**Number Systems**

Contents

[Number Systems 2](#_Toc325285348)

[A - Unsigned Positional Number Systems 3](#_Toc325285349)

[Addition 4](#_Toc325285350)

[Conversion 5](#_Toc325285351)

[Subtraction 6](#_Toc325285352)

[Hexadecimal 8](#_Toc325285353)

[Finite Precision Arithmetic and Overflows 10](#_Toc325285354)

[Precision versus Accuracy 12](#_Toc325285355)

[B - Signed Positional Numbers 1](#_Toc325285356)

[Signed Magnitude 3](#_Toc325285357)

[One’s Complement 4](#_Toc325285358)

[Two’s Complement 5](#_Toc325285359)

[How can I do hex two's complement problems 8](#_Toc325285360)

[A summary of why two’s complement is significant 11](#_Toc325285361)

[C - Number Systems Review 1](#_Toc325285362)

[Unsigned Positional Number Systen Review 1](#_Toc325285363)

[Signed Positional Number Systems Review 2](#_Toc325285364)

[Two’s Complement Number Systems Review 3](#_Toc325285365)

[Hints on converting signed numbers to decimal 4](#_Toc325285366)

[D - Additional Thoughts On Number Systems 1](#_Toc325285367)

[Why does flipping the bits + 1 create the two's complement 2](#_Toc325285368)

[Why does end around carry work for One's Complement 3](#_Toc325285369)

[Why does the carry-in / carry-out rule detect a two's complement overflow 4](#_Toc325285370)

[An alternate way to calculate two's complement 8](#_Toc325285371)

[Can you just look at negative number and tell its decimal value 9](#_Toc325285372)

[How can an overflow be detected in signed magnitude addition 10](#_Toc325285373)

[How can an overflow be detected in ones complement addition 11](#_Toc325285374)

[How can an overflow be detected when adding a two's complement signed number and an unsigned number. 12](#_Toc325285375)

[How can you do math, such as add and subtract, in the various systems 15](#_Toc325285376)

[Additional insight 17](#_Toc325285377)

[E - Fractions 1](#_Toc325285378)

[Conversion of fractions among decimal, binary and hex 2](#_Toc325285379)

# Number Systems

Number systems are the foundation upon which Computer Science is built.

To be successful in the following endeavors:

* understanding computer architecture
* designing computer systems
* assembly language programming
* high level language programming

one must have a solid understanding of number systems.

The basic number system topics include adding, subtracting and converting unsigned and signed integers using bases 2, 10 and 16.

Advanced number system topics include multiplication, division and fractions in those same bases.

The discussion of number systems is broken into these parts.

1. Unsigned number systems
2. Signed number systems
3. A summary of unsigned and signed number systems
4. A collection of problems, proofs and additional thoughts on number systems
5. Fractions

# A - Unsigned Positional Number Systems

The number systems we are interested in are Positional Number Systems. All PNS have a base and set of digits that run from the 0 to the base-1.

For example, our decimal number system has these characteristics.

PNS Name Base Digits

Decimal 10 0...9

When we write a number such as 17910 this is just shorthand for the way that the number really exists in its true positional format which is:

1 x 102 + 7 x 101  + 9 x 100 = 100 + 70 + 9 = 179

That is, there is a set of positions represented by the base raised to successively higher powers (starting with zero) and a number is generated by multiplying the value of that position by a digit.

We use a subscript to indicate the specific base being used. So we know that 10112 is a binary number and not a decimal number.

Not all number systems are positional. Roman Numerals is not a positional number system. Roman Numerals use letters to represent numbers: I=1 II=2 III=3 IV=4 V=5 X=10 L=50 C=100. Below we 1+1 and 2+1 and 3+1. clearly a special rule is needed for the last calculation.

|  |  |  |
| --- | --- | --- |
| I  + I  II | II  + I  III | III  + I  IV |

The primary reason that we are interested in PNS is that it is easy to develop rules to do arithmetic in PNS.

The PNS used by computers is binary. Its base is 2 and its digits are 0 and 1. Computers use binary because each *bi*nary digi*t* (bit) can represent the state of an electronic switch: 0=off, and 1=on. When we write a binary number such as 10112 then this is a shorthand for the way that number really exists which is:

1 x 23 + 0 x 22 + 1 x 21 + 1 x 20

To determine the decimal value of that binary number we can just perform the arithmetic in decimal and we get 8 + 0 + 2 + 1 = 1110

Addition

The key to PNS is that we can perform addition directly on the digits. We do this right to left (least significant position to most significant position) one position at a time following this rule:

If the sum of the digits is less than the base

then the result is represented by a single digit,

Else (the sum is equal or greater than the base)

then the result is reduced by the base and represented by a single digit

and a carry into the next higher position.

This rule may seem obvious, but it will play a major role when we study signed numbers and how computer architects decide how to build modern computers.

Lets apply the rule to two examples of addition in decimal (3+4) and (7+5).

carry

1

3 7 7+5=12 units.

+ 4 + 5 12 units is greater than the base of 10

- -- so it is reduced by the base with the

sum is 7 12 result of 2 and a carry into the next

less than position.

the base

Since there are only two digits in binary, we can see all combinations of adding two binary digits.

carry

1

0 0 1 1

+ 0 + 1 + 0 + 1

- - - -

0 1 1 10

The rule can also be used on multiple digit numbers. We just apply the rule to all the positions, going from the least significant to the most significant.

This example adds the binary value 1011 (equal to decimal 11) to itself.

1 1 <-- carry

1 0 1 12

+ 1 0 1 12

-------

1 0 1 1 02

To verify the result is correct we convert it to decimal by performing the arithmetic of multiplying each digit by its positional value:

16 + 0 + 4 + 2 + 0 = 2210

Conversion

To convert from binary to decimal just perform the arithmetic in decimal.

This is called *expansion of powers,*

so 101102 = 1 0 1 1 0 <-- digits

16 8 4 2 1 <-- positional values

= 16 + 0 + 4 + 2 + 0 = 2210

To convert from decimal to binary perform *reduction of powers*.

2210 = \_ x 25 + \_ x 24 + \_ x 23 + \_ x 22 + \_ x 21 + \_ x 20

We just need to fill in the blank spaces with digits.

25 = 32 which is greater than 22 so the left most binary digit is 0

24 = 16 so that digit is 1 and we have 22-16 = 6 left to convert

23 = 8 which is greater than 6 so that digit is 0

22 = 4 so that digit is 1 and we have 6-4 = 2 left to convert

21 = 2 so that digit is 1 and we have 2-2 = 0 left to convert

20 = 1 which i greater than 0 so that digit is 0.

The converted number is:

2210 = 0x25 + 1x24 + 0x23 + 1x22 + 1x21 + 0x20 = 0101102

Subtraction

Real subtraction is not actually performed by modern computers. The operation C=A-B is performed by adding complements. So C=A-B will be done as C=A+(-B). However, we consider unsigned subtraction for completeness.

To perform real subtraction we work from right to left. If we have to subtract a larger digit from a smaller digit then we re-distribute the data to allow the subtraction process to continue.

First, the decimal example 85 - 7.

We start by rewriting the numbers in their positional format.

85 = 8x101 + 5x100

- 07 = 0x101 + 7x100

The first operation 5-7 cannot be performed. So we redistribute the data from next most significant position. One of 101 values becomes 10x100. Then we perform the subtraction of (10+5) - 7 = 8.

7x101 10x100

85 = 8x101 + 5x100 = ~~8x10~~1 + 5x100

- 07 = 0x101 + 7x100 = 0x101 + 7x100

78 7x101 + 8x100

Now the binary example 100 - 001.

We start by rewriting the numbers in their positional format.

100 = 1x22 + 0x21 + 0x20

- 001 = 0x22 + 0x21 + 1x20

The first operation 0-1 cannot be performed. So we redistribute the data from next most significant positions. Then we perform the subtraction.

1x21 1x20

0x22 ~~1x2~~1 1x20

100 = 1x22 + 0x21 + 0x20 = ~~1x2~~2 + 0x21 + 0x20

- 001 = 0x22 + 0x21 + 1x20 = 0x22 + 0x21 + 1x20

011 0 1 1

Performing unsigned subtraction the way just described is cumbersome when working with large numbers. We now present a technique for speeding up the unsigned subtraction process.

The problem of redistribution occurs whenever we have to subtract 0 - 1.

We can use the technique of *rewriting a 1 bit followed a string of 0 bits*

*as a 0 bit followed by a string of 1 bits plus an additional 1*.

For example:

12

100002 = 011112

Note that both sides have the same value.

