + 2^{N-1} \right) + \sum_{i=0}^{N-2} a_i 2^i \\
&\quad \dots \\
&= a_{N-1} \left(-2^{N-1} \right) + \sum_{i=0}^{N-2} a_i 2^i \\
&= -a_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i \\
&= \text{value represented by } a.
\end{aligned}$$

□

If an instruction such as `addi $8, $9, constant` is being executed, it will automatically *sign-extend* its third argument, and then compute the sum. Sometimes, however, we really want to treat the 3rd operand as a natural number and not sign-extend it. For example, to load a 32-bit word into the register `$9`:

```
lui $9, constant1
addiu $9, $9, constant2
```

The `lui` instruction loads the 16 most significant bits of `$9` with `constant1` and then loads the 16 least significant bits with `constant2` using an `addiu` instruction, which stands for “add immediate unsigned”. The latter instructs the CPU to not sign-extend the third operand, `constant2`. Using an `addi` instruction instead of `addiu` would extend the sign bit (the MSB) of `constant2`. If the MSB of `constant2` happens to be 1, the sign-extension followed by addition would alter the 16 most significant bits written into `$9` by the `lui` instruction. Similar to `addiu`, there are instructions like `lbu` (“load byte unsigned”), etc.

3.1.7 Handling overflow

If overflow is detected, an **exception (interrupt)** is caused. Exceptions (interrupts) are unscheduled procedure calls and are handled by the OS. The program jumps to the **exception handler** and has access to a special exception program counter register **\$EPC** and two regular, OS reserved registers **\$k0** and **\$k1** to deal with exceptions. When an instruction causes an exception, the CPU will do the following:

1. **\$EPC** \leftarrow address of instruction that is causing the exception
2. **\$PC** \leftarrow address of the exception handler

Using **\$EPC**, the CPU can return to the point where it was after handling the exception and continue the execution of the program (since exceptions are unscheduled procedure calls, they cannot make use of **\$RA**, as the unexpected overwriting of **\$RA** could cause problems at a later point). Returning to the exception causing instruction is done as follows:

```
mfc0 $k0, $EPC # move from system control
jr $k0
```

Exception handling is similar to the combination of

```
jal ...
jr $RA
```

However, in the case of an exception handler call, the state of the CPU should be exactly restored, upon return from the exception handler, to what it was before the exception. None of the registers can be overwritten (including the return register **\$RA**), except the registers **\$k0** and **\$k1**, which are reserved for the OS and to be used, e.g., when handling exceptions. So, **\$k0** and **\$k1** are not restored on exceptions. Therefore, compilers will never use them. Exceptions are implemented for overflow in two's complement arithmetic: overflow from **add**, **addi**, **sub** will call the exception handler. For arithmetic instructions using natural numbers (**addu**, **addiu**, **subu**), overflow will not cause an exception. The programmer should implement it, where needed.

3.2 Adder design

In the previous section, we have shown that the two's complement representation allows to implement a simple, universal adder for integers. In this section, we actually design this 32-bit adder, for a MIPS R2000 CPU.

3.2.1 1-bit adder

Since the long hand scheme for 32-bit addition involves adding single bits, we first focus on the design of a 1-bit adder. A 1-bit adder takes two 1-bit inputs, a and b and outputs the 1-bit number s that represents the sum $a + b$ of the two inputs, as well as the 1-bit carry that this addition might generate, c_o (the “carry-out”). The truth table of this 1-bit adder is given by:

a	b	s	c_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The logic design in Figure 3.2 implements this truth table. It is called a **half adder** for reasons explained a little more below.

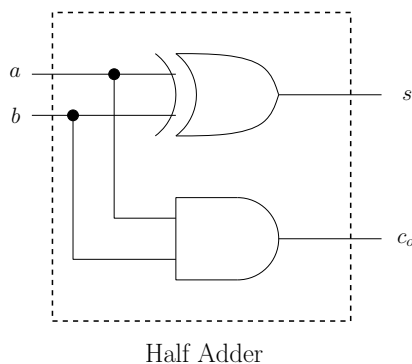


Figure 3.2: Design of 1-bit *half* adder.

If we ignore the internal structure of the 1-bit half adder and simply concentrate on the inputs and outputs, the 1-bit half adder can be represented as in Figure 3.3.



Figure 3.3: 1-bit half adder block.

To build a 2-bit adder, one could concatenate two 1-bit adders. However, the second adder could not account for the *carry-out* of the first adder, since the second 1-bit half adder doesn't have a *carry-in* input. So, to build multi-bit adders, we need 1-bit adders that have both a *carry-out* output as well as a *carry-in* input. Therefore, the 1-bit adder we implemented above is called a **half adder**, while a 1-bit adder that also takes a *carry-in* input is called a **full adder**. To design a 1-bit full adder, we use the K-maps below, which show the sum s and the carry-out c_o given the inputs a and b and the carry-in c_i .

K-map for s

$c_i \backslash a b$	0 0	0 1	1 1	1 0
0	0	1	0	1
1	1	0	1	0

K-map for c_o

$c_i \backslash a b$	0 0	0 1	1 1	1 0
0	0	0	1	0
1	0	1	1	1

The logic design in Figure 3.4 implements these K-maps.

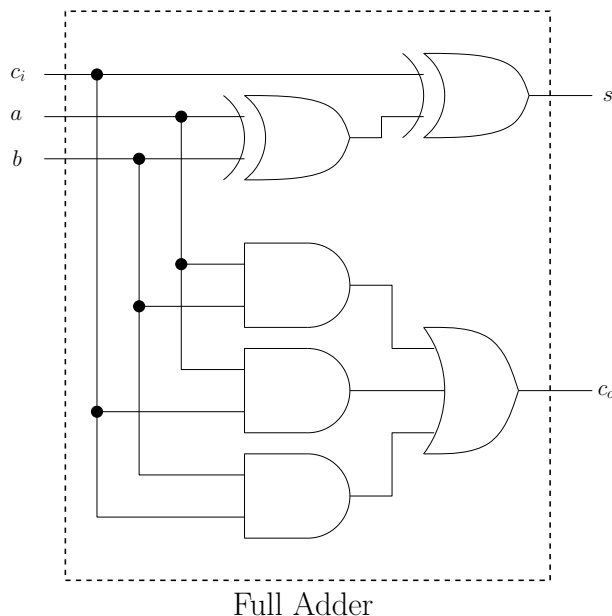


Figure 3.4: Design of 1-bit *full* adder.

Ignoring the internal structure of the 1-bit full adder, it can be represented as the block in Figure 3.5.



Figure 3.5: 1-bit full adder block.

3.2.2 32-bit ripple-carry adder

Now that we designed the 1-bit half adder and 1-bit full adder, we can use them as *building blocks*, to design a 32-bit adder. 32 bits is the appropriate width for an adder in a MIPS R2000 processor, since registers, which contain the arguments of the `add` instruction, are 32 bits wide. Note that the long hand scheme for 32-bit addition consists of a repetition of

1-bit additions, taking into account carry bits. So, the most primitive 32-bit adder can be implemented by simply concatenating 32 1-bit adders, as depicted in Figure 3.6.

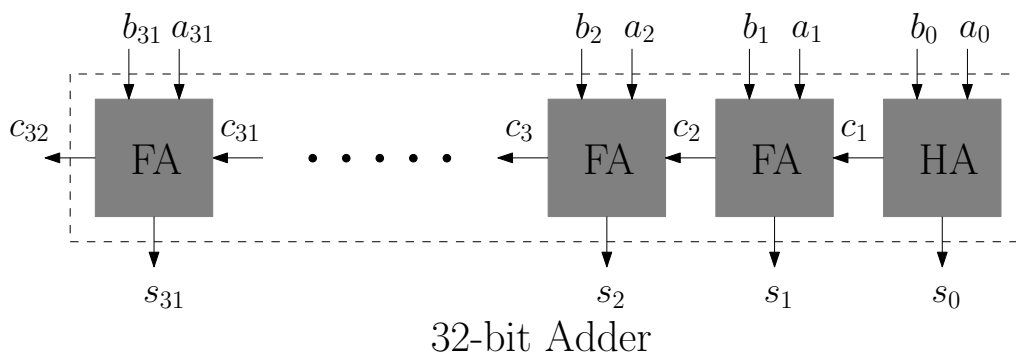


Figure 3.6: One possible design of a 32-bit adder: the **32-bit ripple-carry adder**. In this figure, $a_i, i = 0, \dots, 31$ and $b_i, i = 0, \dots, 31$ are the bits of the arguments a and b , while $s_i, i = 0, \dots, 31$ are the bits of the sum s and $c_i, i = 1, \dots, 32$ are the carry bits.

Note that the rightmost 1-bit adder in Figure 3.6 is a half adder, not a full adder, since there is no carry-in to compute the rightmost, i.e., first bit of the result. We will analyze the performance of this 32-bit ripple-carry adder and optimize the design to enhance its performance in the following sections, which will also explain why it is called a 32-bit **ripple carry** adder.

Implementing subtraction

To compute $a - b$, we observe that $a - b = a + (-b)$ and that, in two's complement representation, $-b$ can be obtained as the two's complement of b , i.e., by inverting every bit of b and adding 1. This allows to implement **subtraction** with the simple variation of the 32-bit adder in Figure 3.7

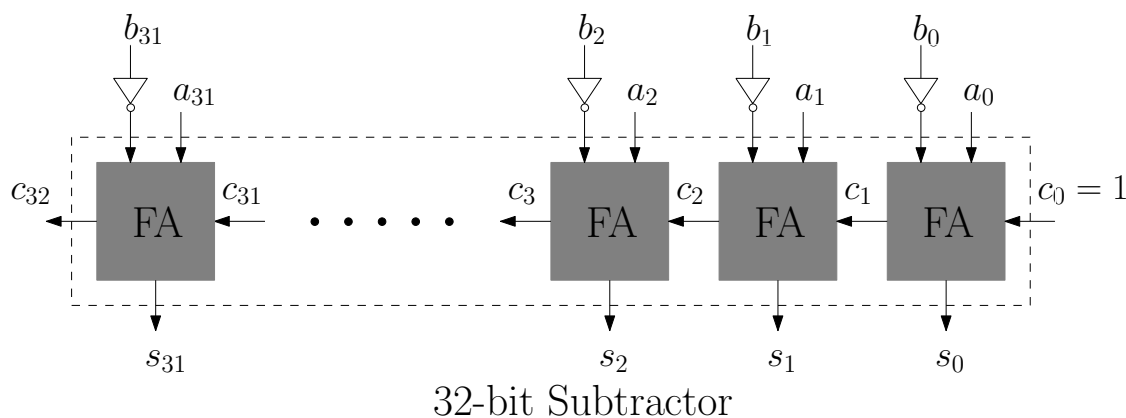


Figure 3.7: Using a 32-bit adder for subtraction.

