Chapter 5 | Arithmetic Unit

● Adder ● Binary Addition ● Half Adder ● Full Adder ● N-bit Adder ● Signed Binary Number ● Signed Magnitude Representation ● Addition and subtraction with signed magnitude representation ● Advantage and Disadvantage ● Complement Representation ● Diminished Radix Complement (N-1)'s Complement ● Radix Complement ● 2's Complement ● Advantages ● Subtractor ● Eliminating The Borrowing

CS102
Computer
Environment

Adder

Addition is the most basic of all arithmetic operations. As a matter of fact addition is the only operation the computer does. All other operations are reduced to addition. This is true for subtraction, multiplication, and division.

Computers store numbers in binary format. Therefore, we need to focus on building a binary adding machine, known as a binary adder.

A single binary adder adds 2 bits together. To add numbers more than 1 bit long several single binary adders are needed.

Binary Addition

Binary addition follows a similar procedure as decimal addition. When adding numbers we add each column separately. Then if the addition of the digits in the column generates a carry the carry is then added to the next column to the left.

Thus, a column (other than the first one) may receive a carry from the column to its right and can generate a carry as well.

The binary addition table showing the carry underlined:

+	0	1
0	00	01
1	01	10

Consider the second column in the following addition:

Example			
Carry	Carry out:1	Carry in :1	
	0	1	1
	0	1	1
Sum	1	1	0

It receives a carry in from column 1 and sends a carry out to column 3

The task at hand is to build a device that adds 2 bits together as shown in column 2 of the previous example. Since column 2 receives a carry in and generates a carry out, it represents the most general case for adding 2 bits. This device is called a “full adder”.

Half adder

In order to build the device one has to take a closer look to the table of addition. The binary addition table showing the carry underlined can be consider to be the merger of two separate tables, one representing the sum and the second representing the carry:

+	0	1
0	0	1
1	0	1

=

sum	0	1
0	0	1
1	1	0

+

carry	0	1
0	0	0
1	0	1

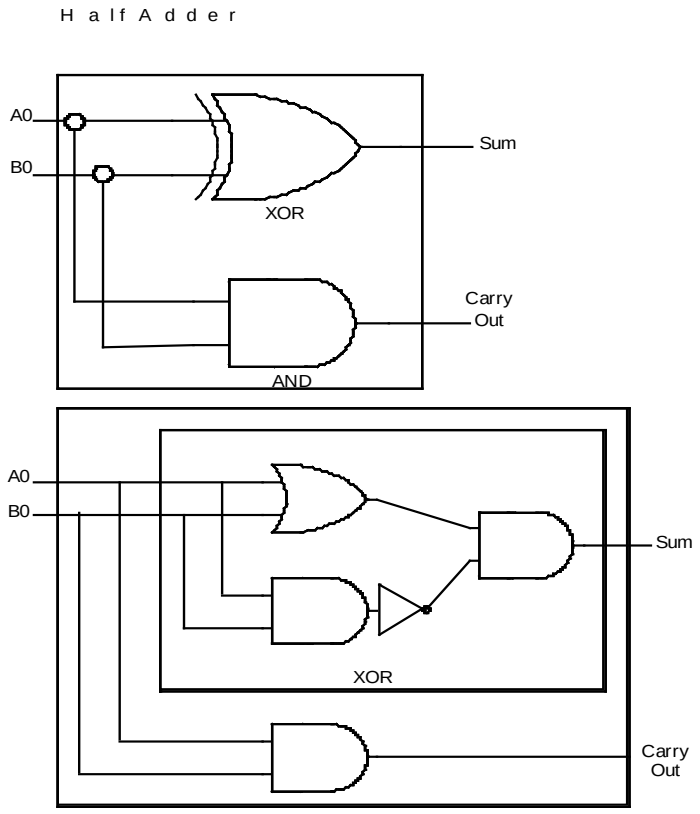
The sum table is identical to the one generated by an XOR gate.

XOR	0	1
0	0	1
1	1	0

And the carry table is identical to the one generated by an AND gate.

