**Chapter 10: Computer Arithmetic**

Table of Contents

[10.1 The Arithmetic and Logic Unit 2](#_Toc50661785)

[10.2 Integer Representation 3](#_Toc50661786)

[Sign-Magnitude Representation 3](#_Toc50661787)

[Twos Complement Representation 3](#_Toc50661788)

[Range Extension 4](#_Toc50661789)

[10.3 Integer Arithmetic 5](#_Toc50661790)

[Negation 5](#_Toc50661791)

[Addition and Subtraction 5](#_Toc50661792)

[Multiplication 9](#_Toc50661793)

[Division 11](#_Toc50661794)

[10.4 Floating-Point Representation 13](#_Toc50661795)

[10.5 Floating-Point Arithmetic 15](#_Toc50661796)

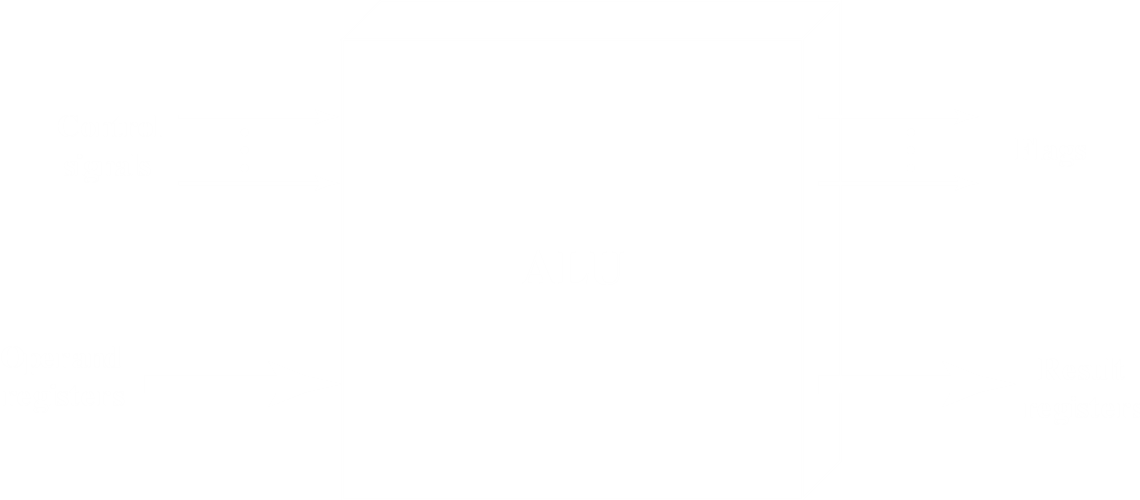
[Addition and Subtraction 16](#_Toc50661797)

[Multiplication and Division 17](#_Toc50661798)

[Precision Considerations 17](#_Toc50661799)

## 10.1 The Arithmetic and Logic Unit

The ALU is the part of the computer that performs arithmetic and logical operations on data. All electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. The figure below shows how the ALU is interconnected with the rest of the processor.



Operands for arithmetic and logic operations are presented to the ALU in registers and the results are stored in registers, both connected by single paths. The ALU may also set flags as the results of an operation, such as an overflow flag when the results of a computation exceed the length of the register into which it is to be stored. The flag values are also stored in registers within the processor. The processor provides signals that control the operation of the ALU and the movement of data into and out of the ALU.

## 10.2 Integer Representation

In binary number systems we can use digits, the minus sign and the radix point to represent numbers. These last two things are not available in computer storage and processing, so we need to turn to alternative methods. The representation of integers in the computer system is sometimes referred to as fixed-point representation, since the radix point is fixed.

### Sign-Magnitude Representation

In sign-magnitude representation, the most significant bit is considered to represent the sign of the number. If it is , the number is positive. If it is , the number is negative.

Drawbacks to this system include the need to consider both the sign and the magnitudes of numbers and the existence of two s, a positive one and a negative one, making it more difficult to test for , an operation that is very frequently used.

### Twos Complement Representation

In twos complement, the most significant bit is still used to represent sign, but the treatment for the other bits is different. We take the number, flip the bits for the binary representation of that number and add a binary 1 to it. Thus, if we need to represent , we take the binary representation of , which is , flip each of its bits making it , and add to it meaning is represented as .

Although this method may seem awkward, it is actually the easiest method to perform arithmetic calculations from the computer’s point of view, which we will see in the next section.

### Range Extension

If we want to represent the same number using a large number of bits, we can simply add in more s. If we are using sign magnitude, the method is as simple as moving the sign bit to the new left-most position.