The 32 invertors complement all the bits of the second argument b , while adding 1 can be achieved by changing the rightmost half adder into a full adder and providing 1 as its carry-in input. So, besides allowing simple adder design, two's complement representation also allows a simple implementation for subtraction of integers!

3.2.3 ALU design

In a next step, we want to upgrade our 32-bit adder to a component that performs addition, subtraction, the logical AND operation, and the logical OR operation. Such component is called an **ALU** (Arithmetic and Logic Unit) and is a core component of the CPU (as we will see later, it will be used for the execution of many MIPS R2000 instructions). As before, for the 32-bit adder, we will build a 32-bit ALU by concatenating 32 1-bit ALU components, which we will design first.

1-bit ALU

The 1-bit ALU computes one of several arithmetic or logic operations. Which operation it computes is selected by op bits: 00 to compute AND, 01 for OR and 10 to compute a SUM. In MIPS R2000, these op bits will be obtained by analyzing the opcode (and the FUNCT field for an R-type instruction). We can implement this component by realizing all operations in parallel (AND, OR, SUM), and then select the appropriate operation using a mux, with the op bits as input. This is shown in Figure 3.8.

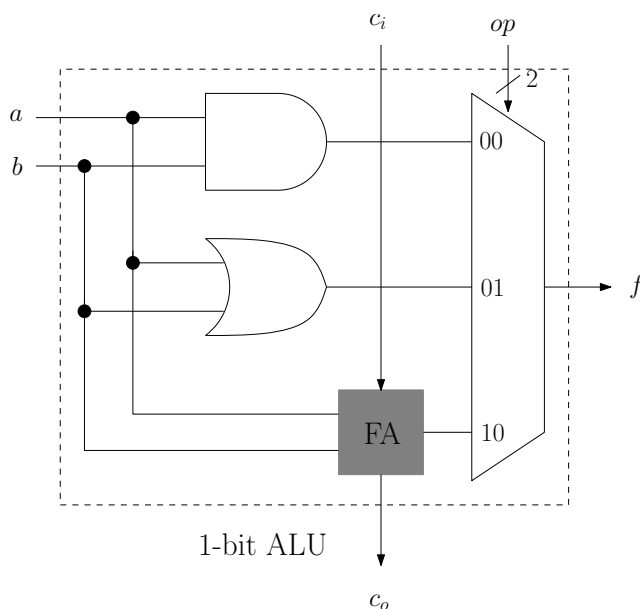


Figure 3.8: Design of 1-bit ALU (+, AND, OR).

To allow subtraction, we use another control bit (in MIPS R2000, this bit will be extracted from the FUNCT field of the instruction, as addition and subtraction share the same opcode, but have a different FUNCT field), which will determine whether the full adder block will compute an addition or a subtraction. This can be implemented as in Figure 3.9.

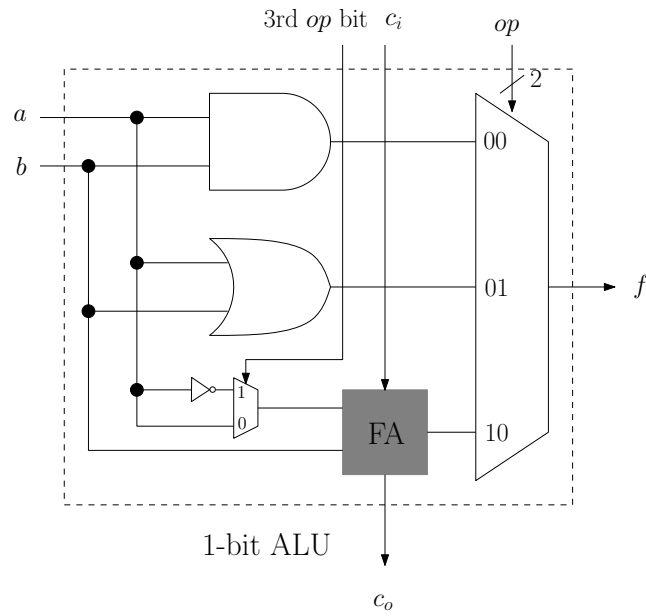


Figure 3.9: Design of 1-bit ALU (+, -, AND, OR).

The 1-bit ALU block can be represented as in Figure 3.10.

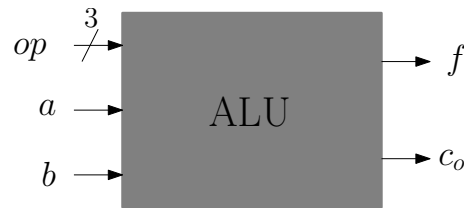


Figure 3.10: 1-bit ALU block.

32-bit ALU

Designing a 32-bit ALU is now straightforward. Just as for a 32-bit adder, it is done by *concatenating* 32 1-bit ALU blocks as depicted in Figure 3.11.

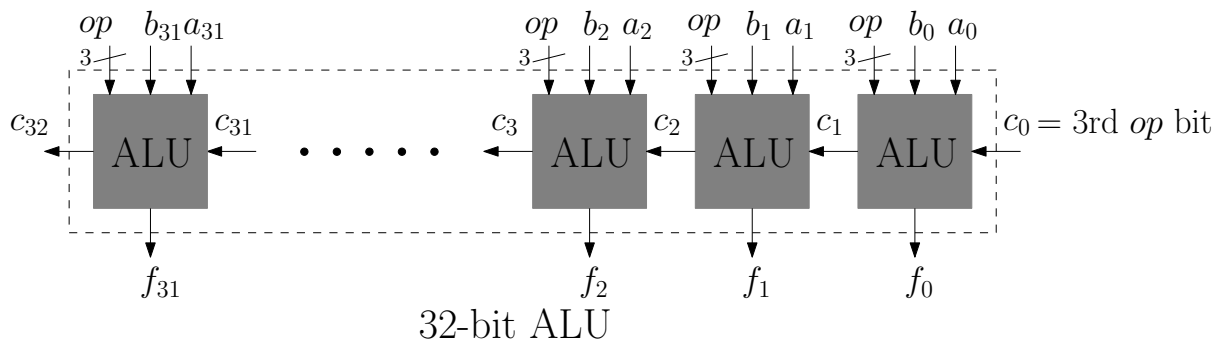


Figure 3.11: Design of 32-bit ALU.

The carry-in c_0 is connected to the 3rd *op* bit: it will be 1 to compute $a - b$ and 0 to compute $a + b$.

3.2.4 Carry lookahead adder

The adder we designed in Figure 3.6 is called a **ripple-carry adder**. The following example illustrates where the name comes from.

Example 3.2.1

Let's assume we want to compute

$$\underbrace{000 \cdots 001}_{32 \text{ bits}} + \underbrace{011 \cdots 111}_{32 \text{ bits}}$$

using the 32-bit adder from Figure 3.6, which is revisited below.

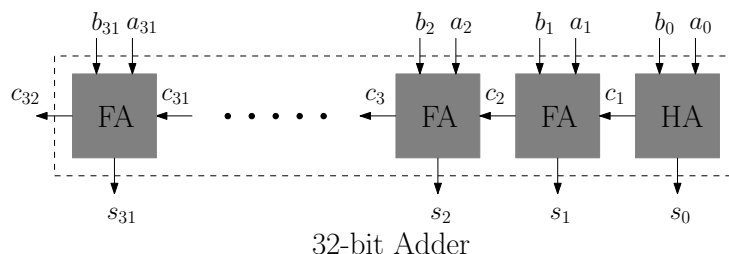


Figure 3.12: The 32-bit *ripple-carry adder* revisited.

In other words, $a_0 = 1, a_1 = a_2 = \dots = a_{31} = 0$ and $b_0 = b_1 = \dots = b_{30} = 1, b_{31} = 0$. Let's examine bit by bit, what happens in hardware, to compute the sum.

$$\begin{aligned} s_0 &= a_0 + b_0 = 0 \Rightarrow c_1 = 1 \\ s_1 &= a_1 + b_1 + c_1 = 0 \Rightarrow c_2 = 1 \\ s_2 &= a_2 + b_2 + c_2 = 0 \Rightarrow c_3 = 1 \\ &\dots \\ s_{30} &= a_{30} + b_{30} + c_{30} = 0 \Rightarrow c_{31} = 1 \\ s_{31} &= a_{31} + b_{31} + c_{31} = 1 \end{aligned}$$

Note that the i -th calculation (s_i) depends on the result of the $(i - 1)$ -th calculation since one of its inputs, i.e., c_i is computed by the $(i - 1)$ -th calculation. With the current adder design, the computation of s_{31} requires waiting for the previous 1-bit adder to compute c_{31} correctly, which requires, in turn, that that previous 1-bit adder waits for c_{30} , etc. So, to compute s_{31} , the leftmost 1-bit adder needs to wait for all other (31) 1-bit adders to compute their carry bits c_i sequentially, i.e., one after the other. This particular example leads to a carry $c_i = 1, i = 1, \dots, 31$ *rippling* all the way from the rightmost adder (c_1) to the leftmost adder (c_{31}). Therefore the name *ripple-carry adder*.

From the previous example, it is clear that the worst-case delay (to compute a stable result) of a ripple-carry adder is very long, due to the serial calculation of all carry bits. To make the implementation of the 32-bit adder more efficient (i.e., decrease its worst-case delay), one possibility is to investigate whether the higher order carry bits, i.e., c_2, c_3, \dots, c_{31} can be computed in parallel, rather than serial (so, without having to wait for a series of 1-bit adders computing their results sequentially). Since any logic function can be computed in two levels of logic, it is possible to compute c_2, c_3, \dots, c_{31} directly from the inputs a, b and c_0 , using only two levels of logic (instead of a sequence of full adders). Indeed, defining $G_i = a_i b_i$ and $P_i = a_i + b_i$ as shorthand notations, the carry bits can be computed as

$$\begin{aligned}
 c_1 &= a_0 b_0 + a_0 c_0 + b_0 c_0 \\
 &= a_0 b_0 + (a_0 + b_0) c_0 \\
 &= G_0 + P_0 c_0 \\
 c_2 &= G_1 + P_1 c_1 \\
 &= G_1 + P_1 (G_0 + P_0 c_0) \\
 &= G_1 + P_1 G_0 + P_1 P_0 c_0 \\
 c_3 &= G_2 + P_2 c_2 \\
 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\
 c_4 &= G_3 + P_3 c_3 \\
 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \\
 &\dots \\
 c_{31} &= ?
 \end{aligned}$$

Although this clearly shows that two levels of logic are sufficient to realize c_2, c_3, \dots, c_{31} from the inputs a, b and c_0 , it also shows that the equations expand rapidly when we get to the higher order bits. So, computing 32 carries efficiently, in parallel, will increase the complexity and therefore the cost of the hardware significantly. Therefore, most schemes to improve efficiency will try to limit the complexity of the equations (to keep the hardware simple), while still providing a substantial speed-up compared to a ripple-carry adder.