* 100002 = 1610
* 011112 + 12 = 1510 + 110  = 1610 = 100002

However the right side makes it easier to perform subtraction.

Subtracting 1 from the right side does not require any redistribtion.

Subtracting 1 from the left side requires a lot of redistribution.

12

100002 011112

- 000012 000012

011112

We perform this process any time we need to subtract 0 - 1. In this example,

bold italics with strike through shows which ***~~10~~*** pair becomes ***01*** + ***1***.

***1***  1 1

***1*** ***1***1 11 ***1*** 11

***01***  ***0~~0~~***1 001 ***01***001

2210 = 101102 101***~~10~~*** 10***~~1~~***10 10110 ***~~10~~***110

- 1110 = - 010112 = 01011 = 01011 = 01011 = 01011

1110 = 010112 1 11 011 01011

Hexadecimal

Many years ago the telephone industry determined that the average person could remember 7 digits and this became the basis for 7 digit telephone numbers.

This means that large binary numbers such as 101101011110101011000011 will be hard to remember and work with. So we use shorthand for binary. We group 4 binary digits together: 1011 0101 1110 1010 1100 0011. Each group of 4 binary digits can have the binary values 0000 to 1111 which correspond to the decimal values 0 to 15.

The digits of a PNS run from 0 to the base-1. So if we have digits that run from (0) to (15) then the base will be 16. This is the hexadecimal system. However *15* is not a good digit! How could we tell the signle digit *15* from the two digit number 15. Instead we need a single character to represent each digit and so the letters A to F are used for the hexadecimal digits *10* to *15*. This table shows the corresponding decimal, hex and binary values.

Decimal Hex Binary

0 0 0000

1 1 0001

2 2 0010

3 3 0011

4 4 0100

5 5 0101

6 6 0110

7 7 0111

8 8 1000

9 9 1001

10 A 1010

11 B 1011

12 C 1100

13 D 1101

14 E 1110

15 F 1111

Conversion between binary and hex just uses this look-up table.

1011 0101 1110 1010 1100 00112 = B5EAC316

Conversion from hex to decimal uses the standard expansion of powers.

1A716 = 1x162 + 10x161 + 7x160 = 42310

Conversion from decimal to hex uses the standard reduction of powers.

42310 = \_\_ x162 + \_\_ x161 + \_\_ x160

162 = 256 There is one 256 in 423 so the digit is 1 and 423-256=167 left.

161 = 16 There are ten 16s in 167 so the digit is A and 167-160=7 left.

160 = 1 There are seven 1s in 7 so the digit is 7.

42310 = 1 x162 + A x161 + 7 x160 = 1A716

Adding and subtracting hex uses the standard rule for PNS.

Below is the math for each column for adding the hex numbers 1A7 + B89. I show the hex digits being added, their decimal equivalents, the final hex result.

Carry ----> 11 hex digits decimal addition final hex result

1A7 1. 7+9 7+9 = 16 0 plus a carry

+ B89 2. 1+A+8 1+10+8 = 19 3 plus a carry

--- 3. 1+1+B 1+1+11 = 13 D

D30

We can check our results by using expansion of powers.

* 1A716 = 42310
* B8916 = 11x162 + 8x161 + 9x160 = 295310
* D3016 = 13x162 + 3x161 + 0x160 = 337610 = 42310 + 295310

Below is the 3 step math process for subtracting D30 - B89.

16

C 2 16

~~D~~ ~~3~~ 0

- B 8 9

-----

1 A 7

Step 3 calulates C-B

***16***

***C*** 2 16

***~~D~~ ~~3~~*** 0

- B 8 9

-----

A 7

Step 2 redistributes data to calculate 2-8

***2 16***

D ***~~3~~*** 0

- B 8 9

-----

7

Step 1 redistributes data to calculate 0-9

Remember that *all electronic circuits are really binary circuits*. Hex is only shorthand used by humans to represent large binary values.

Finite Precision Arithmetic and Overflows

If you are working with a pencil and paper and generate a result that needs additional digit positions, then you can just create those new positions. This is called *infinite precision arithmetic*.

When you build a real machine, such as a calculator or computer, you must fix the number of digits that can be used (e.g. the size of the calculator’s display). This is called *finite precision arithmetic*. If you now generate a result that needs more digits, you are in trouble! This is called an *overflow*.

*An overflow is the generation of a result*

*that does not fit in the data field being used*

The implication of an overflow is that the result calculated is wrong and should not be used in further operations.

For unsigned numbers, an overflow occurs when you:

1. add two numbers and the result is too large to fit in the assigned data field
2. subtract a larger number from a smaller number attempting to generate a negative number

The way computers handle unsigned overflows is very different from the way calculators handle overflows.

If you have a calculator with an 8 digit capacity and you generate a 9 digit answer then the calculator stops and displays the word *ERROR* in the display. You usually need to clear the calculator and start again.

Overflows in computers are handled as follows.

**Addition:** In a computer, if you generate an unsigned overflow on an addition, the system completes the execution of the instruction. The result generated is the correct low order part of the desired result. There will also be a flag, called the *carry flag*, set to indicate that an overflow occurred and the result is wrong.

If we add the 4-bit (1 hex digit) unsigned binary numbers (11112 + 00102) we expect to get 100012. Since this is a 5 bit result, and does not fit in a 4-bit binary (1 hex digit) field, instead we get 00012 and an overflow indication.

Note the correct result should be 100012 and the actual result is the correct low order four bits.

We show 3 bases: binary, hex and decimal

Carries

**The carry out of the most significant digit, shown as *1*, signifies an overflow.**

***This rule is true for all bases if the numbers are unsigned.***

Binary Hex Decimal

***1***11 ***1***

11112 --> F16 --> 1510

+ 00102 --> 216 --> 210

----- --- ---

00012 116 110

The reason the hardware works this way is to allow the programmer to handle the overflow condition and possibly write software to support large variable length data sizes such as 100 digit numbers.

**Subtraction:** In a computer, if you generate an unsigned overflow on a subtraction, the system completes the execution of the instruction. The result is generated as if the system could perform a borrow (redistribute data) into the most significant digit. This result will be the correct low order part of the desired result. The same carry flag will be set to indicate that a borrow was assumed to occur into the most significant digit and thus an overflow occurred and the result is wrong.

For example, if we subtract 410 - 710 the correct result of -310 is not a valid unsigned number.

In binary this would look like 01002 - 01112.

The result calculated is 11012 and an overflow indication.

Simulated borrow into the most significant digit

01002 = 410 Operation done as this ***1***01002

- 01112 = - 710 - 01112

---- -- ----

11012 1310 11012

The result looks like 13 instead of -3, which of course is wrong!

It is wrong because -3 does not exist in the world of unsigned numbers.

If we did this in hex, it looks like:

In decimal, the top value becomes 16 + 4 =20

The bottom value is 7

20 - 7 = 13 in decimal or D in hex.

416 Operation done as ***1*** 416

- 716 -716

- -

D16

***A borrow into the most significant digit signifies an unsigned overflow***

***when subtracting. This rule is true for any base.***

Note that ***if*** the top number were 5 digits and the most significant digit was 1 then the correct result would be 11012. So the actual result shown above is the correct low order four bits.

101002 = 2010

- 001112 = - 710

----- --

011012 1310

Again, the reason the hardware works this way is to allow the programmer to handle the overflow condition and possibly write software to support large variable length data sizes.

Precision versus Accuracy

Stop for a moment and think about the two words ***precision*** and ***accuracy***.

Do you know what they mean? Do you know how they differ?

A computer performs this calculation: 1.00000000 + 2.00000000 = 4.78438721

* Is the result 4.78438721 precise?
* Is the result 4.78438721 accurate?

Precision and accuracy are quite different.

*Precision is a measure of the number of digits used to perform the calculation*.

*Accuracy is the measure of how close your result is to the correct value*.

Given this calculation: 1.00000000 + 2.00000000 = 4.78438721

* The result 4.78438721 is very precise. It has lots of digits.
* The result 4.78438721 is not accurate. The result is wrong.

# B - Signed Positional Numbers

Introduction and Goal

Modern computers do not perform subtraction the way we did it for unsigned positional number systems.

Instead, they use complement arithmetic and perform A - B as A + (-B).

This process allows the CPU to use the same adder circuit for both addition and subtraction. A separate subtraction circuit is not needed, thus saving hardware.

However, in order for this to work, we must ***select a representation for signed numbers that uses the exact same rule for addition that is used by unsigned PNS***.

When we find such a number system then we can then have one single machine instruction that works on both unsigned and signed numbers. We can have an instruction such as *ADD X,Y* that correctly adds the variables X and Y regardless of whether X and Y are unsigned or signed values.