However, this method does not work with twos complement. Instead, we must use the sign bit, whether it be or , to fill in the new spots, still keeping the actual sign bit on the left-most bit.

## 10.3 Integer Arithmetic

### Negation

There are two problems with the process of negating a number using twos complement. The first is if you try to negate , there is an overflow. This is not much a problem since we can simply ignore the overflow and the result is still valid. The second problem is a bit more concerning. If we try to use numbers that take up the sign bit as part of their magnitudes, we will get confusing results. For example, the number , which is meant to represent , would still be after negation. As a result of this problem, under the twos complement system, the range of numbers we use is from to , where is the number of bits.

### Addition and Subtraction

Addition on twos complement numbers are carried out as though it were normal binary numbers. Sometimes a carry may occur, which is ignored. An overflow occurs and causes an incorrect result if and only if two positive or two negative bits are being added and the sign bit changes. In this case, the ALU must indicate that an overflow has occurred and the result should not be used.

1001 = -7

+0101 = +5

1110 = -2 (positive and negative numbers; no carry; no overflow)

1100 = -4

+0100 = +4

10000 = 0 (positive and negative numbers; carry; no overflow)

0011 = +3

+0100 = +4

0111 = +7 (positive numbers; no carry; no overflow)

1100 = -4

+1111 = -1

11011 = -5 (negative numbers; carry; no overflow)

0101 = +5

+0100 = +4

1001 = (positive numbers; no carry; overflow)

1001 = -7

+1010 = -6

10011 = (negative numbers; carry; overflow)

When subtracting one number from another, you take the twos complement of the number being subtracted and add it to the number being subtracted from. The same overflow rule still applies for the addition part.

0010 = +2

~~-0111 = +7~~

+1001 = -7

1011 = -5 (positive and negative number; no carry; no overflow)

0101 = +5

~~-0010 = +2~~

+1110 = -2

10011 = +3 (positive and negative number; carry; no overflow)

1011 = -5

~~-0010 = +2~~

+1110 = -2

11001 = -7 (negative numbers; carry; no overflow)

0101 = +5

~~-1110 = -2~~

+1110 = +2

0111 = +7 (positive numbers; no carry; no overflow)

0111 = +7

~~-1001 = -7~~

+0111 = +7

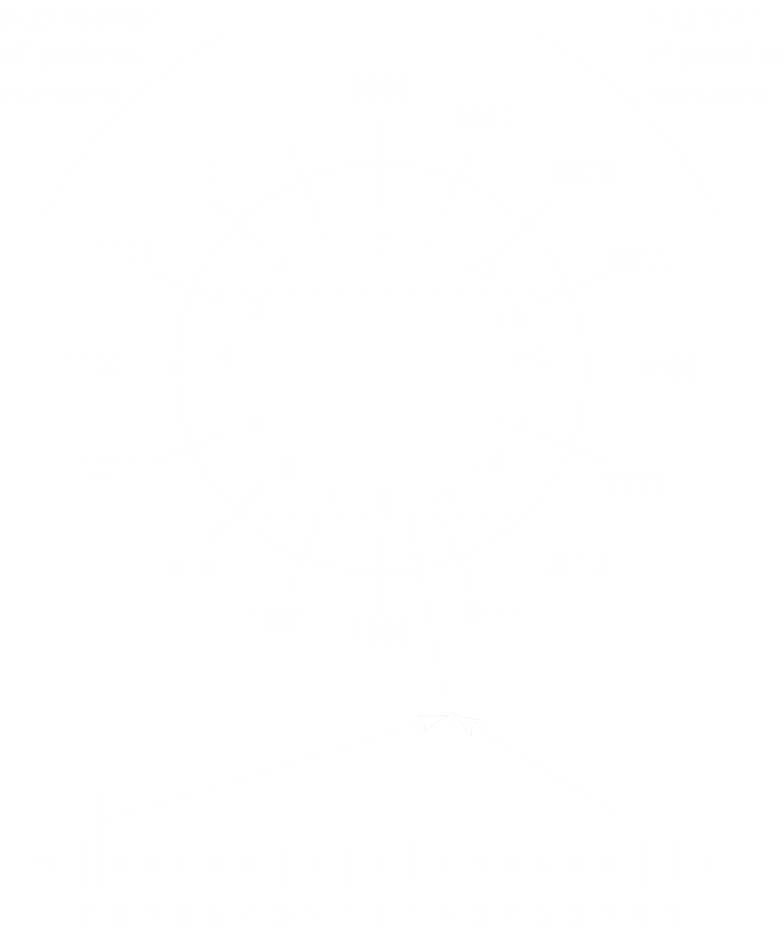
1110 = (positive numbers; no carry; overflow)