4-bit CLA

To keep the complexity of the equations low, let's limit ourselves to implementing a 4-bit carry lookahead adder, by computing c_1, c_2, c_3 and c_4 in parallel. This can be done using the previous logic equations, implemented by a logic block called "Carry Lookahead". Starting from a 4-bit ripple-carry adder, a **4-bit CLA** (Carry Lookahead Adder) can be implemented by removing the sequential computation of carry bits and replacing it by a parallel computation using "Carry Lookahead" logic, as illustrated in Figure 3.13.

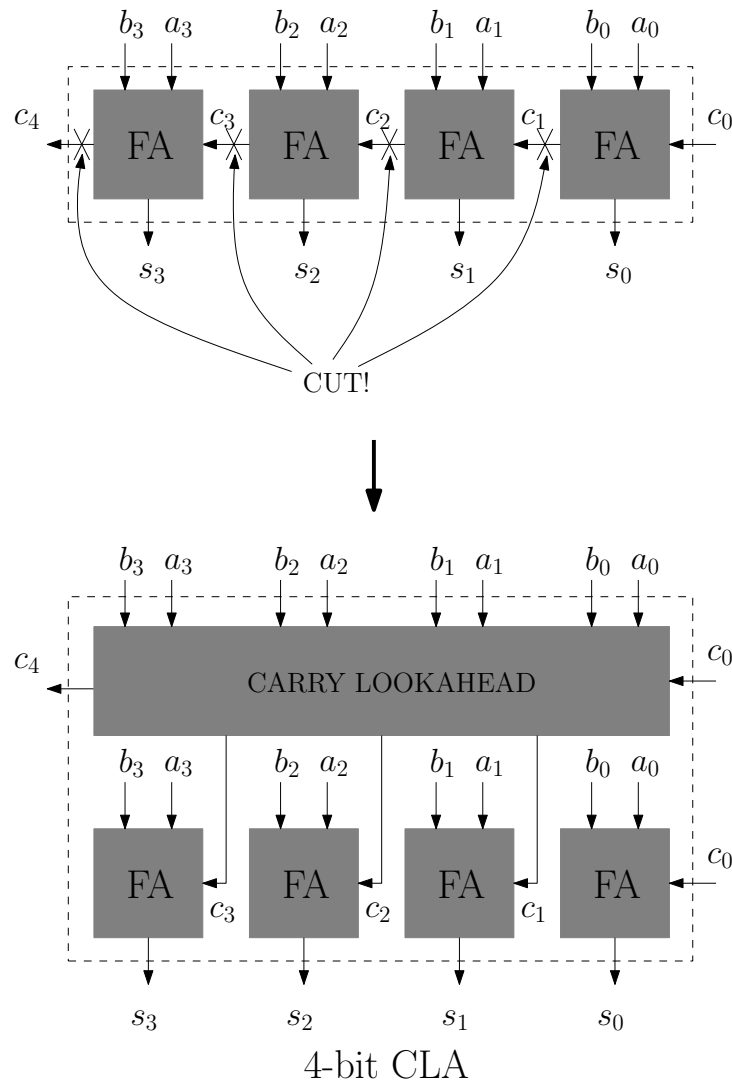


Figure 3.13: 4-bit CLA.

In the 4-bit CLA, each FA block takes its carry-in from the carry lookahead block *without delay* (instead of waiting for the carry-out of the previous FA block). This improves performance for a 4-bit adder.

16-bit CLA

A 16-bit adder can be implemented by concatenating four 4-bit CLAs, as shown in Figure 3.14.

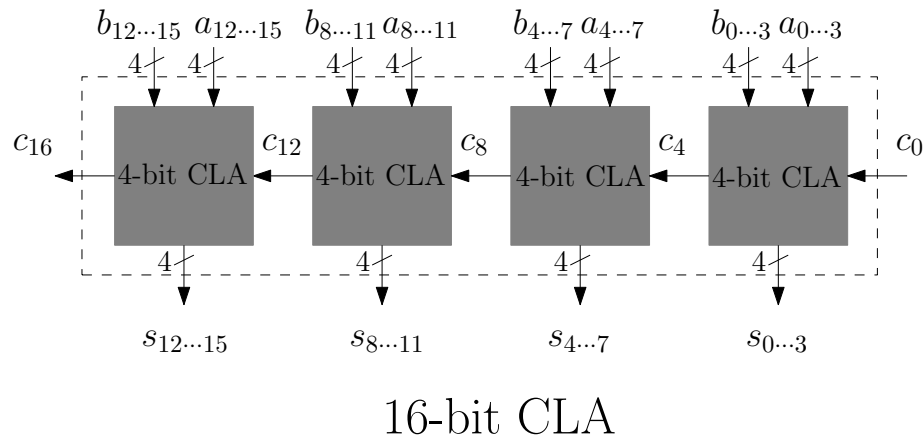


Figure 3.14: Design of 16-bit CLA, with ripple between 4-bit stages.

Simply concatenating the 4-bit CLAs will create a ripple effect again, now between the 4-bit CLA stages. Although each individual 4-bit CLA stage will be more efficient than a 4-bit ripple-carry adder, efficiency will still suffer from the ripple effect between the 4-bit CLA stages. One alternative to improve efficiency is to build a 16-bit CLA computing all 16 carry bits in parallel, just like we built the 4-bit CLA. However, that would increase the hardware complexity and cost significantly. Another alternative is to realize that the scheme in Figure 3.14 is amenable to a modification similar to the one presented in Figure 3.13: remove the sequential computation of carry bits c_4, c_8, c_{12} and c_{16} and replace it by a parallel computation using “Second Level Carry Lookahead” logic, as illustrated in Figure 3.15.

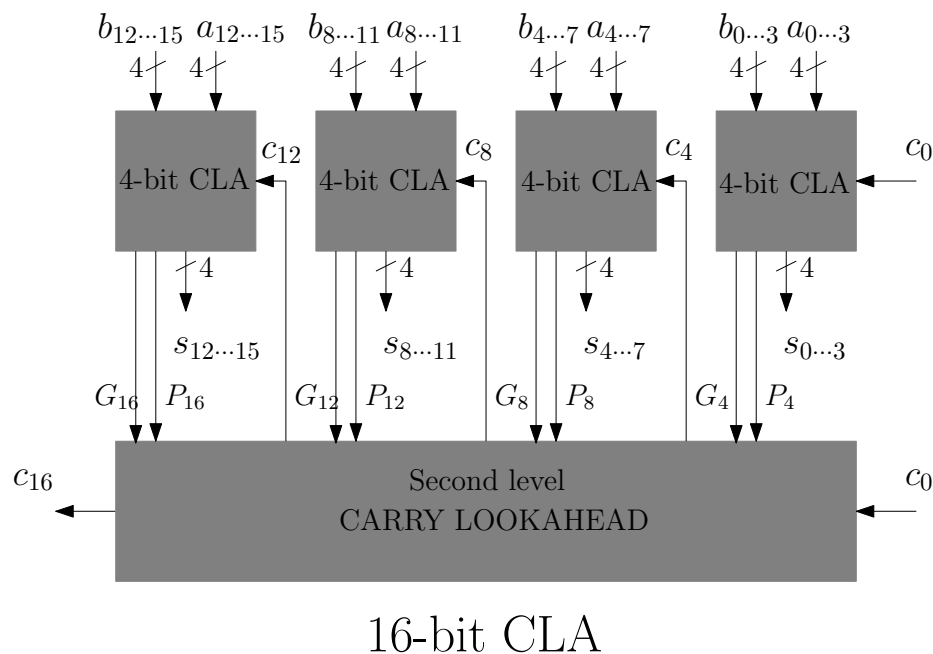


Figure 3.15: Design of 16-bit CLA, with second level carry lookahead.

To make sure this design is indeed possible, we need to show that c_4, c_8, c_{12} and c_{16} can be computed by the proposed scheme. Letting $P_4 = P_3P_2P_1P_0$ and $G_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$, we have

$$c_4 = G_4 + P_4c_0.$$

Note that G_4 and P_4 are independent of c_0 , so they can get computed directly by the rightmost 4-bit CLA. Similarly,

$$c_8 = G_8 + P_8c_4,$$

where we know that $c_4 = G_4 + P_4c_0$, so

$$c_8 = G_8 + P_8G_4 + P_8P_4c_0,$$

where G_8 and P_8 can get computed directly by the second rightmost 4-bit CLA. Similarly, c_{12} and c_{16} can be computed by the ‘‘Second Level Carry Lookahead Logic’’. The equations don’t get too complex, using this hierarchical scheme of carry lookahead logic, which makes the adder significantly more efficient, while preventing a hardware explosion.

By adding a third level of carry lookahead logic, a 64-bit CLA could be implemented. In general, this hierarchical scheme of carry lookahead logic can speed up an N -bit adder from $O(N)$ (ripple-carry scheme) to $O(\log(N))$. For example, a 4-bit CLA requires one layer of carry lookahead logic, which can be implemented using 2 levels of logic, where $O(\log(4)) = O(2)$. A 16-bit CLA requires two layers of carry lookahead logic, i.e., two times two or four levels of logic: $O(\log(16)) = O(4)$. For a 64-bit CLA, three layers of carry lookahead logic or six levels of logic would be required: $O(\log(64)) = O(6)$.

3.3 Multiplication

So far, we have studied addition/subtraction of two 32-bit numbers and the corresponding design of a 32-bit adder. In this section, we will study multiplication of two numbers and the corresponding design of hardware for multiplication. Let’s start with an example.

Example 3.3.1

To multiply the natural numbers 6 and 7 using a 4-bit representation, we apply the following long hand scheme to 0110_2 and 0111_2 :

	0 1 1 0		Multiplicand
×	0 1 1 1		Multiplier
	0 1 1 0		$0110_2 \times 1$
	0 1 1 0		0110 ₂ × 1 and shift ←
0	1 1 0		0110 ₂ × 1 and shift ←←
0	0 0 0		0110 ₂ × 0 and shift ←←←
	0 1 0 1 0 1 0		Product

The first operand, 0110_2 , is called the *multiplicand*, while the second operand, 0111_2 , is called the *multiplier*. One can verify that the result, 0101010_2 represents the correct product, i.e., 42.

In general, a long hand scheme for multiplication using binary representations boils down to *adding shifted versions of the multiplicand*. In the above example, the simple addition of shifted versions of 0110_2 computes the correct product. So, a multiplication can be implemented by

1. Initialize product to zero and i to 1.
2. If the i th bit of the multiplier is 1, shift the multiplicand $i - 1$ times to the left and add the shifted version of the multiplicand to the product.
3. If the i th bit of the multiplier is 0, do nothing.
4. Increment i and go back to 2, until all bits of the multiplier have been processed.