The Bad News

There are many ways to represent signed numbers and not all of them have the desired characteristic of using the same rule for addition that is used by unsigned PNS.

For example, the everyday decimal number system we use, called signed magnitude, does not have this feature.

We create a signed magnitude number by placing a + or - sign in front of an unsigned number. For example, 10 becomes +10 or -10. If we try to add signed magnitude numbers using the rule for adding unsigned numbers, we quickly encounter major problems.

unsigned signed magnitude

10 +10

+ 05 + -05

-- --- We don’t know how to handle signs and the

15 ?15 <- magnitude is wrong.

We can expand the rule for adding digits in unsigned PNS to handle signed magnitude numbers. These are the results we are looking for.

+10 +10 -10 -10

+05 -05 +05 -05

-- -- -- --

+15 +05 -05 -15

This leads to the rule for addition of signed magnitude numbers:

If the signs are the same

then add the digits using rule for unsigned PNS

and the result has the sign of the input numbers

Else if the magnitudes are different

then subtract the digits of the smaller number

from the digits of the larger number

and the result has the sign of the larger input number

Else if the magnitudes are the same

then the result is zero and the sign is either + or -

This is an extension of the basic rule for addition in unsigned PNS. We could use it, but that would mean different rules for unsigned and signed numbers and separate computer instructions for unsigned and signed numbers. The rule is also pretty complicated and harder to convert into hardware logic.

Instead, lets search for a signed number system that can use the same rule for addition as is used for unsigned numbers.

A Tale of Three Signed Number Systems

We will look at three different ways to represent signed binary numbers.

These three different number systems are named:

1. signed magnitude
2. one’s complement
3. two’s complement

For each of these number systems we look at three characteristics:

1. What is rule for adding bits?
2. How many ways can you represent zero?
3. Is there the same quantity of positive and negative values?

We will look at these three number systems in binary. We will use 4 bit binary numbers. The left most bit will be the sign bit.

* 0 sign bit indicates a positive number
* 1 sign bit indicates a negative number

We will see that all 3 number systems use the same representation for their positive values and these positive values look like unsigned values.

*It will be negative values that differ in these 3 number systems.*

Signed Magnitude

In signed magnitude, you negate a value by inverting the sign bit.

This process works both ways. Invert the sign of a positive number and it becomes negative. Invert the sign of a negative number and it becomes positive. Here are some signed magnitude values.

bin dec dec bin

------------------------

0000 +0 -0 1000

0001 +1 -1 1001

0010 +2 -2 1010

0011 +3 -3 1011

0100 +4 -4 1100

First let us see if we can add bits using the rule for unsigned PNS.

If you add +1 and -1 using the unsigned rule you get:

0001

+ 1001

----

1010 = -2 This is clearly not the right answer.

How does signed magnitude match our three characteristics?

1. It needs an extended addition rule to work properly. For use with binary, the *sign is the left most bit* and the *digits are all the remaining bits*.

If the signs are the same

then add the digits using rule for unsigned PNS

and the result has the sign of the input numbers

Else if the magnitudes are different

then subtract the digits of the smaller number

from the digits of the larger number

and the result has the sign of the larger input number

Else if the magnitudes are the same

then the result is zero and the sign is either + or -

2. It has two values for zero (+0 and -0).

3. It has the same quantity of positive and negative values.

Primarily because it requires an extension to the rule for addition, signed magnitude is not used by computer manufacturers.

Section 2D shows how to detect an overflow when adding signed magnitude numbers.

One’s Complement

In one’s complement, you negate a value by inverting all the bits.

This process works both ways. Invert all the bits of a positive number and it becomes negative. Invert all the bits of a negative number and it becomes positive. Here are some one's complement values.

bin dec dec bin

------------------------

0000 +0 -0 1111

0001 +1 -1 1110

0010 +2 -2 1101

0011 +3 -3 1100

0100 +4 -4 1011

If you add +1 and -1 using the unsigned rule you get:

0001

+ 1110

----

1111 = -0

The above example works correctly, but now try -1 + -1

1110

+ 1110

----

1100 = -3 This is wrong.

You cannot add one's complement numbers using the standard rule for unsigned PNS. However, you can extend the unsigned PNS rule to work for one’s complement.

The extended rule for one's complement addition is called *end around carry*.

Its derivation is shown in section D of these notes.

Add the bits using the rule for unsigned PNS. If there is a carry out of the most significant bit then add it into the least significant bit.

Lets apply it to adding -1 + -1

111 <--- carry

1110 <--- -1

+ 1110 <--- -1

----

1100

1

----

1101 = -2 which is correct.

How does one's complement match our three characteristics.

1. It needs an extended addition rule to add. This extension is *end around carry.*

2. It has two values for zero (+0 and -0).

3. It has the same quantity of positive and negative values.

Computers have been built using one’s complement: DEC PDP-1 in 1960, the first mini computer and the Univac 1100, in 1962, a popular mainframe.

Section 2D shows how to detect an overflow when adding one’s complement numbers.

Two’s Complement

The definition of two’s complement is: **N + TC(N) = 2d**

N = Any number.

TC(N) = The complement of N which is also known as -N.

d = The number of binary digits (bits) being used.

The complement is calculated by rewriting the equation and only saving the result to d digits and ignoring carries out of the sign position.

TC(N) = 2d - N = 100002 - N if d = 4

So, for example, the two's complement of 00012 is 100002 - 00012 = 11112

If you complement positive zero you get positive zero. The two's complement of 00002 is 100002 - 00002 = 00002 (remember we only save the low 4 bits).

There is no bit pattern that represents negative zero. It does not exist.

Here are some more two's complement values.

bin dec dec bin

------------------------

0000 +0 -0 There is no negative zero

0001 +1 -1 1111

0010 +2 -2 1110

0011 +3 -3 1101

0100 +4 -4 1100

If you add +1 and -1 using the rule for unsigned PNS then you get the correct answer of +0.

0001

+ 1111

----

0000 = +0

How does two's complement match our three characteristics.

1. It uses the same rule for addition as unsigned numbers.

2. It has one value for zero.

3. It has the same quantity of positive and negative values

*but* they are not symmetrical around zero. There is always

one more negative non-zero value than positive non-zero value.

Two's complement is the number system selected for signed numbers by modern computer manufacturers.

There is a shortcut for calculating two’s complement that does not require subtraction. The shortcut is to flip the bits and add 1.

The shortcut is: TC(N) = flip\_bits + 1

Most people use this method. Of course it works both ways. Complement a positive number and you get its negative value. Complement a negative number and you get its positive value.

Sometimes viewing a number system pictorially (a number wheel) helps you understand what is happening.

To simplify the picture we only use 3 binary digits instead of 4 binary digits.

The binary values are shown on the outside of the wheel and the decimal values are shown on the inside of the wheel.

0002 Unsigned number wheel with 3 digits

010  When 0012 is added to 1112 the result is

111 7 1 001 0002 and a carry. The carry indicates

an unsigned overflow occurred and that

the result 0002 is wrong.

110 6 2 010

101 5 3 011

4

100

0002 Signed number wheel with 3 digits

+010 When 0012 is added to 1112 the result is

111 -1 +1 001 0002 and a carry. The result of 0002 is

correct and the carry does **NOT** indicate

a signed overflow occurred.

110 -2 +2 010

An overflow occurs when 0012 is added to

0112 yielding 1002. This is (+110) + (+310)

101 -3 +3 011 yielding (-410) which is wrong.

-4

Note there is a -4 but not a +4. With

100 3 binary digits the range of decimal

values is -1 to -4 and +0 to +3.

*There is one more non-zero negative*

*number than non-zero positive number*.

A binary two's complement overflow can be detected in two ways.

1. If you add two numbers with like signs and the result has a different sign then an overflow occurred.

You cannot have an overflow when adding unlike signs since the magnitude of the result cannot be bigger than the magnitude of the inputs, which were valid.

1. If the Carry-Out of the sign position is not equal to the Carry-In to the sign position then an overflow occurred.

*Both rules always work and both rules always give the same indication of whether a signed overflow occurred*.

Lets look at an example of adding 3 bit two's complement numbers.

We add decimal +2 to itself. We would like to get +4 as the result, but +4 is outside the range of valid answers for 3 bit two's complement numbers. That range, from the previous page, is -4 to -1 and +0 to +3.

Decimal +2 has the bit pattern 010.

sign position

carry\_out carry\_in

0 ≠ 1

0102 positive

+ 0102 positive

---

100 negative

*Rule 2 catches the error*.

The carry into the sign position is 1 and the carry out of the sign position is 0. Since these differ an overflow has occurred.

*Rule 1 catches the error*.