AND	0	1
0	0	0
1	0	1

The following diagram shows how the XOR and the AND gates are coupled together in order to generate the first version *(this version cannot do the entire job)* of the 2-bit adder we are trying to build. This early version is called a half adder:



The half adder has:

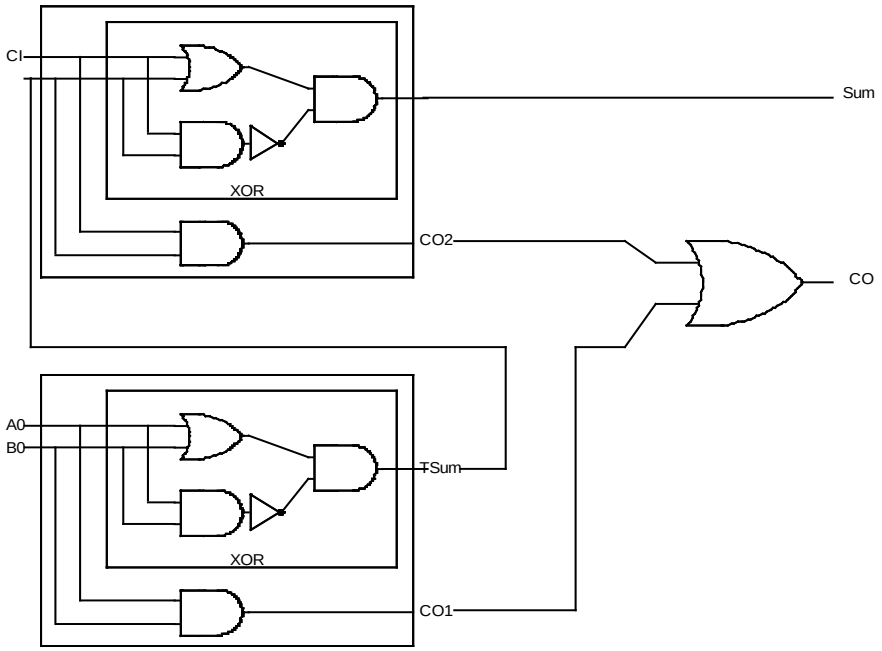
- 2 inputs (noted Bit1 and Bit2 in the picture above). These are the 2 bits to add.
- 2 outputs, one from the XOR which represents the sum and the other from the AND which represent the carry out.

The half adder gets its name from the fact that it does half the job (2/3d maybe more accurate!). The half adder does not have a way to accept any carry in bit resulting from a previous carry out. The solution to this problem is simple and is described in the next section.

Full adder

The solution to the problem is obtained by using two half adders connected together with an OR gate as shown in the diagram below:

Full Adder



The newly obtained device is called a full adder.

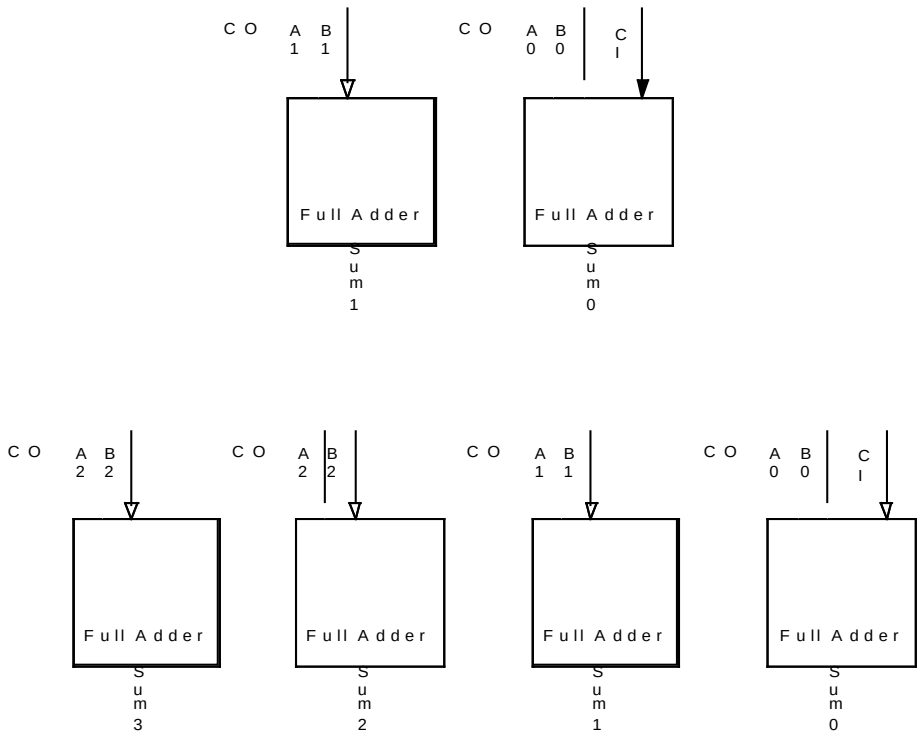
The truth table of the full adder is shown below:

A0	B0	TSum = ***A0 XOR B0***	Co1 = ***A0 AND B0***	CI	Sum = ***CI XOR TSum***	Co2 = ***CI AND TSum***	CO = Co1 OR Co2
0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0
1	0	1	0	0	1	0	0
1	1	0	1	0	0	0	1
0	0	0	0	1	1	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	0	1

Note: The temp Carry out 1 (tCo1) and the temp Carry out 2 (tCo2) are never at the same time high (value = 1). Therefore, an OR gate can be used to get the final Carry out (CO).

N-bit Adder

Several full adders can be combined together to add numbers that are more than one bit wide. The following diagrams represent a 2-bit adder and a 4-bit adder respectively:



Signed binary number

When counting, positive numbers can grow indefinitely towards +?. In a similar way negative numbers can grow indefinitely towards -?.

-?, ..., -2 , -1 , 0 , +1 , +2 , ..., +?

The '-' symbol is used to indicate negative numbers

The '+' symbol is used to indicate positive numbers. In many cases the '+' symbol is omitted. Signed numbers has a single '0' (zero); having no sign.

The '+' symbol could be eliminated as it could be assumed, but the '-' symbol must still be represented somehow. Since computers do not understand anything but numbers, the '-' and the '+' symbols must be represented using two specific coded numbers. These codes could be represented using any two numbers as long as we understand them as the '-' and '+' signs when they occur at the beginning of any number. This means that the two above codes must be eliminated from the list of all possible numbers!

Over time many schemes to represent negative and positive numbers in computers were proposed. Some schemes seem to have been more successful than others. In all schemes a fixed number of bits is adopted when representing numbers.

There are several commonly known negative number representations. Since the start of computer development in the early 1950's these methods have been adopted and used at various times.

Signed magnitude representation

This early representation is intuitive since the sign and magnitude (absolute value) of a number are represented separately.

A 4 bit binary system could represent 16 ($16 = 2^4$) different numbers ranging from 0 (0000_2) to 15 (1111_2):

	Bits	3	2	1	0
0	=	0	0	0	0
.		.	.	.	.
15	=	1	1	1	1

Bit 3 is called the sign bit:

- If bit 3 = 0 => number is positive.
- If bit 3 = 1 => number is negative.

Since the MSB is reserved for the sign, 3 bits (bits 0 to 2) are left to represent the number.

The range of binary numbers represented in sign magnitude is: $-2^{n-1} - 1$ to $+ 2^{n-1} - 1$. For example, for 4 bits, the range is: -7 to + 7

The above table would become:

	Bits	Sign bit	2	1	0
+ 7	=	0	1	1	1
+ 2	=	0	0	1	0
+ 1	=	0	0	0	1
+ 0	=	0	0	0	0
- 0	=	1	0	0	0
- 1	=	1	0	0	1
- 2	=	1	0	1	0
- 7	=	1	1	1	1

Addition and subtraction with signed magnitude representation

Addition and subtraction in signed binary numbers is not straight forward, why?

Check the sign

if (Signs are Equal)
    Add magnitudes
    Append the sign to the result

else //Signs are different
    Compare the results
    Subtract the smaller from the larger
    Append sign of the greater to result

Example 1: (-3 – 2 = -5) or 1011 + 1010
(010 + 011 = 101)
append sign (1) = 1101

Example 2: (2 - 5 = -3) or 0010 + 1101
(101 – 010 = 011)
append sign (1) = 1011

Advantage and Disadvantage

Advantages:

The method is simple and intuitive for humans to represent as many negative numbers as positive numbers.

Disadvantages:

The method requires several tests and decisions (such as switching the order of operands) when performing arithmetic. It is harder to implement in computers, costing additional circuitry and execution time. In addition, dealing with the concept of borrowing is hard to implement with hardware. Finally, it has 2 representations for zero: +0 and -0! Adding +1 to -1 leads to either 1000 or 0000 both representing 0. Consequently it could be a poor choice for a computer system.

Complement representation

In order to simply computer arithmetic circuits it would be nice if subtraction is handled the same way as addition without the need to deal with borrowing.

- 5-3 = 5 + (-3)
- (-3) should be represented in a certain format.

In mathematics, the "method of complements" is a technique used to subtract one number from another using only addition of positive numbers. In brief, the number to be subtracted is first converted into its "complement", and then added to the other number.