Note that multiplying an n -bit multiplicand with an m -bit multiplier will require $n + m$ bits for the product, to ensure all possible products can be represented. Therefore, multiplication hardware in MIPS R2000 will provide 64 bits for the product. If we eventually wish to represent the product with 32 bits, we will have to check for overflow.

The previous multiplication procedure is implemented in Figure 3.16.

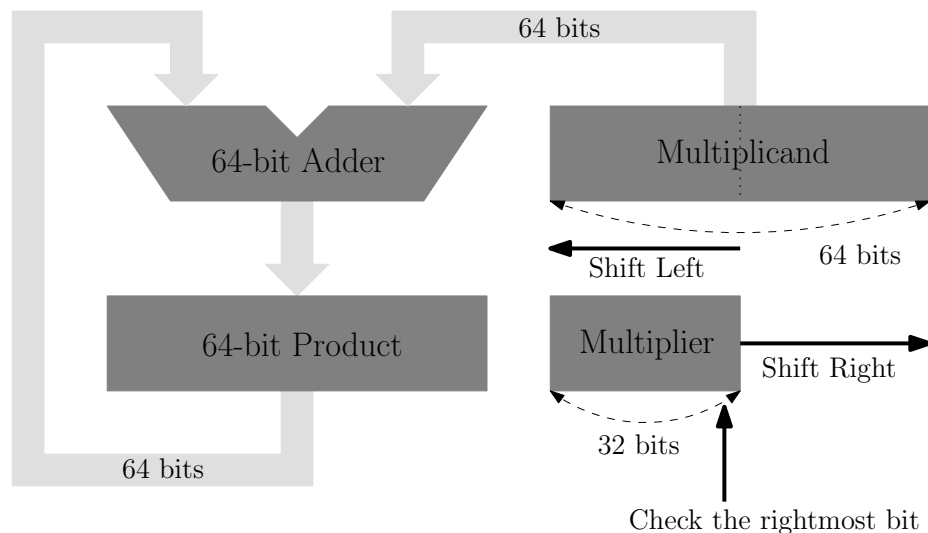


Figure 3.16: Hardware design for 32-bit multiplication. The rectangular boxes are registers: 64 bits for product and multiplicand and 32 bits for the multiplier. At each step of the computation, the multiplicand is shifted one bit to the left, so a shifted version is available for addition with the 64-bit adder. The right half (32 least significant bits) of the multiplicand register is initialized with the 32-bit multiplicand, while the left half (32 most significant bits) is initialized with zeros. The multiplier is shifted to the right, so checking the rightmost (least significant) bit of the multiplier register at each step suffices to decide whether to add the shifted multiplicand to the product or do nothing.

Let's illustrate how this multiplication hardware computes the result for the previous example, i.e., for the multiplication of two 4-bit operands with an 8-bit result. Figure 3.17 shows the initialization.

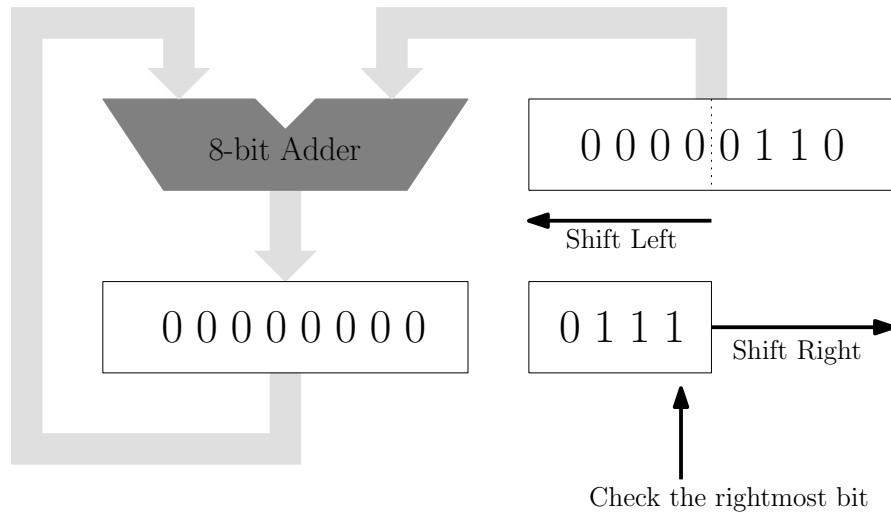


Figure 3.17: Initialization.

Checking the rightmost bit of the multiplier (0111_2) returns 1, so the 0-shifted multiplicand needs to be added to the product at this step. The intermediate result is depicted in Figure 3.18

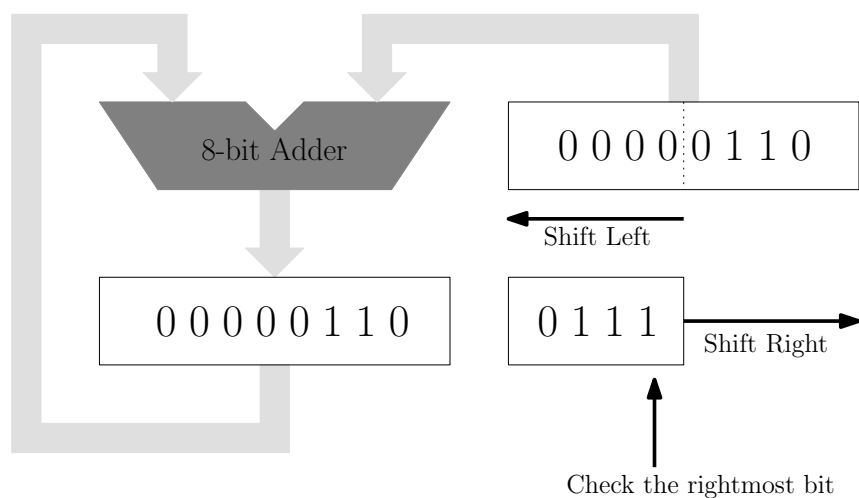


Figure 3.18: After the 1st addition.

After adding the 0-shifted multiplicand, the second bit of the multiplier needs to be inspected to decide whether the single-left-shifted multiplicand is to be added to the (inter-

mediary) product. Therefore, the multiplicand register is shifted 1 bit to the left and the multiplier register is shifted 1 bit to the right, resulting in Figure 3.19.

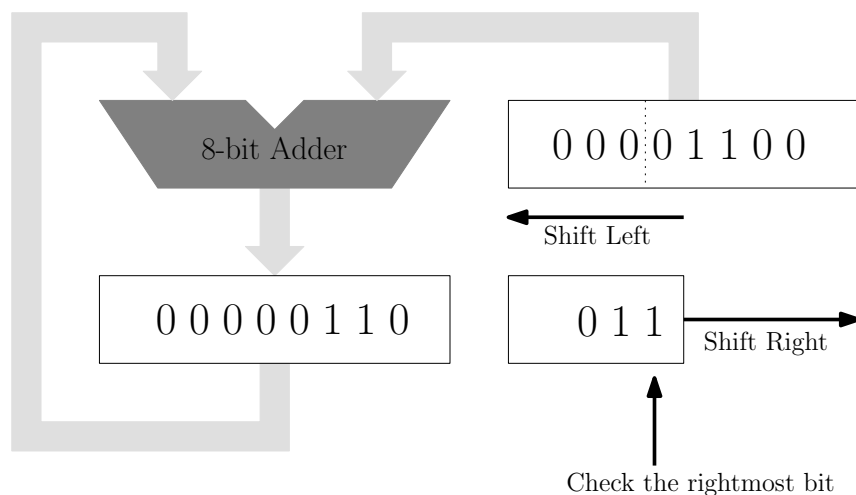


Figure 3.19: After shifting the multiplicand and multiplier register.

Checking the rightmost bit in the multiplier register returns 1, so the single-left-shifted multiplicand is added to the (intermediary) product, in Figure 3.20. This addition corresponds to the second line in the long hand scheme for this example.

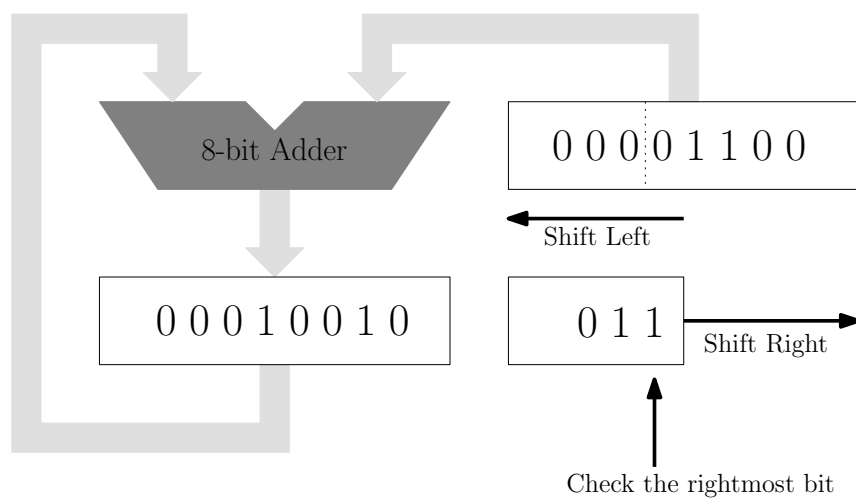


Figure 3.20: After the 2nd addition.

Repeating similar steps results in Figures 3.21 through 3.25.

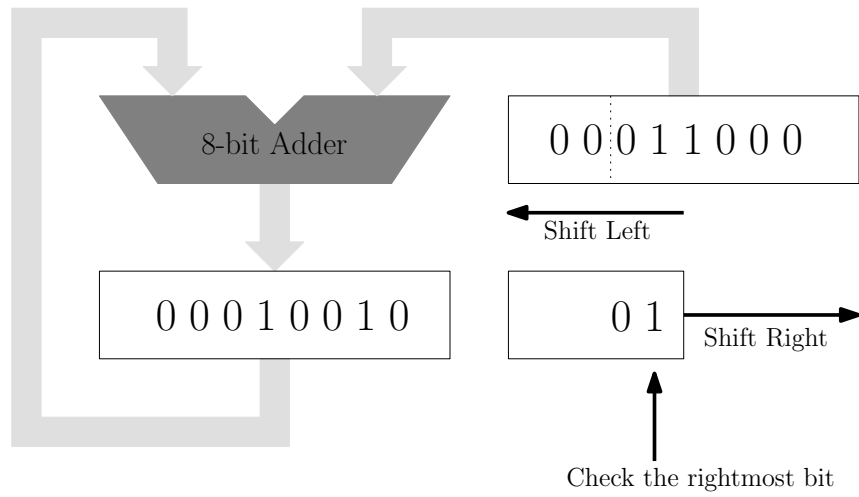


Figure 3.21: After shifting the multiplicand and multiplier register.