We added two positive numbers but the result is negative.

How can I do hex two's complement problems

Remember, there is not any hex hardware. Hardware is binary. Although numbers in the computer are binary, humans often write those binary numbers in hex format.

You can do arithmetic in hex or you can convert it to binary and do the work. Add these two's complement hex numbers and indicate if a signed overflow occurred.

***01***11

5016 ---> 0101 00002 This example converts to binary

+ 7016 ---> 0111 00002 and then does the addition.

-- ---------

C016 <--- 1100 00002

Did a signed overflow occur? Yes. Both rules show an overflow occurred.

* The Carry\_in to the sign position was 1 and the Carry\_out of the sign position was 0 so a signed overflow occurred.
* Both input values are positive but the result is negative, so a signed overflow occurred (5016 = 8010 and 7016 = 11210 so the correct result should be 19210 which does not fit in a signed byte).

As you develop experience, you can do the addition without converting.

5016

+ 7016

--

C016

The result is straightforward: 50 + 70 = C0.

But how can you tell if a signed overflow occurred?

We will use the rule that states an overflow occurred when you add two numbers with the same sign and the result has a different sign.

You can tell if a hex two’s complement number such as 50, 70 or C0 is positive or negative. Convert just the left most digit to binary, so 516 = 01012. Then look at the left most bit. If it is 0 then the number is positive, if it is 1 then the number is negative. So 50 is positive, 70 is positive, C0 is negative.

An even faster way is to recognize that

* hex digits 0-7 all have 0 as the left most bit so they are all positive
* hex digits 8-F all have 1 as the left most bit so they are all negative

This means that 50 + 70 = C0 and we have the sum of two positive numbers being negative so a signed overflow occurred.

5016 <--- positive

+ 7016 <--- + positive

--

C016 <--- negative result means there was a signed overflow

*Watch out for the following trap!*

If you use the carry-in/carry-out rule, it ***looks*** like the carry-in equals the carry-out and that means there should not be an overflow.

carry-out --> 0 0 <-- carry-in

5016

+ 7016

---

C016

What happened?

The problem is that the carry-in/carry-out rule is a binary rule. That rule looks at the carry-in and the carry-out of the binary sign position. In the example above, the carries are between hex digits and not the binary sign position.

If you convert to binary and do the math then the rule does work.

You can do hex two's complement subtraction the same ways.

Either convert to binary or work directly in hex.

First lets convert to binary and calculate A-B as A+(-B).

Subtract these two's complement hex numbers C0 - D0 and specify whether a signed overflow occurred.

***00***001 111

C016 ---> 1100 00002 ---> 1100 0000

- D016 ---> 1101 00002 ---> 0010 1111 flip the bits

1 and add 1

-- --------- ---------

F016 ---> 1111 00002 1111 0000

Did a signed overflow occur? No. Both rules show that to be the case.

* The Carry\_in to the sign position was 0 and the Carry\_out of the sign position was 0 so a signed overflow did not occur.
* When adding a positive and negative number, there never is an overflow.

We can verify our result by converting the numbers to decimal.

C016 = -6410 and D016 = -4810 so the correct answer be -1610 which is F016.

Now lets do the calculation directly in hex.

Step 1 Step 2

0-0=0 We cannot do C-D so we assume there is a

1 in the next significant position and

redistribute data. C16=1210 and D16=1310 so we

then calculate (1610+1210) - 1310 = 1510 = F16

***0*** ***16***

***~~1~~***

C016 C016 C 016

- D016 - D016 - D 016

-- -- ---

0 F 0

Did an overflow occur? C0 is negative and D0 is negative. The calculation of

negative - negative is equal to negative + (- negative) or negative + positive.

There is never an overlow when adding unlike signs.

From this logic we can even create a new rule.

*There is never an overlow when subtracting like signs*

*If you perform (negative - positive) then a positive result indicates an overflow*

*If you perform (positive - negative) then a negative result indicates an overflow*

A summary of why two’s complement is significant

1. Two’s complement was chosen as the way computers store signed numbers because then computers can use the same electronic circuits, and machine instructions, for adding unsigned positional numbers and signed two’s complement numbers.
2. The hardware ***cannot*** tell if a bit pattern represents an unsigned positional number or a signed two’s complement number. It is the responsibility of the ***programmer*** to keep track whether a variable is unsigned or signed.

For example, the binary number 11111111 can be either decimal 255 as an unsigned value or -1 decimal as a signed number. The hardware cannot tell and does not know; the programmer must know.

1. The way that one detects unsigned and signed overflows differs. For example if you add 11111111 + 00000001 the expected results differ.

Actual Expected Expected

Binary Unsigned Signed

Carries 1 = 1111 111

1111 1111 255 -1

0000 0001 1 +1

--------- --- -

0000 0000 256 +0

The ***expected unsigned*** result is 256. The actual result was 0. The carry out of the most significant position signifies that the value of 0 is not correct, an overflow occurred.

The ***expected signed*** result is +0. The actual result was +0. The fact that the Carry-In to the sign position equals the Carry-Out of the sign position signifies that +0 is the correct answer.

* Subtraction can always be performed in *either* of two ways.

1. Pure subtraction: A - B

2. Adding the complement: A + B + 1 (B means invert the bits of B)

For example: 00000001 - 11111111 can be done either way.

1. Pure subtraction

1

1111 111

~~1 0000 000~~1 A

- 1111 1111 - B

0000 0010 <--- Answer

*You aways get the same*

2. Adding of the two's complement *numerical result*

0000 0001 A

0000 0000 + B inverted

+ 1 + 1

0000 0010 <--- Answer

You always get the ***SAME NUMERICAL RESULT***.

Most people would rather do the adding of the complement instead of the pure subtraction. Since you get the same result either way and the hardware does not know if the numbers are signed or unsigned you can even subtract unsigned numbers using this method.

***However, the indication of an overflow is different depending on how the programmer is interpreting these values***.

1. If these are unsigned values then we have 1 minus 255 and the borrow into the most significant bit signifies the answer of 2 is wrong.
2. If these are signed values then we have (+1) minus (-1) and the Carry-In to the sign equal to the Carry-Out of the sign (both 0) signifies the answer of +2 is correct.

This last example should answer the burning question:

*How do you detect an overflow on a binary two’s complement subtraction?*

The answer is that you perform the subtraction by adding the complement and then look at the Carry-In and the Carry-Out of the sign position. If they are equal then there was not an overflow.

Warning ... adding the complement is done in a single step.

Okay: A - B = A + B + 1 Not okay: A - B ≠ A + ( B + 1)

The use of the one's complement and a low order one instead of the

two's complement of the second operand is necessary for the proper recognition of overflow when subtracting the most negative number.

# C - Number Systems Review

Unsigned Positional Number Systen Review

1. Positional number systems are the foundation of mathematics.
2. Each positional number system has a name, a base, and a set of digits that range from 0 through the base-1.
3. Positional number systems allow you to perform operations directly on the digits.
4. In binary, numbers range from 0 to 2d - 1, where d = the number of binary digits (bits) being used.
5. The rule for adding digits is:

If the sum of the digits is less than the base

then the result is represented by a single digit

Else

the sum is reduced by the base and represented by a single

digit and a carry.

1. An overflow is the generation of a result that does not fit in the number of digits available. It means the answer you calculated is wrong.

***An unsigned overflow is detected by a carry out of the most significant digit on addition, or a borrow into the most significant digit on subtraction.***

The rule above is true for any base.

Signed Positional Number Systems Review

There are multiple ways to represent signed numbers. We looked at three:

1. Signed Magnitude
2. One’s Complement
3. Two’s Complement

They all have these characteristics:

1. The left most bit is the sign bit where 0 indicates a positive number and 1 indicates a negative number.
2. The positive numbers and unsigned numbers all look identical.
3. It is the representation of *negative values* that differ.

The three characteristics of interest for each signed number system are:

1. Can you perform arithmetic operations directly on the bits just as if they were unsigned numbers?
2. How many representations of zero are there in the number system?
3. How many non-zero positive and non-zero negative digits are there in the number system?

Signed Magnitude number system:

1. To negate a number you invert the sign bit.
2. Can you do arithmetic operations directly on the data?

No, you need a more complex algorithm.