1010 = -6

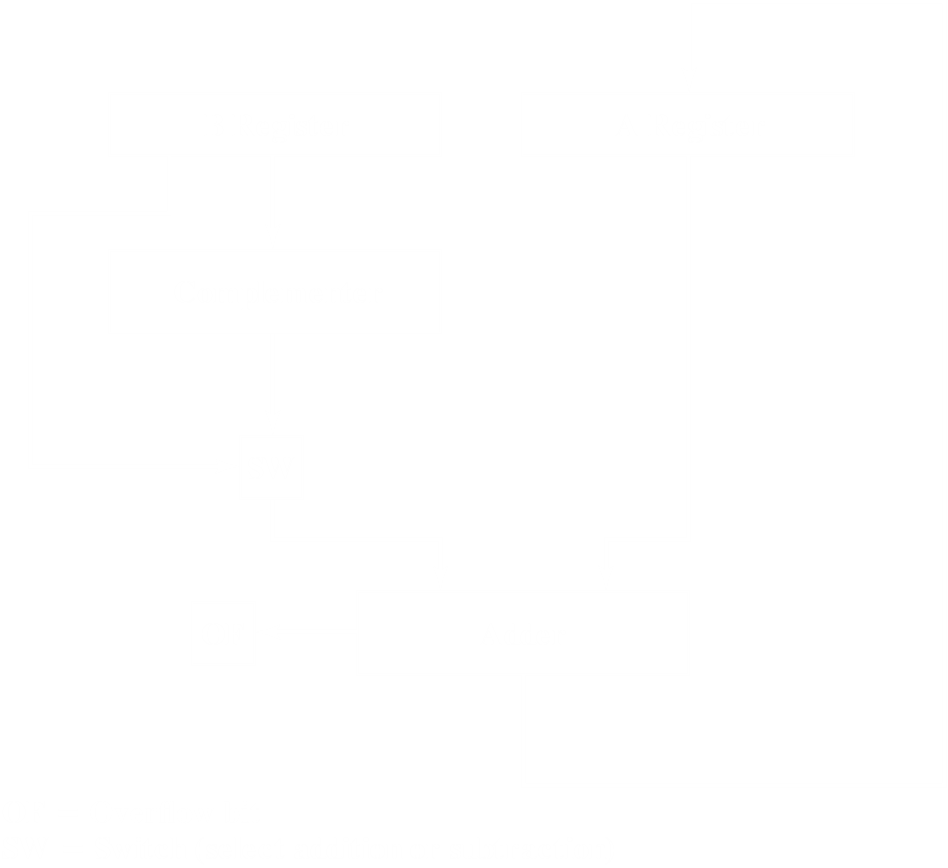
~~-0100 = +4~~

+1100 = -4

10110 = (negative numbers; carry; overflow)



The figure above should make addition and subtraction in twos complement even easier for humans. Starting at any number, we can add by moving positions clockwise and subtract by moving positions counter-clockwise. If we go beyond when adding or when subtracting, an overflow has occurred. Similar figures can be made for bit numbers.



The figure above shows the data paths and hardware elements needed for addition and subtraction. The main element is the binary adder, which takes two numbers and produces a sum and an overflow indicator. It treats the numbers as though they were unsigned integers. They are taken from two registers and the result may be stored in one of the registers or a new one. The overflow indicator is stored in a 1-bit overflow flag. For subtraction, the number being subtracted is passed through a twos complementor before being passed to the adder. Note that the control signals needed to tell the setup if addition or subtraction should be done is not shown here.

### Multiplication

First let us consider the multiplication of unsigned binary integers. The process of multiplication involves generating partial products for each digit in the multiplier and then summing them up.