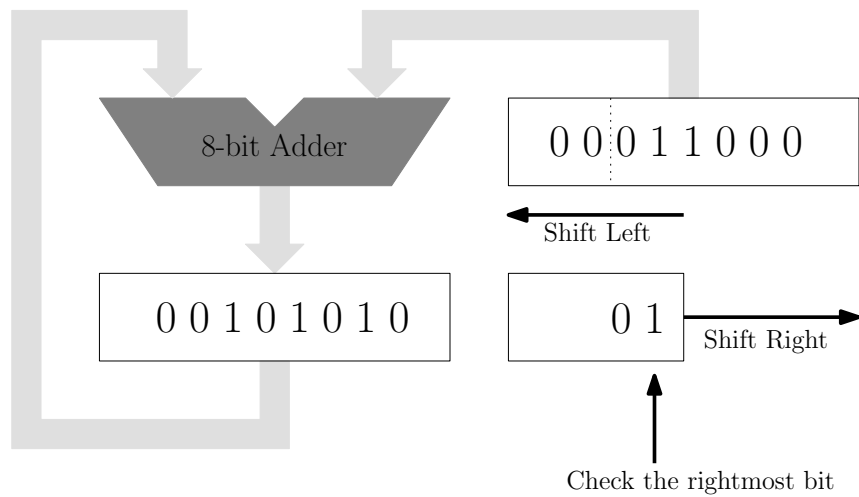


Figure 3.22: After the 3rd addition.

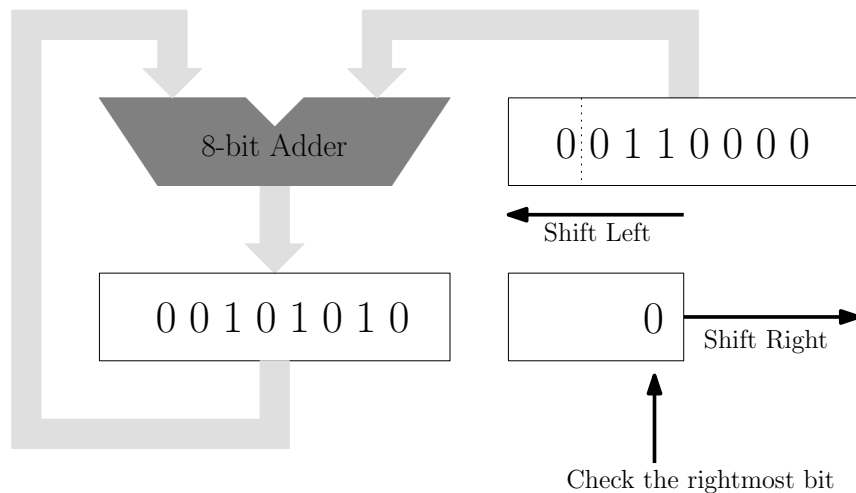


Figure 3.23: After shifting the multiplicand and multiplier register.

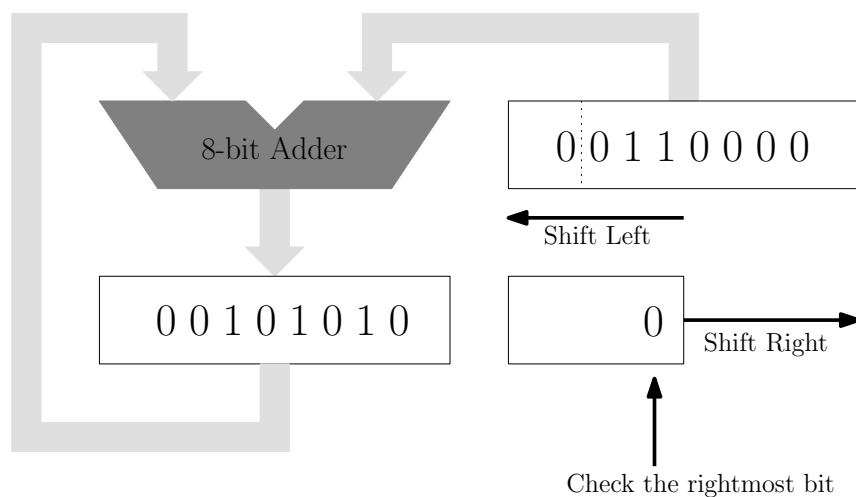


Figure 3.24: No 4th addition.

Notice how checking the rightmost bit in the multiplier register returns 0, in Figure 3.23, so no addition occurs for the triple-left-shifted multiplicand. This corresponds to the fourth line in the long hand scheme for this example.

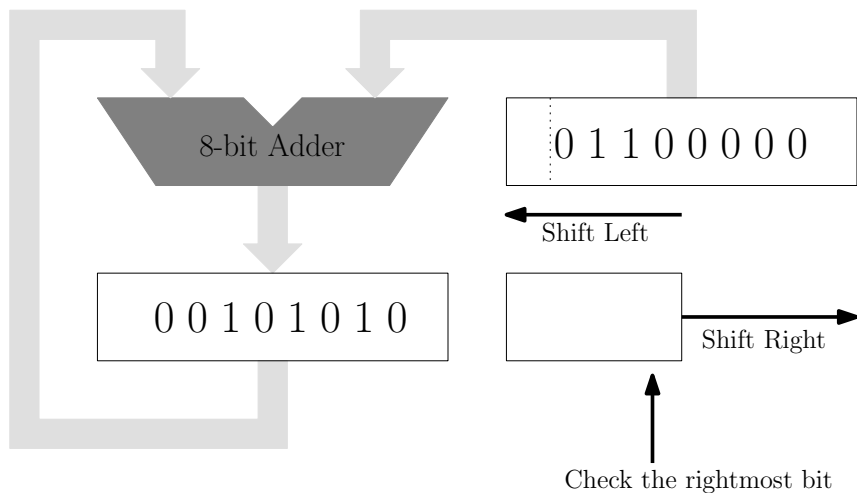


Figure 3.25: After shifting the multiplicand and multiplier register one last time.

Although the previous implementation works correctly (as illustrated by the example), it requires two 64-bit registers and a 64-bit adder (or, for the example, two 8-bit registers and one 8-bit adder). A first reduction in hardware complexity can be accomplished by observing that shifting the product register to the right achieves the same result as shifting the multiplicand register to the left. This leads to the design in Figure 3.26.

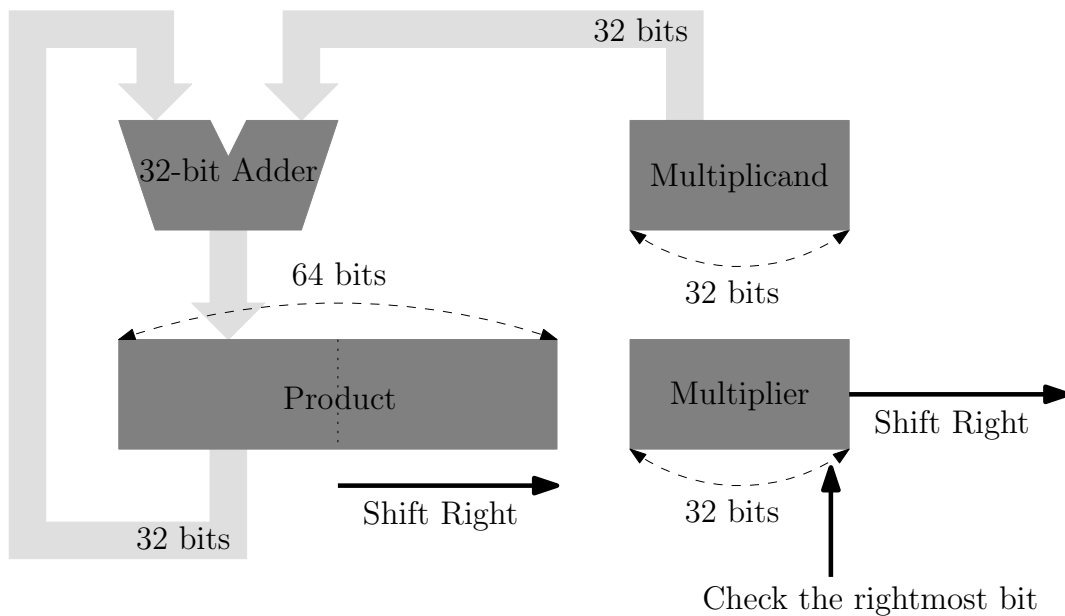


Figure 3.26: Optimized multiplication hardware with a 32-bit adder, a 32-bit multiplicand register and 32-bit connections between the different components.

The product register is initialized with $000\cdots 00_2$ and will be shifted from left to right as more and more steps are computed. After each 32-bit addition, the sum will be stored in the leftmost (most significant) 32 bits of the product register.

To obtain a second reduction in hardware complexity, note that, initially, the rightmost (least significant) 32 bits of the product register can be chosen randomly without affecting the computation. This is exactly how many bits are needed to store the multiplier, initially. After one addition and one right shift, the leftmost 33 bits of the product register store an important intermediary result and the rightmost (least significant) 31 bits of the product register can still be chosen randomly, without affecting the computation. Only 31 bits of the multiplier remain to be stored at that point. And so on. Therefore, storage for the product and the multiplier could be combined by using the rightmost, unused bits of the product register to store what remains to be stored of the multiplier. This allows to eliminate the multiplier register, which simplifies the hardware, as depicted in Figure 3.27.

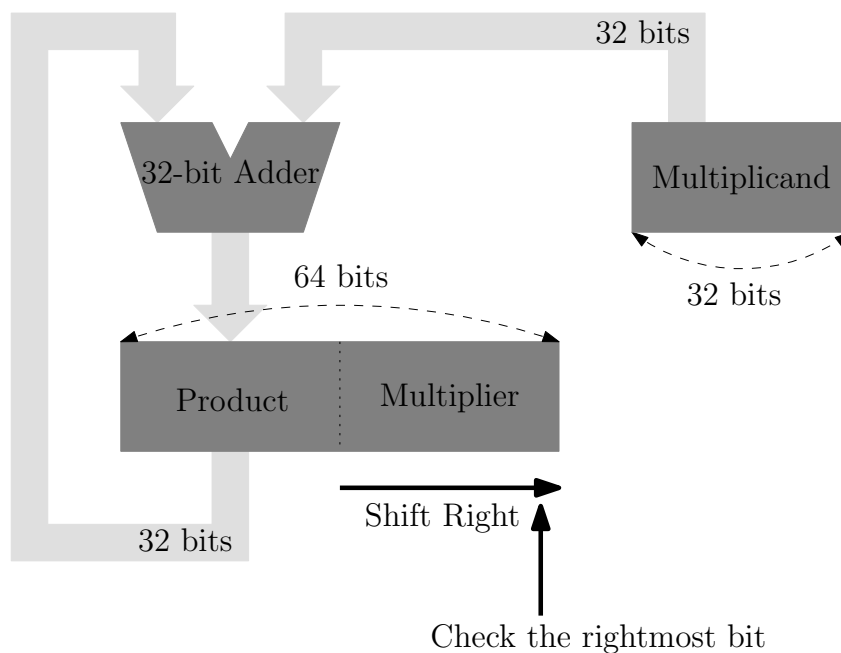


Figure 3.27: Optimized multiplication hardware, without multiplier register.