1. There are two representations of zero.
2. There are the same number of non-zero positive and negative digits.
3. Example: 0010 = +2 1010 = -2

One’s Complement number system:

1. To negate a number you invert all the bits.
2. Can you do arithmetic operations directly on the data?

No, you need a more complex algorithm

1. There are two representations of zero.
2. There are the same number of non-zero positive and negative digits.
3. Example: 0010 = +2 1101 = -2

Two’s Complement number system:

1. To negate a number you invert all the bits and add 1.
2. Can you do arithmetic operations directly on the data? Yes!
3. There is one representation of zero.
4. There are the same number of positive and negative digits, however, they are not symmetrical around zero.
5. Example: 0010 = +2 1110 = -2

Two’s Complement Number Systems Review

1. Two's complement extends unsigned positional number systems to handle negative numbers for most modern computers.
2. It is chosen because it uses the same rule for adding digits as unsigned positional numbers. The sign bit is treated just like any other digit.
3. Half the bit patterns are positive (those with the sign bit of 0) and half are negative (those with the sign bit of 1). Because there is only a positive zero, there will be one more non-zero negative digit than non-zero positive digit; this will always be the most negative number possible.
4. The definition of two's complement is: N + TC(N) = 2d
5. The two's complement of a number can be calculated in different ways.

TC(N) = 2d - N

TC(N) = flip the bits and add 1

1. Complementing a positive number yields a negative number (except zero).

* Complementing a negative number yields a positive number (except the most negative number).

1. An overflow on addition (incorrect result) can be detected in two ways:

If you add two numbers with like signs but the result has the opposite sign then an overflow occurred.

If the carry out of the sign bit differs from the carry into the sign bit then an overflow occurred.

1. Complementing the most negative number will result in an overflow.

Hints on converting signed numbers to decimal

We have discussed three systems used to represent signed binary numbers:

* Signed Magnitude
* One's Complement
* Two's Complement

The general rule to convert *any* signed binary numbers to decimal is:

1. If the number is positive, the left most bit = 0, then just use expansion of powers to determine the decimal magnitude of the binary number. The sign of that magnitude is +.
2. If the number is negative, the left most bit = 1, then apply the rules of the signed number system being used to convert the binary number to positive. Complementing the negative number makes it positive. Now use rule #1 above to determine the magnitude of the number being converted. The sign of that magnitude is -.

Examples.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System** | **Sample** | **Pos/Neg**  **Number** | **Conversion Rule** | **Result** |
| Signed Magnitude | 01010000 | Positive | 1. Expand powers. 2. Add a + sign. | 8010  +8010 |
| Signed Magnitude | 11010000 | Negative | 1. Invert sign. 2. Expand powers. 3. Add a - sign. | 010100002  8010  -8010 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System** | **Sample** | **Pos/Neg**  **Number** | **Conversion Rule** | **Result** |
| One's complement | 01010000 | Positive | 1. Expand powers. 2. Add a + sign. | 8010  +8010 |
| One's complement | 11010000 | Negative | 1. Invert all bits. 2. Expand powers. 3. Add a - sign. | 001011112  4710  -4710 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System** | **Sample** | **Pos/Neg**  **Number** | **Conversion Rule** | **Result** |
| Two's complement | 01010000 | Positive | 1. Expand powers. 2. Add a + sign. | 8010  +8010 |
| Two's complement | 11010000 | Negative | 1. Flip bits + 1. 2. Expand powers. 3. Add a - sign. | 001100002  4810  -4810 |

# D - Additional Thoughts On Number Systems

One of the greatest assets you can bring to an employer is the ability to take the knowledge you have and apply it toward the solution of *new problems*.

The skill of solving new problems comes through practice.

Toward that end, we continually try to develop problems and questions that you have not seen before, but which can be solved by applying both the basic information you have learned and logic.

This section provides some of these new problems for you to solve.

Although we provide solutions, you should try to derive the solutions yourself.

Why does flipping the bits + 1 create the two's complement

Notation: N A binary number in the two's complement number system.

TC(N) The two's complement of N that is equal to -N.

d The number of binary digits being used.

Given:

The definition of two's complement is: N + TC(N) = 2d

You calculate the two's complement by re-arranging terms: TC(N) = 2d - N

We use the notation N to represent the result of flipping the bits of N.

1. Any binary number composed of ***d*** digits will be a string of 1s and 0s.
2. If you flip all the bits of that number to calculate N then every digit that was a 1 will now be a 0 and every digit that was a 0 will be a 1.

For example: N=10001111 N=01110000

1. If you add N + N you will always get a string of 1 bits that is d digits long. That is ***N + N = a string of 1 bits***

Using the value above: 10001111

+ 01110000

11111111

1. 2d is defined as a 1 bit followed by ***d*** 0 bits.

For example: d=2 22 = 100 <-- 2 zeroes

d=3 23 = 1000 <-- 3 zeroes

d=4 24 = 10000 <-- 4 zeroes

1. Subtracting 1 from 2d will create a string of ***d*** 1 bits.

That is ***2d - 1 is a string of 1 bits***.

For example: d=2 22 - 1 = 11 <-- 2 1 bits

d=3 23 - 1 = 111 <-- 3 1 bits

d=4 24 - 1 = 1111 <-- 4 1 bits

1. Combining the results from #3 and #5 we get: ***N + N = 2d - 1***
2. Rearranging terms we get: ***N + 1 = 2d - N***
3. Therefore, flipping the bits + 1 yields the two's complement of a number.

Why does end around carry work for One's Complement

It is possible to perform addition on One's Complement data by making a simple modification to the rule for addition of unsigned positional numbers.

Any carry out of the most significant digit is added into the least significant digit (LSD). This is called end around carry.

For example, using 4 bits:

1 1

+5 ---> 0101

+ -3 ---> 1100

+2 0001

1 end around carry is added to LSD

0010 = +2

Why does this work?

1. With any reasonable number system, if you count sequentially by adding the number 1, you would expect to see the following progression:

... -3, -2, -1, 0, +1, +2, +3 ...

1. However, with One's Complement you get this progression:

... -3, -2, -1, -0, +0, +1, +2, +3 ...

That is, when add 1 to -0 you do not get the expected result of +1. Instead you get +0. This is like saying that if you have no money and someone gives you a dollar then you still have no money. It is not logical.

1. To fix this illogical fact, we want to skip over the second appearance of zero when counting by 1. Also note that when you go from -0 to +0 by adding 1 you get a carry out of the most significant bit.

You get a carry here when you add 1 to -0

1111 + 1 = 0000 and a carry

0000 = +0

1111 = -0 0001 = +1

1110 = -1 0010 = +2

1. The extension for one's complement just uses the carry obtained when adding 1 to -0 to ***skip over +0*** and go to +1. Now the number system works logically.
2. Can you ever get two end around carries in a calculation? The answer is no. Here is a simple explanation using 4 bits. To get a second end around carry, the result of the original addition would have to generate a carry (the first end around carry) and a numerical value of 1111. Then the first end around carry when added to 1111 would generate the second end around carry.

However, a result of a carry and 1111 means that the original addition generated the value 11111. This cannot happen. The largest result will be generated when you add the largest 4-bit values 1111 + 1111 = 11110. So it is not possible to generate a carry and a result of 1111.

Why does the carry-in / carry-out rule detect a two's complement overflow

Rule: A two’s complement overflow occurred if the carry out of the sign bit

is not equal to the carry into the sign bit.

In reviewing textbooks, I have seen a number of textbooks that that have one of these problems.

* They incorrectly explain how to detect a two's complement overflow.
* They correctly provide one of the ways to detect a two's complement overflow but do not explain why it works.
* They have no reference at all to how the overflow flag is set.

Here are some quotes from textbooks.

* *Correct but without explanation*. The hardware detects a signed overflow by comparing the carry into the sign bit with the carry out of the sign bit. If they are different then an overflow has occurred and OF is set to 1. If they are the same then no overflow occurred and OF is set to 0.
* *Correct but without enough detail*. If an arithmetic operation produces a result that exceeds the capacity of the register then the OF flag is set.
* *Incorrect*. When adding integers of the same sign, a carry into the sign position will result in the OF flag being set.
* *Incorrect*. The OF flag indicates a carry into and out of the sign bit following a signed arithmetic operation.

It is not surprising that new assembler students will be confused, if the people writing the texts do not clearly present the information.

Therefore, we present the derivation of the rule that describes how to detect a two's complement overflow.

First, the logic of the situation. We will use three bit numbers for examples.

0002

+010

111 -1 +1 001

110 -2 +2 010

101 -3 +3 011

-4

100

1. Can two positive numbers be added and have the result *wrong* and *positive*?

That is, can we ever wrap around the wheel and have the sum of two positive numbers be a smaller positive number than either of the input numbers? Lets look at the worst case, where we add the largest positive number to itself. This would be decimal (+3) + (+3) or binary (011) + (011). The result is binary (110). That does *not* wrap around the wheel. Instead it generates a result that is in the negative range of numbers. We conclude that if we add two positive numbers and the result exceeds the largest valid positive number then that result will be *wrong* and will appear to be *negative*.