1011

x1101

1011

0000

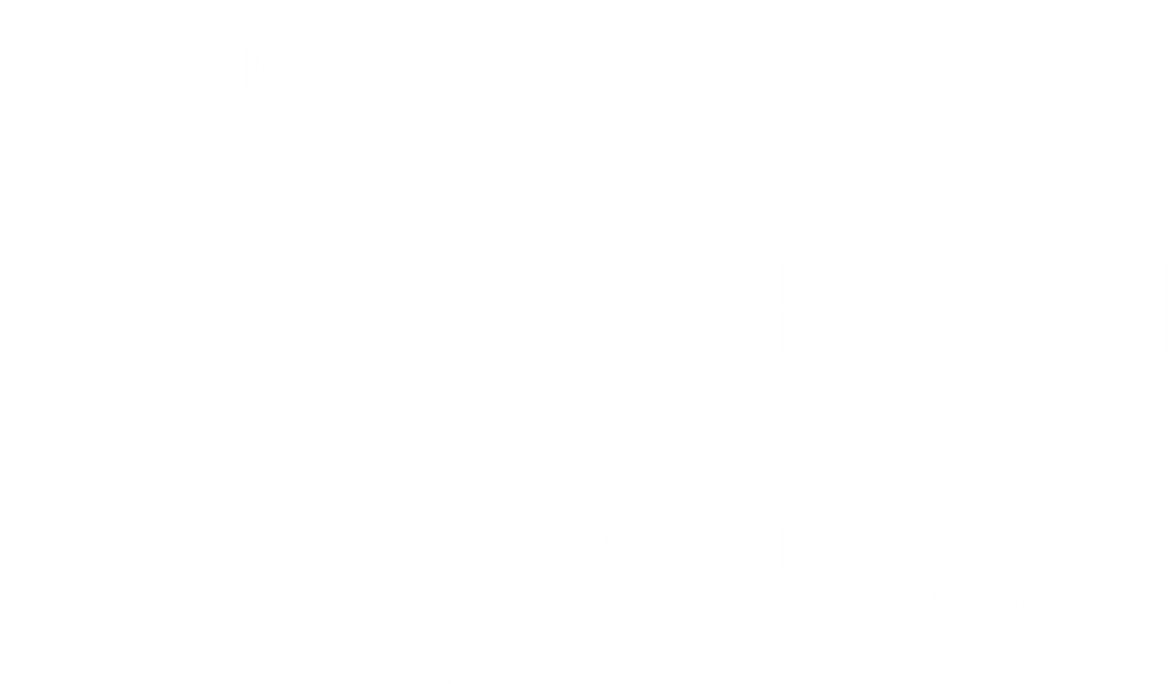
1011

1011 \_

10001111

The partial products are easy to find. During the summation, each successive partial product is shifted one position to the left relative to the preceding partial product. The multiplication of two -bit binary integers results in a product of up to bits in length.

With computers, we make a few improvements to the process. We can perform a running addition on the partial products rather than waiting until the end, which means less storage and fewer registers are needed. We can also reduce some time during generation of partial products, since a would require an add and a shift operation but a would only require a shift operation.



The multiplier and multiplicand are loaded into two registers, while a third register holds the results. There is another -bit register used to hold a potential carry bit resulting from addition. Control logic reads the multiplied one at a time. If it reads a , then the multiplicand is added to the result with any overflow bit also being stored. Then, the overflow, result and multiplicand are all shifted -bit to the right so that goes into , goes into and is lost. For a -bit, no addition is performed, only the shift. This process is repeated for each bit in the multiplier. The resulting -bit product is stored in the and registers.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C | A | Q | M |  |  |
| 0 | 0000 | 1101 | 1011 | Initial Values | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| 0 | 0101 | 1110 | 1011 | Shift |
| 0 | 0010 | 1111 | 1011 | Shift | Second Cycle |
| 0 | 1101 | 1111 | 1011 | Add | Third Cycle |
| 0 | 0110 | 1111 | 1011 | Shift |
| 1 | 0001 | 1111 | 1011 | Add | Fourth Cycle |
| 0 | 1000 | 1111 | 1011 | Shift |

A simple method like this will, unfortunately, not work with twos complement. Consider the multiplication below. It is incorrect.

1001 (-7)

0011 (+3)

00001001

00010010

00000000

00000000

00011011 (+27)

If either the multiplicand or multiplier or both are negative, multiplication like this does not work. This is because the sign bits do not line up since the actual multiplication is still treating the numbers as though they were unsigned. If we take the sign bit into account while generating the partial products, the calculations will be correct.

1001 (-7)

0011 (+3)

11111001

11110010

00000000

00000000

11101011 (-21)

In multiplications where the multiplier is negative instead, the most significant bit carries a sign value and a magnitude. Thus, it needs to be twos complemented.

0111 (+7)

1101 (-3)

00000111

00000000

00011100

11001000 (This value has been complemented.)

11101011 (-21)

1001 (-7)

1101 (-3)

11111001

00000000

11100100

00111000 (This value has been complemented.)

00010101 (+21)

Note: Parts of this section have been skipped over.

Translation: বই পড়ে কিছু বুঝি নাই।

### Division

We can perform binary division like this:

0010

0011|0110

0

01

0

011

011

00

0

x

If either of the number is negative, we need to take the twos complement of the quotient. If both the numbers are negative, the quotient we find is the answer.