Figures 3.28 to 3.36 illustrate the functionality of the optimized hardware with the example we previously discussed (multiplying the 4-bit arguments 0110_2 and 0111_2 to obtain an 8-bit product).

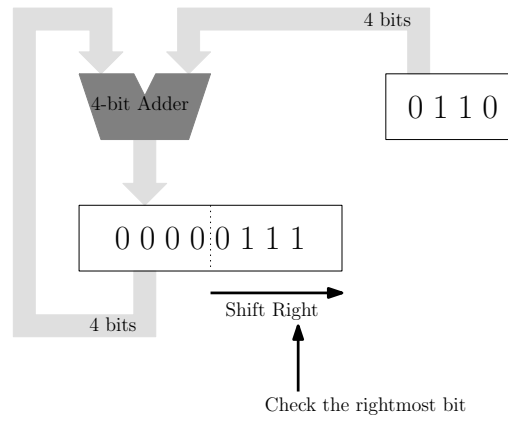


Figure 3.28: Initialization.

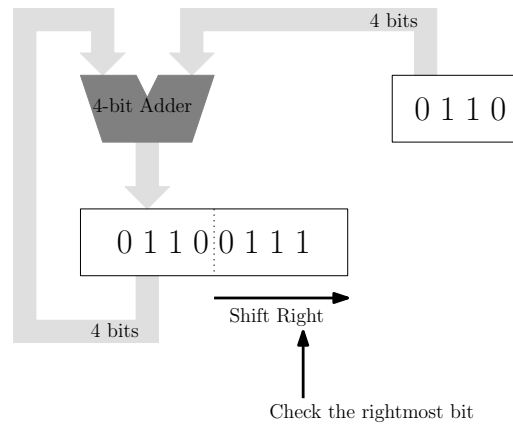


Figure 3.29: After the 1st addition.

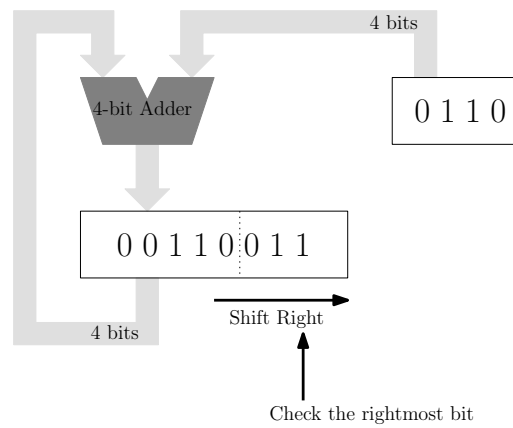


Figure 3.30: After shifting the product register.

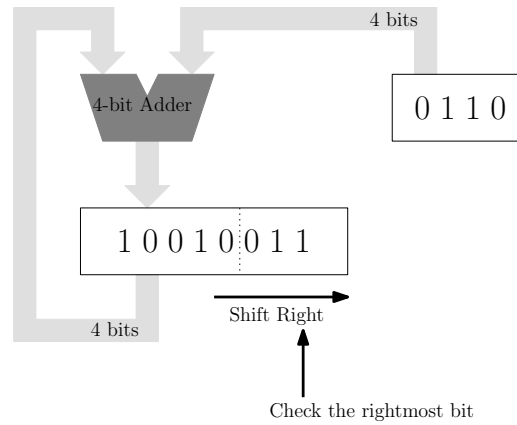


Figure 3.31: After the 2nd addition.

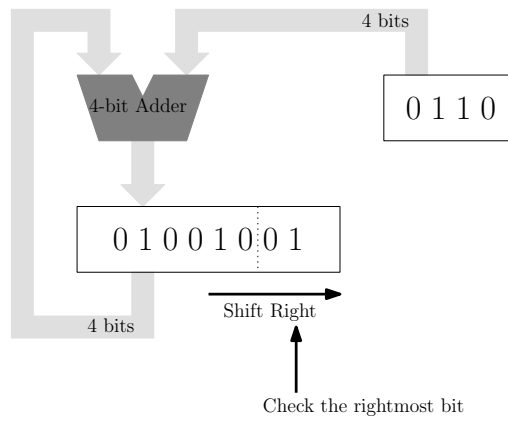


Figure 3.32: After shifting the product register.

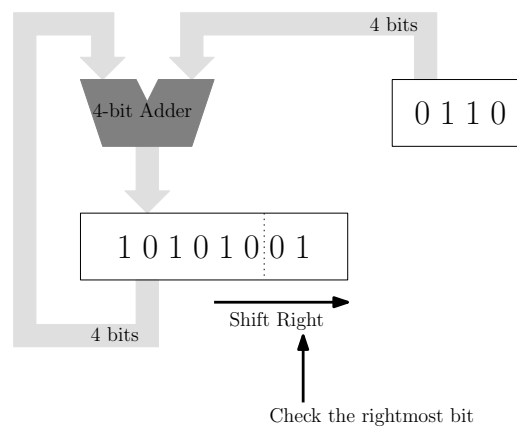


Figure 3.33: After the 3rd addition.

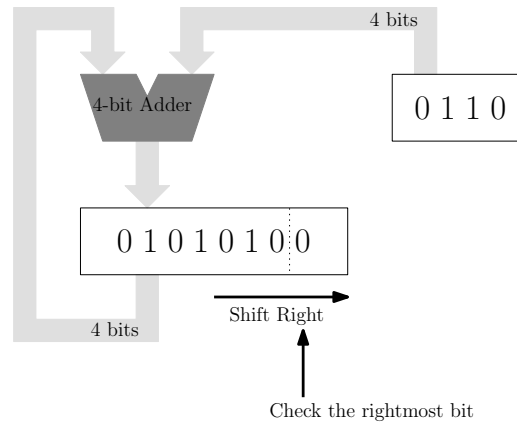


Figure 3.34: After shifting the product register.

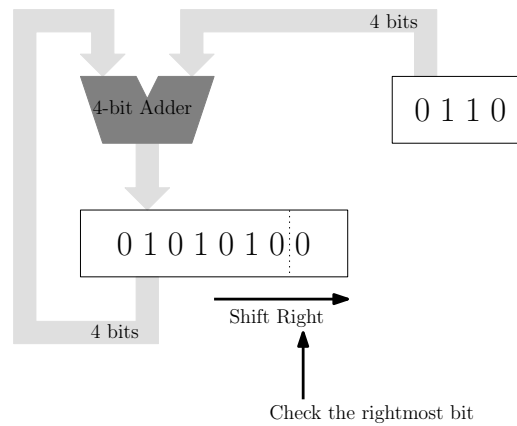


Figure 3.35: No addition since checking the multiplier bit returns 0.

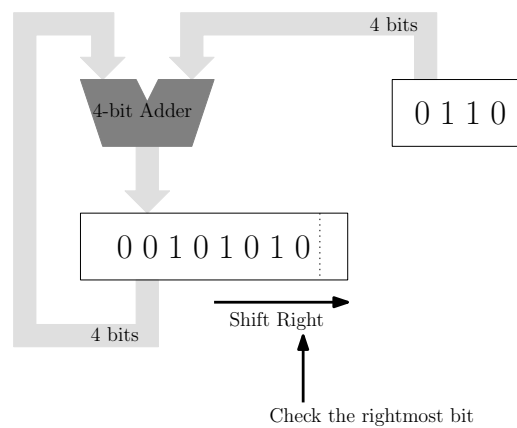


Figure 3.36: Final result, after shifting the product register one last time.

3.3.1 Booth's Algorithm

Replacing the adder in Figure 3.27 by a 32-bit ALU allows to compute subtractions as well as additions. To see how this could benefit the computation of multiplication, we note that:

$$\begin{aligned} 0110_2 \times 0111_2 &= 0110_2 \times (1000_2 - 0001_2) \\ &= 0110_2 \times 1000_2 - 0110_2 \times 0001_2 \\ &= 0110000_2 - 0000110_2 \end{aligned}$$

This shows that the product can be obtained by computing one subtraction, of the 0-shifted multiplicand (i.e., -0000110_2), and one addition, of the triple-left-shifted multiplicand (i.e., $+0110000_2$), as opposed to the 3 additions that were needed in the previously explained long hand scheme in Example 3.3.1 (requiring $+0000110_2$, $+0001100_2$ and $+0011000_2$). This allows us to compute the product more efficiently (i.e., faster), using only 2 instead of 3 arithmetic operations. The previous observation is generalized in “**Booth's algorithm**” which is outlined below. We will first illustrate the algorithm and then prove its correctness.

Booth's Algorithm

Let a be the multiplier and b the multiplicand.

- Examine two bits, a_i and a_{i-1} of the multiplier at a time (let $a_{-1} = 0$).
- Given a_i and a_{i-1} , follow the table:

a_i	a_{i-1}	operation
0	0	do nothing
0	1	add (shifted version of) b
1	0	subtract (shifted version of) b
1	1	do nothing

For example, for $0110_2 \times 0111_2$,

$$\begin{aligned} \text{multiplier: } 011 \underbrace{1}_{=10} &\Rightarrow \text{subtract 0-shifted multiplicand} \Rightarrow -0000110_2 \\ 01 \underbrace{11}_{=11} &\Rightarrow \text{do nothing} \\ 0 \underbrace{11}_{=11} 1 &\Rightarrow \text{do nothing} \\ \underbrace{01}_{=01} 11 &\Rightarrow \text{add triple-left-shifted multiplicand} \Rightarrow +0110000_2 \end{aligned}$$

This shows that Booth's algorithm applies to Example 3.3.1.

So far, we only considered positive numbers. Although Booth's algorithm was originally invented for speed (to minimize the number of arithmetic operations), it turns out that

it handles signed integers, represented in two's complement representation, easily as well. That, by itself, is an important reason to use Booth's algorithm to compute multiplication. We will prove it later.

Before giving an example, consider the following two cases:

- **The multiplicand b is a negative integer, represented in two's complement representation.** Since multiplication boils down to the long hand addition and/or subtraction of shifted versions of the multiplicand, the product will be computed correctly if the additions and/or subtractions of the shifted versions of b are computed correctly. We have shown that long hand addition (or, for subtraction, long hand addition of $-b$, i.e., its two complement) works correctly for two's complement representation. Since b is represented using two's complement representation, the additions and/or subtractions of shifted versions of b will be computed correctly: the proposed scheme will work fine for negative b .
- **The multiplier a is a negative integer, represented in two's complement representation.** In this case, it is not obvious that Booth's algorithm will still compute the correct result. Let's consider an example of applying Booth's algorithm to compute the product of a positive multiplicand with a negative multiplier.