1. Can two negative numbers be added and have the result be *wrong* and *negative*?

That is, can we ever wrap around the wheel and have the sum of two negative numbers be a less negative number than either of the input numbers? Lets look at the worst case, where we add the most negative number to itself. This would be decimal (-4) + (-4) or binary (100) + (100). The result is binary (000). That does *not* wrap around the wheel. Instead it generates a result that is in the positive range of numbers. We conclude that if we add two negative numbers and the result exceeds the most negative number then that result will be *wrong* and will appear to be a *positive* number.

1. Can a positive and a negative number be added and get a result that is wrong.

Lets add all four boundary conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| most positive  + most negative | most positive  + least negative | least positive  + most negative | least positive  + least negative |
| (+3) + (-4) = -1 | (+3) + (-1) = +2 | (+0) + (-4) = -4 | (+0) + (-1) = -1 |
| result is okay | result is okay | result is okay | result is okay |

All the operations with the boundary values are correct. Any non-boundary values will be less demanding and thus will also be correct. Therefore, if we add a positive number and a negative number then an overflow cannot occur.

This yields our basic logical rule for detecting overflows.

If you add two numbers with like signs and the sign of the result is not the same as the original signs then a signed overflow occurred. If you add two numbers with unlike signs the result will always be correct.

Now we expand our analysis to show that the rule that uses the carry into the sign position (CI) and the carry out of the sign position (CO) will detect signed overflows consistent with our logical rule previously derived.

If the CO = CI then a signed overflow did not occur.

If the CO ≠ CI then a signed overflow did occur.

**Case 1.** If we add two positive numbers then we have the following situation.

sign position

0 x x x ...

+ 0 x x x ...

The carry out of the sign position will always be zero. Why is that true?

1A. If the carry into the sign position is zero then we have the following.

CO=0 0=CI

0 x x x ...

+ 0 x x x ...

0

1B. If the carry into the sign position is one then we have the following.

CO=0 1=CI

0 x x x ...

+ 0 x x x ...

1

* In 1A two positive numbers were added and got a correct positive result. How do we know the result is correct? Remember, the logical rule previously derived specifies that if you add numbers with like signs and sign of the result has that same sign then the result is correct. Note that in this situation the CO = CI.
* In 1B two positive numbers were added and got an incorrect negative result. In this situation the CO ≠ CI.

**Case 2**. If we add two negative numbers then we have the following situation.

sign position

1 x x x ...

+ 1 x x x ...

The carry out of the sign position will always be one. Why is that true?

2A. If the carry into the sign position is zero then we have the following.

CO=1 0=CI

1 x x x ...

+ 1 x x x ...

0

2B. If the carry into the sign position is one then we have the following.

CO=1 1=CI

1 x x x ...

+ 1 x x x ...

1

* In 2A two negative numbers were added and we got an incorrect positive result. In this situation the CO ≠ CI.
* In 2 two negative numbers were added and got a correct negative result. In this situation the CO = CI.

**Case 3**. If we add a positive and a negative number then we have the following.

sign position

0 x x x ...

+ 1 x x x ...

The carry out of the sign position will always be the same as the carry into the sign position. Why is that true?

3A. If the carry into the sign position is zero then we have the following.

CO=0 0=CI

0 x x x ...

+ 1 x x x ...

1

3B. If the carry into the sign position is one then we have the following.

CO=1 1=CI

0 x x x ...

+ 1 x x x ...

0

In case 3A and 3B the CO = CI and we know from our logical rule that the results in case 3A and 3B are correct.

We have shown that the rule using the carry into the sign position and the carry out of the sign position generates the same result as our logical rule.

We derived and know the logical rule is correct. Thus this rule is also correct.

If the CO = CI then a signed overflow did not occur.

If the CO ≠ CI then a signed overflow did occur.

An alternate way to calculate two's complement

A new algorithm for calculating the two's complement is being proposed.

*Start at the left and invert all the bits until you reach the last 1 bit*

For example: Binary original number = 0 0 0 0 1 0 1 0 = +10 (dec)

invert leave

Binary two's complement= 1 1 1 1 0 1 1 0 = -10 (dec)

Answer these questions.

1. Why this is the same as *flipping the bits + 1* ?
2. There is one value for which the algorithm does not work. What is that value?
3. Modify the algorithm so that it will work for all inputs.

A. Why is this the same as *flipping the bits + 1*

Any number to be complemented by *flipping the bits and adding 1* can be broken into two parts.

* Part-A is all bits to the left of the right most 1 bit
* Part-B is the right most 1 bit and any trailing 0 bits.

Using the example above we have:

0 0 0 0 1 0 1 0

Part-A Part-B

Part-B , by its definition, will always consist of a 1 bit followed by a string of 0 bits. Anytime you invert a 1 bit followed by a string of 0 bits and add 1 you end up with the original value and there will not be a carry out of the left most 1 bit. For example: 1000 = 0111 + 1 = 1000

This means that when we complement by *flipping the bits + 1* initially Part-A and Part-B will be both be inverted.

0 0 0 0 1 0 1 0 1 1 1 1 0 1 0 1 + 1 1 1 1 1 0 1 1 0

Part-A Part-B Part-A Part-B Part-A Part-B

However, when we add the 1, Part-B will return to its original value while Part-A will remain inverted.

Well, this is the same as what the new algorithm does. It inverts all bits up to the right most 1 bit, and leaves the right most 1 bit and any trailing 0 bits alone. So the new algorithm is the same as *flipping the bits + 1*.

B. The algorithm, as stated, will not work if all the bits are zero bits.

C. *For any non zero number, start at the left and invert all the bits until you reach the last 1 bit*

Can you just look at negative number and tell its decimal value

Many people can just look at a positive two's complement value and quickly determine its value. For example 0101 is +5.

It is not so easy to just look at a negative two's complement number and determine its value. For example 1101 equals what decimal value?

We present a technique that will help.

1. Take the left most bit and assign to it, its negative positional value.
2. Treat all the other bits as positive and add their sum to the negative value.

So 1101 is calculated as follows.

= +5 -8 + 5 = -3 (flip the bits and add 1 to verify)

= -23 = -8

Why does this work?

Look at all the possible negative values using 4 bits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| binary | decimal | hi bit | low bits | hi bit neg. value | low bits sum | how far from most negative value of -8 are we ... |
| 1111 | -1 | 1 | 111 | -8 | 7 | 7 |
| 1110 | -2 | 1 | 110 | -8 | 6 | 6 |
| 1101 | -3 | 1 | 101 | -8 | 5 | 5 |
| 1100 | -4 | 1 | 100 | -8 | 4 | 4 |
| 1011 | -5 | 1 | 011 | -8 | 3 | 3 |
| 1010 | -6 | 1 | 010 | -8 | 2 | 2 |
| 1001 | -7 | 1 | 001 | -8 | 1 | 1 |
| 1000 | -8 | 1 | 000 | -8 | 0 | 0 |

You can see that the sum of the low bits is exactly how far away we are from the maximum negative value, so by adding that sum to the maximum negative value we get the decimal equivalent.

Example with 8 bits.

What is the decimal value of 10101010 ?

Answer: -128 + 32 + 8 + 2 = -128 + 42 = -86

Verify: Take 10101010 and flip bits + 1 = 01010101 + 1 = 01010110 = +86

So if the negative of this number is +86 then this number must be -86.

How can an overflow be detected in signed magnitude addition

This is the rule for adding in signed magnitude.

|  |  |
| --- | --- |
| Case | Operation |
| 1 | If the signs are the same  then add the digits using rule for unsigned PNS  and the result has the sign of the input numbers |
| 2 | Else if the magnitudes are different  then subtract the digits of the smaller number  from the digits of the larger number  and the result has the sign of the larger input number |
| 3 | Else if the magnitudes are the same  then the result is zero and the sign is either + or - |

In cases 2 and three, an overflow cannot occur because the magnitude of the result can never be larger than the inputs.

In case 1 there can be an overflow. The digit bits that are used in the addition are all the bits except the sign bit. If there is a carry out of these digit bits into the sign position then the result does not fit.

This leads to the following rule:

A carry out of the most significant digit bit signifies an overflow.

In the example below we add +7 and +3. The digit bits are shown in italic.

The carry out of the most significant digit,

into the sign position signifies an overflow.

**1**11

0111 +7

+ 0011 +3

010 <-- wrong magnitude for this addition

How can an overflow be detected in ones complement addition

This is the rule for adding in one's complement.

|  |
| --- |
| Add the bits using the rule for unsigned PNS. If there is a carry out of the most significant bit then add it into the least significant bit. |

We can use our basic logical rule for detecing an overflow.