If the dividend is negative, we need to take the twos complement of any remaineder as well.

Note: Parts of this section have been skipped over.

Translation: বই পড়ে কিছু বুঝি নাই।

## 10.4 Floating-Point Representation

With fixed-point notation, it is possible to represent a range of positive and negative integers cantered on or near . By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well. The problem is that very large numbers or very small fractions cannot be represented, and the fractional part of the quotient in a division of two large numbers could be lost.

For decimal numbers, such problems are avoided by using scientific notation. The same approach can be taken with binary numbers in the format . Here, is the significant portion, is the base and is the exponent. Thus, the speed of light would be . Computers store floating point numbers like this as well, with the base being .

0 10010011 10100010000000000000000

In the above number, the first bit is the sign bit. If it is , the number is positive and if it is the number is negative. The next 8 bits are the exponent. These can be read just like a normal binary number, so in this case the number is . This value is actually in biased representation. We actually subtract from this to get the actual exponent. This means we can store both very large floating-point numbers and very small ones. Here, the actual exponent value is . The last 23 bits represent the actual significand number. From left to right, we are dealing with negative powers of base . Thus, here, the value is . However, the way this system works is, we need to add to this value so the actual value is . This is called normalizing the number. Thus, our final floating-point number is .

Using floating-point representation, we can use numbers from to and from to . In either the negative or positive side, anything larger is called an overflow and anything small is called an underflow.

There is a trade-off between range and precision here. If we increase the number of bits devoted to the exponent, we can increase the range. However, in doing so, we will have fewer bits for the significand, which will reduce the precision.

An exponent of and a fraction of represents a . An exponent of all s and a fraction of represents infinity.

## 10.5 Floating-Point Arithmetic

For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment.

(assuming )

(assuming )

We may face a few conditions:

* Exponent overflow where a positive exponent exceeds the maximum possible exponent value. Some systems may report this as infinity.
* Exponent underflow where a negative exponent exceeds the minimum possible exponent value. Some systems may report this as .
* Significand underflow where digits flow off the right end off the significand when trying to align them. This will require some form of rounding.
* Significant overflow where the most significant bit may be lost. This can be fixed by realignment.

### Addition and Subtraction

The four basic phases for addition and subtraction are:

* Check for s
* Align the significands
* Add or subtract the significands
* Normalize the result

The two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit (the that needs to be added) it must be made explicit.

In the first phase, if it is a subtraction, we flip the sign of the subtrahend. Next, if either operand is 0, the other is reported to be the answer.

In the second phase, we manipulate the numbers so that the two exponents are equal. Alignment is achieved by shifting either the smaller number to the right or the larger number to the left. Since either operation may result in the loss of digits, the smaller number is shifted so that any digits lost are of small significance. If this process results in a value for the significand, the other number is reported as the result. Thus, if two numbers have significantly different exponents, the less number is lost.

In the third phase, the significands are added together, taking into account their signs. There is a possibility the result will be here, or that there will be a significand overflow by digit. If there is an overflow, the significand is shifted to the right and the exponent is incremented. An exponent overflow could occur here which would be reported and the operation halted.

In the last phase, the significand digits are shifted left until the most significant digit is non-zero. Each shift causes a decrement of the exponent. We could have an exponent underflow here. The result is rounded off and reported.

### Multiplication and Division

In multiplication, if either operand is , the result is reported as . In the other case, the exponents are added. Since the exponents are in biased form, the result would have double the bias so the bias value must be subtracted. If there is an exponent overflow or underflow, this is reported and the algorithm ends. If the exponent is in the correct range, the significands are multiplied. This is the same as for integer multiplication. The result will be double in length and the extra bits are lost during rounding. The result needs to be normalized as well, which could again cause an exponent underflow.

In division, if the divisor is an error is reported, or the result is set to infinity. If the dividend is , the result is reported to be . Otherwise, the divisor exponent is subtracted from the dividend exponent. This causes the bias to be lost so it must be added back in. Exponent overflow or underflow is also checked for. Next, the significands are divided followed by normalization and rounding.

### Precision Considerations

When loading the exponent and significand into the ALU registers, the register almost always has enough space. It also has guard bits used to pad out the right end of the significant with s. This is to ensure least significant bits are not lost during alignment.

The rounding policy also affects precision. The result of any operation on significands is usually stored in longer registers. When the result is put back into the floating-point format, extra bits must be eliminated in a way so as to produce a result closest to the exact result. There are four approaches to this:

* Round to nearest representable number
* Round up towards positive infinity
* Round down towards negative infinity
* Round towards , simply ignoring extra bits