Example 3.3.2

Let's apply Booth's algorithm to compute $0010_2 \times 1101_2$. Since 0010_2 represents the number 2 and 1101_2 represents the number -3, in 4-bit two's complement representation, the answer should be $2 \times (-3) = -6$, or 1111010_2 in 8-bit two's complement representation. Applying Booth's algorithm results in the following scheme:

$$\begin{aligned}
 \text{multiplier: } 110 \underbrace{1}_{=10} &\Rightarrow \text{subtract 0-shifted multiplicand} \Rightarrow -00000010_2 \\
 11 \underbrace{01}_{=01} &\Rightarrow \text{add single-left-shifted multiplicand} \Rightarrow +00000100_2 \\
 1 \underbrace{10}_{=10} 1 &\Rightarrow \text{subtract double-left-shifted multiplicand} \Rightarrow -00001000_2 \\
 \underbrace{11}_{=11} 01 &\Rightarrow \text{do nothing}
 \end{aligned}$$

So, to compute the product, we need to compute

$$\begin{aligned}
 -00000010_2 + 00000100_2 - 00001000_2 &= (-00000010_2) + 00000100_2 + (-00001000_2) \\
 &= 11111110_2 + 0000100_2 + 11111000_2 \\
 &= 00000010_2 + 11111000_2 \\
 &= 11111010_2,
 \end{aligned}$$

where the subtractions are computed by adding the two complement of the argument.

Figures 3.37 to 3.45 show how these calculations are performed using the multiplication hardware presented in Figure 3.27, extended with a small, 1-bit register to store the extra bit of the multiplier Booth's algorithm needs to check (initialized at 0 since $a_{-1} = 0$). At any time, this bit corresponds to the bit of the multiplier that was most recently shifted out of the product register. Also, note that the right shift of the product register will require *sign extension*, to honor the sign of the intermediary result.

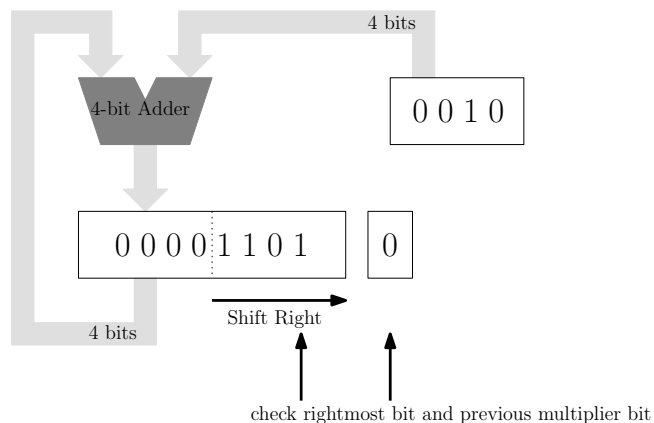


Figure 3.37: Initialization, with $a_{-1} = 0$. The control bits for Booth's algorithm are 10, so, next, a subtraction is required. Instead of subtracting 0010_2 , we will add the two complement, 1110_2 .

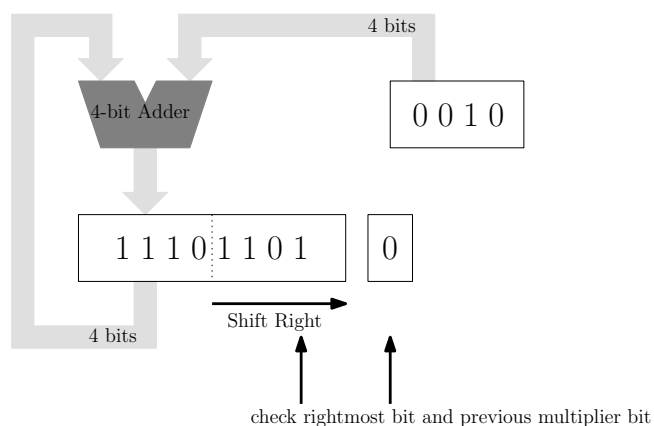


Figure 3.38: After the 1st subtraction, the 4 leftmost bits of the product register have been updated to 1110_2 .

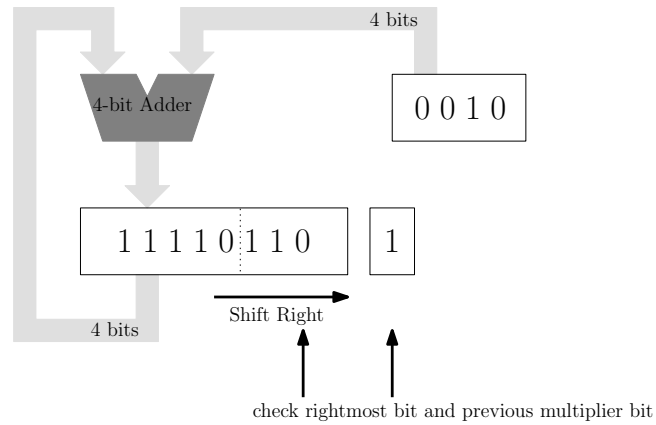


Figure 3.39: After shifting the product register. Note that *sign extension* occurs with the shift. Now, the control bits for Booth's algorithm are 01 , so, next, an addition is required.

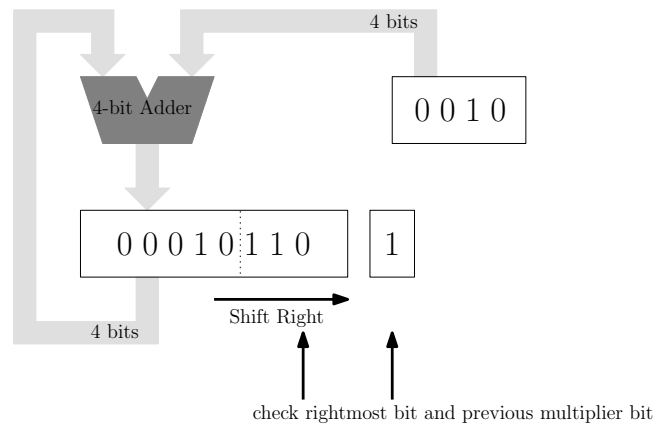


Figure 3.40: After the addition: the 4 leftmost product bits have been updated to 0001_2 .

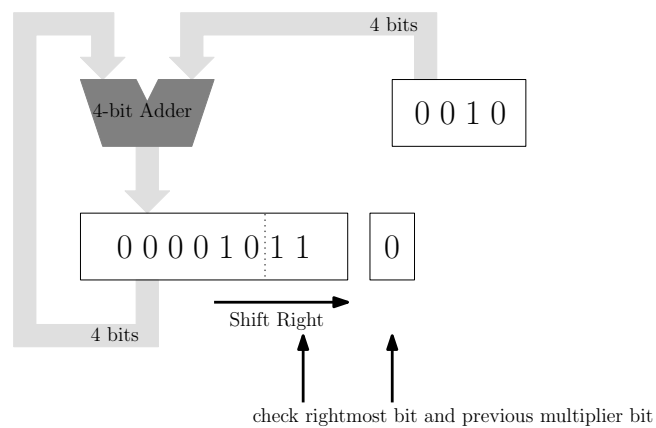


Figure 3.41: After shifting the product register, the control bits, 10 , require a subtraction.

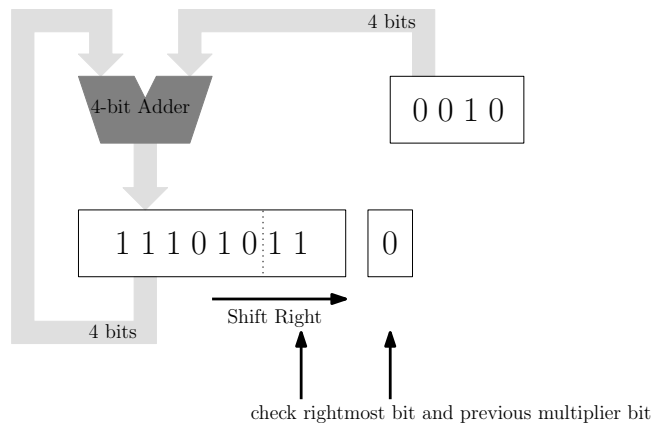


Figure 3.42: After subtracting 0010_2 , i.e., adding 1110_2 , the 4 leftmost bits are 1110_2 .

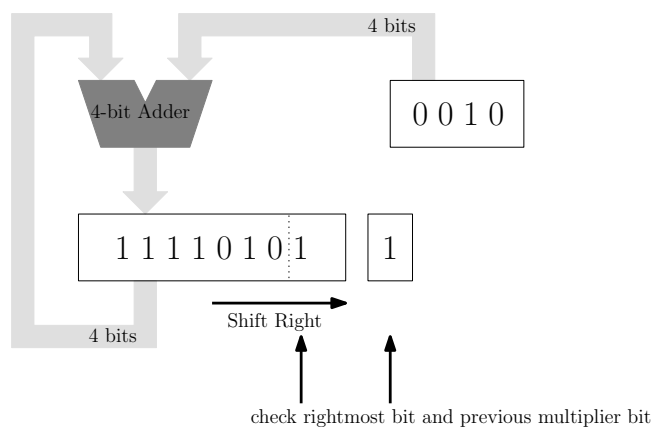


Figure 3.43: After shifting the product register, the control bits, 11, require to do nothing.

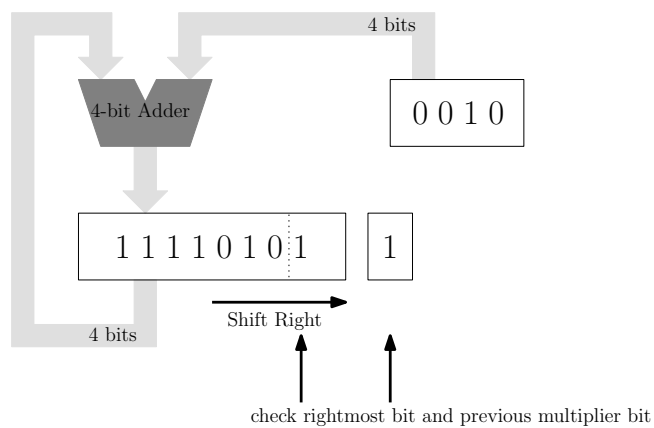


Figure 3.44: Doing nothing.