*You cannot have an overflow when adding unlike signs since the magnitude of the result cannot be bigger than the magnitude of the inputs.*

*If you add two numbers with like signs and the result has a different sign then an overflow occurred.*

Example 1. There is not an end around carry

+7 0111 positive input

+3 + 0011 positive input

1010 negative result means an overflow

Example 2. There is an end around carry

1011

-4 1011 negative input

-4 +1011 negative input

0110

1

0111 positive result means an overflow

How can an overflow be detected when adding a two's complement signed number and an unsigned number.

* When adding unsigned binary numbers, an overflow is detected when there is a carry out of the most significant bit.
* When adding two's complement signed numbers, an overflow is detected when the carry out of the sign bit is not equal to the carry into the sign bit.

Develop a rule for detecting an overflow when you add a signed number to an unsigned number with the result being an unsigned number.

For this exercise, we assume the unsigned number is a byte (0...255), the signed number is a byte (-128...+127), and the result is an unsigned byte (0...255).

The technique will only use as test variables these bits:

* The carry flag.
* The Most Significant Bits (MSB) of the inputs.
* The Most Significant Bit (MSB) of the result.

There are many rules that can be developed. We start with the simplest, most intuitive, least mathematical rule.

Break the problem into 4 cases based upon the value of the MSB of the two inputs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MSB  Unsigned | MSB  Signed | Unsigned byte range | Signed byte range | Unsigned Result |
| 0 | 0 | 000 ... 127 | +0 ... +127 | 000 ... 254 no overflow possible |
| 1 | 1 | 128 ... 255 | -1 ... -128 | 000 ... 254 no overflow possible |
| 1 | 0 | 128 ... 255 | +0 ... +127 | overflow if result > 255 |
| 0 | 1 | 000 ... 127 | -1 ... -128 | overflow if result < 0 |

The first two cases show that if the MSB of the two inputs are the same then the result will always be in the valid range of an unsigned byte and thus an overflow is not possible.

In the third case, the MSB of the signed byte is zero. This means it looks like an unsigned value. We can, therefore, treat both numbers as unsigned values and use our standard rule for detecting overflows for unsigned numbers. A carry out of the MSB is an overflow.

In the fourth case, an overflow will occur if the magnitude of the negative number is greater than the magnitude of the unsigned number. In this case, we would be trying to generate a negative result (e.g. 17 + (-20) = -3). This can be detected by the MSB of the result becoming a 1.

These four cases lead to one of many possible rules:

If msb of signed equals msb of unsigned then an overflow never occurs

Else if msb signed is 0 and msb of the unsigned is 1

then a carry out of the msb of the result indicates an overflow occurred

Else if msb of the signed is 1 and msb of the unsigned is 0

then if msb of result is 1 an overflow occurred.

For those who want more math, we present the following, which is based on the previous table and uses these bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MSBU | left bit of unsigned input |  | MSBS | left bit of signed input |
| MSBR | left bit of unsigned result |  | - | don't care about this bit |
| CI | carry into MSB of calculation |  | CO | carry out of MSB of calculation |

Case 1. MSB of unsigned = 0 MSB of signed = 0

Unsigned Signed MSBU MSBS CI CO MSBR Notes

000...127 +0...+127 0 0 - 0 - Always ok

0xxxxxxx U

+ 0xxxxxxx S

--------

Case 2. MSB of unsigned = 1 MSB of signed = 1

Unsigned Signed MSBU MSBS CI CO MSBR Notes

128...255 -1...-128 1 1 - 1 - Always ok

1xxxxxxx U

+ 1xxxxxxx S

--------

Case 3. MSB of unsigned = 1 MSB of signed = 0

Unsigned Signed MSBU MSBS CI CO MSBR Notes

128...255 +0...+127 1 0 0 0 1 Ok: result < 256

128...255 +0...+127 1 0 1 1 0 Error: result > 255

The error occurs when the expected result is > 255. When this occurs, the carry out is 1 and the MSB of the result is set to 0. It appears that by adding a positive value to an unsigned number, the result got smaller which is not possible.

1xxxxxxx U

+ 0xxxxxxx S

--------

Case 4. MSB of unsigned 0 MSB of signed = 1

Unsigned Signed MSBU MSBS CI CO MSBR Notes

000...127 -1...-128 0 1 0 0 1 Error: result < 0

000...127 -1...-128 0 1 1 1 0 Ok: result > 0

The error occurs when the expected result is < 0. When this occurs, the carry out is 0 and the MSB of the result is set to 1. It appears that by adding a negative value to an unsigned number, the result got bigger which is not possible.

0xxxxxxx U

+ 1xxxxxxx S

--------

These cases can be converted into many different rules. Here are three.

1. If the MSB of the inputs are different and the MSB of the unsigned byte changes then an overflow occurred.
2. If the CO is not equal to the MSB of signed byte then overflow occurred.
3. If the signed number is positive then a carry out means overflow,

Else if the signed number is negative then no carry out means an overflow.

If those were not enough, we present one last method for detecting an overflow when adding a signed number to an unsigned number.

This is a table with all possible combinations of the left most bit of the inputs and results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MSB  Unsigned | MSB  Signed | MSB  Result |  |  |
| 0 | 0 | 0 | no overflow | We added 0...127 and 0...127 and the result was still 0...127 |
| 0 | 0 | 1 | no overflow | We added 0...127 and 0...127 and the result was 128...255 |
| 0 | 1 | 0 | no overflow | We added 0..127 and a negative number and the result was still 0...127 |
| 0 | 1 | 1 | **overflow** | **We added 0...127 and a negative number and the result was negative which cannot be represented as an unsigned value** |
| 1 | 0 | 0 | **overflow** | **We added 128...255 and 0...127 and the result exceeded 255 which can not be represented as an unsigned byte** |
| 1 | 0 | 1 | no overflow | We added 128...255 and 0...127 and the result was still 128...255 |
| 1 | 1 | 0 | no overflow | We added 128...255 and a negative number and the result was 0...127 |
| 1 | 1 | 1 | no overflow | We added 128...255 and a negative number and the result was still 128...255 |

You can use any Boolean function to detect an overflow. This is one.

If (MSBU != MSBS) and (MSBR != MSBU) then an overflow occurred.

How can you do math, such as add and subtract, in the various systems

You always have two choices.

1. Do the math using the rules of the specific system being used.
2. Convert to decimal, do the math, convert back to the system being used.

We present some examples.

Given the following 4 bit binary signed magnitude number: 1001

What is the 4 bit signed magnitude result of doubling that value.

This example converts to decimal, does the operation and converts result to the requested system.

* signed magnitude 1001 = -1
* doubling -1 yields -2
* -2 = 1010 as signed magnitude

Given the following 4 bit binary one's complement number: 1011

What is the one's complement result of dividing that value by 2.

This example converts to decimal, does the operation and converts result to the requested system.

* one's complement 1011 = -4
* -4/2 = -2
* -2 = 1101 as one's complement

Given these 4 bit binary signed magnitude numbers: 1011 and 0011

What is the 8 bit signed magnitude result of multiplying those two numbers.

* signed magnitude numbers

This example converts to decimal, does the operation and converts result to the requested system.

1011 \* 0011 = -3 \* +3 = -9

* -9 =10001001 as signed magnitude

Given these 4 bit binary one's complement numbers.

What is the 4 bit one's complement quotient resulting from the division.

11101011 / 0100 = ?

This example converts to decimal, does the operation and converts result to the requested system.

* 11101011 / 0100 = -20 / +4 = -5
* -5 = 1010 as one's complement

Given these 4 bit binary signed one's complement numbers: 1010 and 0110

What is the one's complement sum of the two numbers.

In this example we do the math in one's complement, using end around carry.

111

1010 1010 <--- Negative input

+ 0110 0110 <--- Positive input

0000

1

0001 <--- Result must be correct

We have rules for adding numbers in signed magnitude, ones complement and two’s complement. When doing subtraction we can always convert the subtraction to an addition of a complement. That means A – B can always be done as A + (-B)

Given these 4 bit binary signed magnitude numbers: 0100 and 1001.

What is the binary signed magnitude difference of the two numbers.

First we perform the operation by converting to decimal.

0100 +4 +4

- 1001 - -1 + +1

+5 = 0101 as signed magnitude

Alternatively we work directly in signed magnitude, doing the operation

A - B as A + (-B).

0100 0100 A No carry out of the most significant digit

- 1001 + 0001 + -B into the sign means the result

0101 is correct.

Given these 4 bit binary signed one’s complement numbers: 0100 and 1110.