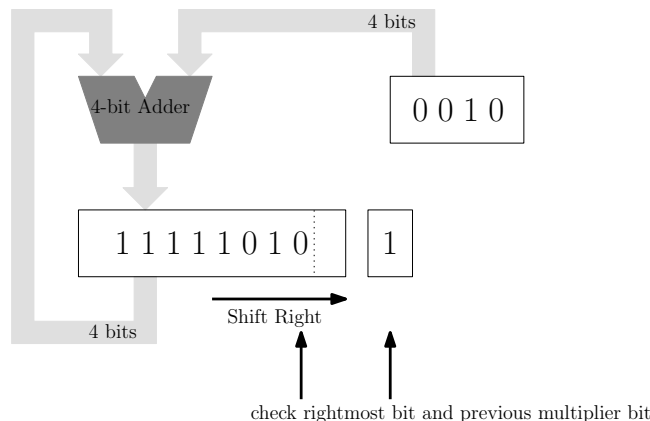


Figure 3.45: After the final shift, the product is available in the product register: 11111010_2 which is the 8-bit two's complement representation of the number -6 .

After illustrating the validity of Booth's algorithm with several examples, we will now prove that this algorithm always provides the correct result, when multiplying integers in two's complement representation. We will provide the proof for 32-bit two's complement representation, however, it can easily be generalized to any N -bit representation.

Proof. Assume we want to calculate $b \times a$, where b is the multiplicand and a the multiplier. Using Property 2 of two's complement representation, we obtain:

$$\begin{aligned}
 b \times a &= b \times \left(-a_{31}2^{31} + \sum_{i=0}^{30} a_i2^i \right) \\
 &= b \times \left(-a_{31}2^{31} + 2 \cdot \sum_{i=0}^{30} a_i2^i - \sum_{i=0}^{30} a_i2^i \right) \\
 &= b \times \left(-a_{31}2^{31} + \sum_{i=0}^{30} a_i2^{i+1} - \sum_{i=0}^{30} a_i2^i \right) \\
 &= b \times \left(\sum_{i=1}^{31} a_{i-1}2^i - a_{31}2^{31} - \sum_{i=0}^{30} a_i2^i \right) \\
 &= b \times \left(\sum_{i=1}^{31} a_{i-1}2^i - \sum_{i=0}^{31} a_i2^i \right) \\
 &= b \times \left(\sum_{i=0}^{31} a_{i-1}2^i - \sum_{i=0}^{31} a_i2^i \right) \\
 &= b \times \left(\sum_{i=0}^{31} (a_{i-1} - a_i)2^i \right) \\
 &= \sum_{i=0}^{31} (a_{i-1} - a_i) b2^i,
 \end{aligned}$$

since $a_{-1} = 0$. This formula matches Booth's algorithm:

- If $a_{i-1} = a_i = 1$ or $a_{i-1} = a_i = 0$, then $a_{i-1} - a_i = 0$, so the corresponding term in the previous sum will not make any contribution. This confirms Booth's algorithm, which instructs to “do nothing” if $a_{i-1} = a_i = 1$ or $a_{i-1} = a_i = 0$.
- If $a_{i-1} = 1$ and $a_i = 0$, then $a_{i-1} - a_i = 1$, so the corresponding term in the previous sum becomes $b2^i$. Since $b2^i$ is nothing but b , shifted to the left i times, this corresponds to adding the i -left-shifted version of b . This, again, confirms Booth's algorithm, which instructs to “add shifted version of b ” if $a_{i-1} = 1$ and $a_i = 0$.
- If $a_{i-1} = 0$ and $a_i = 1$, then $a_{i-1} - a_i = -1$, so the corresponding term in the previous sum becomes $-b2^i$. This corresponds to subtracting the i -left-shifted version of b and confirms Booth's algorithm, which instructs to “subtract shifted version of b ” if $a_{i-1} = 0$ and $a_i = 1$.

This proves the correctness of Booth's algorithm for multiplying integers represented in two's complement representation. \square

Notice that applying Booth's algorithm to multiply any multiplicand with a small (in absolute value), negative multiplier in 32-bit two's complement representation is very efficient. Indeed, the representation of the multiplier will have a long series of leading 1 bits (due to sign extension). While the multiplication scheme in Example 3.3.1 would incur many additions (one for each 1 bit), Booth's algorithm causes significantly fewer arithmetic operations (since a series of 1 bits doesn't require to do anything). So, Booth's algorithm a) can be very efficient and b) allows to deal nicely with negative integers.

3.3.2 Multiplication instructions in MIPS R2000

As shown in Figure 3.27, the product requires a 64-bit register. In MIPS R2000, two additional 32-bit registers, `$Hi` and `$Lo`, in the CPU, implement this 64-bit product register. While `$Hi` stores the 32 most significant bits of the product, `$Lo` stores the 32 least significant bits.

The MIPS R2000 instruction set provides the instruction `mult` for multiplications. After completing a `mult` instruction, the 64-bit product will be stored in `$Hi` and `$Lo`. The registers `$Hi` and `$Lo` cannot be accessed directly and their contents can be moved into two of the 32 general purpose CPU registers using `mfhi` and `mflo` instructions. `mfhi` (move from `$Hi`) will move the contents of `$Hi` to the register specified in the instruction, and `mflo` (move from `$Lo`) will move the contents of `$Lo` to its argument. For example,

```
mult $8,$9 ⇔ multiply two numbers, stored in $8 and $9.
mfhi $10 ⇔ move MSB of the product, in $Hi, to $10.
mflo $11 ⇔ move LSB of the product, in $Lo, to $11.
```

After invoking the three instructions above, the result can be retrieved by accessing the registers `$10` and `$11`. One can then determine how to store the product in a single 32-bit

register only (e.g., if the product is not too large, one can continue to work with `$11` only).

To multiply two *natural* numbers, the instruction `multu` can be used. The two arguments of `multu` are interpreted as natural numbers and the result stored in `$Hi` and `$Lo` also represents a natural number.

To divide, the instructions `div` and `divu` are provided in the MIPS R2000 instruction set. Similar to `mult` or `multu`, the result will be stored in `$Hi` and `$Lo`. For a division, the result consists of a *quotient* and a *remainder*. The quotient will be stored in `$Lo` and the remainder in `$Hi`. We can access the quotient and the remainder using the same `mflo` and `mghi` instructions as for multiplication. The actual division algorithm is beyond the scope of this class.

3.4 Representing Real Numbers

3.4.1 Fixed point number representation

In a fixed point number representation, the location of the binary point is fixed. While the representation provides an integer part and a fractional part (left respectively right of the binary point), computing the value corresponding to a given representation is similar to doing so for a signed or unsigned integer. For example, if the location of the binary point is between bit 3 and 4, `101100011` represents

$$\begin{array}{r}
 \underbrace{10110}_{\text{integer part}} \quad \underbrace{\cdot}_{\text{fixed (binary) point}} \quad \underbrace{0011}_{\text{fractional part}} \\
 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} \\
 = 22.1875
 \end{array}$$

This representation is specified as a 5:4 fixed point representation, which indicates the integer part consists of 5 bits and the fractional part of 4 bits. A fixed point representation can be unsigned, or it can be a signed (two's complement) representation, analogous to how integers were represented. For example, interpreting `1101` as an unsigned fraction (where a fraction is a number smaller than 1, in absolute value) puts the binary point to the left of the MSB and results in a value of

$$.1101 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = .8125$$

Interpreting `1101` as a signed fraction puts the binary point to the right of the MSB (sacrificing the MSB for the signed representation) and, using a two's complement representation, results in a value of

$$1.101 = 1 \cdot (-2^0) + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = -0.375$$

Moreover, the two's complement principles work similarly for a fixed point representation. For example, the two's complement of `0.101` (representing `0.625`) can be obtained by flipping all bits and adding 1 to the least significant bit, resulting in `1.011`, which represents `-0.625`.

Although it is simple to represent real numbers using a fixed point representation, the range of represented numbers is rather limited, since the location of the binary point is fixed, which immediately fixes the number of bits representing the integer part and thus the range of representable numbers. The floating point representation, which is introduced in the next subsection, addresses this issue.

3.4.2 Floating-point number representation

The floating-point number representation is adapted in the IEEE 754 standard. This representation can handle a wider, dynamic range. Also, the number of bits after the binary point does not need to be specified and fixed. There is a single precision and a double precision floating-point representation. They have the same structure, just a different number of bits.

Single precision

A single precision floating-point representation consists of 32 bits. The bit usage is given by the following table.

31	30-23	22-0
sign	exponent	mantissa

This represents the following value: $(-1)^{\text{sign}} \times \text{mantissa} \times 2^{\text{exponent}}$.

- The mantissa m is a normalized fraction, i.e., $1 \leq m < 2$. The mantissa can always be brought into this range by adjusting the exponent. For example,

$$0.01011 \times 2^{E+2} = 1.011 \times 2^E.$$

In order to pack more bits into the representation, the bit to the left of the binary point of the mantissa is made *implicit*. It is always 1 anyways. Thus, if $b_{22}b_{21}...b_0 = 01110...000$, then the number actually represents $1.01110...000 \times 2^E$.

- The exponent is a *biased* integer (bias = 127). This means that the 8 exponent bits represent a value that can be derived as follows:

$$\begin{aligned} b_{30}...b_{23} = 0000\ 0000 &\text{ represents } 0 - 127 = -127; \\ b_{30}...b_{23} = 0111\ 1111 &\text{ represents } 127 - 127 = 0; \\ b_{30}...b_{23} = 1000\ 0000 &\text{ represents } 128 - 127 = 1; \\ b_{30}...b_{23} = 1111\ 1111 &\text{ represents } 255 - 127 = 128. \end{aligned}$$

The bias allows to represent not only positive powers but also negative powers of 2.

Double precision

A double precision floating-point representation consists of 64 bits. The bit usage is given by the following table.

63	62-52	51-0
sign	exponent	mantissa

The roles of sign, exponent, and mantissa are the same as for the single precision case.

Example 3.4.1

The following is a single precision floating-point number. What number does this represent?

31	30-23	22-0
1	10000001	01100...000

- sign = 1
- exponent = $10000001_2 - 127 = 129 - 127 = 2$
- mantissa = $1.01100\dots00 = 1 \cdot 2^0 + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1.375$

Therefore, it represents $-1.375 \times 2^2 = -5.5$.

Example 3.4.2

What is the single and double precision floating-point representation of -0.875 ?

Since $0.875 < 1$, it cannot be the mantissa for the representation. To obtain a mantissa $1 \leq m < 2$, we can rewrite this as

$$-0.875 = (-1)^1 \times 1.75 \times 2^{-1}$$

. Therefore,

- sign = 1
- exponent = $-1 = 126 - 127 = \underbrace{01111110_2}_{\text{exponent representation}} - 127$
- mantissa = $1.75 = 1 + 0.5 + 0.25 = 1.1100\dots00_2 = 1 + 0. \underbrace{1100\dots00_2}_{\text{mantissa representation}}$

The single precision floating-point representation is thus given by

31	30-23	22-0
1	01111110	11000...000

Similarly, the double precision floating-point representation is given by

63	62-52	51-0
1	0111111110	11000...000