What is the binary one’s complement difference of the two numbers

0100 +4 +4 0100 A Added two postive numbers

- 1110 - -1 + +1 + 0001 + -B and the result is positive

+5 +5 0101 and thus it is correct.

Additional insight

Given this two's complement hex number: ABCD

Is the number *ODD* or *EVEN*? Is the number positive or negative.

*EVEN* numbers are multiples of two. *ODD* numbers are not. It is not necessary to convert the whole number to decimal to see if it is a multiple of two. Only convert the low hex digit to binary ( D = 1101 ). The low order binary digit has the positional value 20, which is 1. All other positional values are multiples of two. So, if the low order bit is 1 then the number is *ODD*. Since in ABCD the low order bit is 1 then ABCD is *ODD*.

The sign bit is the left most bit. To determine the sign of ABCD, we only need to convert the left most hex digit to binary ( A = 1010 ). The left most bit is 1 so ABCD is negative.

The two's complement number ABCD16 = -2155510 which is negative and odd.

Given a 10 bit binary two's complement data field, what range of decimal numbers can be represented.

* 210 = 1024
* half the number of digits will be negative and half positive
* there is only a positive zero so there will be one more non-zero negative value than positive value
* the 512 negative values are -512 ... -1
* the 512 positive values are +0 ... +511
* the range of values is then -512 ... +511

Given these 20 digit hex unsigned numbers, will there be an unsigned overflow if they are added together.

A5E793F0A22B8C5D1F6D

+ 4BAF667DA0C5B3AA99F1

It is not always necessary to actually add the numbers to determine if an unsigned overflow will occur. An unsigned overflow occurs if there is a carry out of the left most bit. When adding two digits in any column, the only value for a carry is 0 or 1. This means that the worst case for the left most bit column will be A+4+1 where the 1 is a carry into that position. Well, A+4+1=F and there is no carry out, so there will not be an overflow.

If the numbers were two’s complement signed numbers, the top number is negative and the bottom number is positive. Thus no overflow can occur.

# E - Fractions

We start with a review of decimal fractions. Fractions are just an extension of positional number systems to include negative exponents. So the number 25.73 in its true positional format is ....

2x10+1 + 5x100 + 7x10-1 + 3x10-2 = 2x10 + 5x1 + 7/10 + 3/100 = 25.73

In order to perform arithmetic, it is necessary to align the decimal points.

So 10.8 + 7.32 must be written with the decimal points correctly lined up.

10.8 ---> fill the empty position with 0 ---> 10.80

+ 7.32 + 7.32

----- -----

18.12

Note that the decimal point is not physical but instead it is just a logical concept. It is the location where all digits to the left of the decimal point are integers and all digits to the right of the decimal point are fractional values.

You would get exactly the same numerical digits if you omit the decimal point and remember that the last two digits are fractions.

1080

+ 732

----

1812

This is really important, because many CPUs only have integer arithmetic hardware. For example, in the 8086, only fixed-point integer bytes and words can be processed. There were not enough electronic circuits on that chip to natively support floating point numbers.

So there are three basic ways to support fractions.

1. Add floating-point hardware in the form of a co-processor like the 8087.
2. Add floating-point hardware on the chip when there are enough circuits. This occurred when the 80486 was introduced.
3. Use integer arithmetic hardware and just assume that a hex point exists and all digits to the left are integers and all the digits to the right are fractions. On the 8086 one location to place this assumed hex point is between the high byte and low byte of a word. This will most easily allow one to perform addition, subtraction, multiplication and division. Unfortunately, only using one byte to store a fraction will severely limit the number of fractional digits that can be stored.

A one byte hex fraction is ?x16-1 + ?x16-2 where we fill in the digits. The smallest value would be .0116 = 0x16-1 + 1x16-2 = .0039062510 which means that we can only handle 2 to 3 decimal digits.

Conversion of fractions among decimal, binary and hex

Conversion from ***other bases to decimal*** uses expansion of powers.

All the math is done in base 10 since we are converting to decimal.

Binary to decimal

.1012 = 1 x 2-1 + 0 x 2-2 + 1 x 2-3

.1012 = 1 x .500 + 0 x .250 + 1 x .125

.1012 = .500 + .000 + .125

.1012 = .62510

If you are working with a fixed number of binary fractional digits, then you can speed up the process. With three binary fractional digits the low order position is 2-3 which is 1/8 so .1012 can be looked at as 5/8 which is .625

Hex to decimal

.A216 = 10 x 16-1 + 2 x 16-2

.A216 = 10 x .0625 + 2 x .00390625

.A216 = .625 + .0078125

.A216 = .632812510

In this example we are working with 2 hex digits. The low order position is 16-2 which is 1/256 so .A2 can be calculated as 162/256 which is .6328125

Another way to look at it is:

.A216 = 10 + 2 = 160 + 2 = 162 = .632812510

16 256 256 256 256

Conversion from ***decimal to other bases*** uses reduction of powers.

First we will do it the basic way. We find the missing binary digits.

.62510= \_? x 2-1 + \_? x 2-2 + \_? x 2-3 + ...

2-1 = .5000

2-2 = .2500

2-3 = .1250

2-4 = .0625

Step 1

Since .625 is more than .5000 the first digit is 1 and we have .125 left to convert.

.62510= 1 x 2-1 + \_\_ x 2-2 + \_\_ x 2-3 + ...

.500

.125

Step 2

Since .125 is less than .2500 the second digit is 0 and we have .125 left to convert.

.62510= 1 x 2-1 + 0 x 2-2 + \_\_ x 2-3 + ...

.500

.125

Step 3

Since .125 is equal to .1250 the third digit is 1 and we have .000 left to convert.

.62510= 1 x 2-1 + 0 x 2-2 + 1 x 2-3 + ...

.500

.125

.125

000

The final result is: .62510= .1012

Speeding up the conversion of ***decimal to other bases***. Follow this logic.

We know that .62510= .1012 from the previous page.

If we multiply both sides by 2 then the equation must still be true.

We do each multiplication in the correct base.

.62510 x 210 = 1.25010

.1012 x 22 just moves the binary point and the result is 1.012

So it must be true that: 1.25010 = 1.012

The integer parts must be same: 1 = 1 which you can see.

The fractions must be same: .25010 = .012 which you can show from the previous page

We can subtract the integer part from both sides yielding .25010 = .012

We can then repeat the process.

How does this help? *Each time we repeat the process, the binary digit directly to the right of the binary point is revealed to us. It is revealed as the integer part of the multiplication*.

Example: Convert .62510 to binary.

To do this multiply by 2. The integer part of the result is next binary digit. Remember the digit so you can build the binary number. Remove the integer part of the result. Repeat until the result is zero or until you have enough binary digits.

.625 x 2 = 1.250

.250 x 2 = 0.500

.500 x 2 = 1.000

These are the string of binary digits: .62510 = .1012

Example: Convert .3 to binary.

.3 x 2 = 0.6

At this point we will repeat the sequence of decimal values 6,2,4,8 and they yield the repeating binary sequence 1001

.6 x 2 = 1.2

.2 x 2 = 0.4

.4 x 2 = 0.8

.8 x 2 = 1.6

.6 x 2 = 1.2

.310 = .0 1001 1001 1001 ... 2

I used a calculator to convert .0 1001 1001 10012 to decimal and got .29993 so it is converging on .3 as its value.

Example: Convert .632812510 to hex.

.6328125 x 16 = 10.125

.125 x 16 = 2.0

.632812510 = A216

This is an example of using fractions in a computer.

Let us assume that we have a fractional decimal number such as 3.14159, which you may recognize as PI. We wish to convert PI to a hex fractional number so we can do arithmetic operations using PI on a computer such as the 8086.

We will use one byte for the integer and one byte for the fraction.

Using a one-byte hex fraction requires us to truncate decimal PI to 3.14.

We convert 3.1410 to hex in two steps.

* First convert the integer.
* Second convert the fraction.

Each part is stored as a byte, which is 2 hex digits.

1. The integer 3 is easy to convert. It becomes 0316.

2. To convert the decimal fraction we use the process previously shown.

.14 x 16 = 2.24

.24 x 16 = 3.84 This yields .23D16

.84 x 16 = 13.44

We stop, not because we have a perfect conversion, but because we can only store two hex digits in the byte we have allocated for the fraction.

We have converted three digits to allow us to decide whether to truncate or round up the result. Each hex digit can be 0-F and we round up if the last digit is 8-F. Since D is more than 8 we round the result to .2416

Combining the integer and fraction, 3.1410 = 03.2416 and if we store that in a hex word with the assumed hex point between the high and low byte we have PI=032416.