The complement of a number M is a value that together with M makes the whole number. It is determined by the base and number of digits.

Diminished radix complement (N-1)'s complement

Positive numbers are still represented as in the sign magnitude method.
The range of numbers represented when using n bits is – $(2^{n-1} - 1)$ to $(2^{n-1} - 1)$ for 4 bits: -7 to +7

	Bits	Sign bit	2	1	0
+ 7	=	0	1	1	1
+ 2	=	0	0	1	0
+ 1	=	0	0	0	1
+ 0	=	0	0	0	0
- 0	=	1	1	1	1
- 1	=	1	1	1	0
- 2	=	1	1	0	1
- 7	=	1	0	0	0

In 1964 a supercomputer (the CDC 6600) built by Semour Cray (while at CDC - Control Data Corporation) was based on 1's complement representation. Other computers such as the PDP-1 and UNIVAC 1100/2200 series also used the 1's complement representation for doing arithmetic.

Nowadays, 1's complement representation is not used in modern computer systems.

Disadvantages:

a) It has 2 representations for zero: +0 and -0! Even though they are the same algebraically. This causes problems when doing tests on arithmetic results.
b) Not a good choice to represent negative numbers.
c) Most computers now use a variation of 1's complement (called 2's complement) that eliminates the above problems

Radix complement

Ten's complement

In Decimal (radix = 10), the Radix Complement is called the Ten's complement.

The 10's complement of 1 decimal digit is a number that must be added to it to produce 10 ($=10^1$).

- 10C of 2 = 10-2 = 8
- 10C of 10 = 10 - 10 = 0
- 10C of 0 = 10 - 0 = 10

The 10's complement for 2 decimal digits is a number that must be added to it to produce 100 ($=10^2$)

- 10C of 2 = 100 - 2 = 98
- 10C of 50 = 100 - 50 = 50 (special case)
- 10C of 0 = 100 - 00 = 100 but since we are using 2 digits it becomes 00

For 2 decimal digits, the 10's complement of 13 is calculated as (100-13 = 87).

Eliminating the borrowing

Consider the following example:

779 – 89 = 690								
borrow						6	17	
Minuend						~~7~~	~~7~~	9
Subtrahend	-						8	9
Difference						6	9	0

The above example produced a borrow of a 10 from the 7 in the 3rd column and added it to the 7 in the 2nd column. This way the subtraction could be performed without any complication on a column by column basis and thus the answer 690 could be obtained. It is precisely this borrowing process that is difficult to implement and explain to computer circuitry.
Now consider the following alternative: 779 – 89 = 779 – 89 + 999 + 1 – 1000
Here a thousand (999+1) is added and then subtracted, hence, keeping the same result.
We can also permute the operands without changing the end result:

- 779 – 89 = ((779 + (999 – 89)) + 1) – 1000
- 779 – 89 = (((999 – 89) + 1) + 779) – 1000

The new order of operations eliminates the need for borrowing!

779 – 89				
		9	9	9
-			8	9
=		9	1	0
+				1
=		9	1	1
+		7	7	9
=	1	6	9	0
-	1	0	0	0
=		6	9	0

The same process can be applied to binary subtraction.

Binary subtraction follows the same procedure. Consider the following subtraction of two binary numbers: 1100111 – 0110011 = 110100

Binary Subtraction								
borrow								
		0	10					
			~~0~~	10				
Minuend		1	~~1~~	~~0~~	0	1	1	1
Subtrahend	-	0	1	1	0	0	1	1
Difference		0	1	1	0	1	0	0

In other words:

Binary Subtraction								
borrow								
Minuend		0	10	10	0	1	1	1
Subtrahend	-	0	1	1	0	0	1	1
Difference		0	1	1	0	1	0	0

Now consider the following alternative:

110 0111 – 011 0011 = 110 0111 – 011 0011 + 111 1111+ 1 – 1000 0000

Here a 1000 000 (111 1111 + 1) was added and then subtracted, hence, keeping the same result.

We can also permute the operands without changing the end result:

110 0111 – 011 0011 = (((111 1111 – 011 0011) + 1) + 110 0111) – 1000 000

The new order of operations eliminates the need for borrowing!

	Bits	7	6	5	4	3	2	1	0
			1	1	1	1	1	1	1
-			0	1	1	0	0	1	1
=			1	0	0	1	1	0	0
+									1
=			1	0	0	1	1	0	1
+			1	1	0	0	1	1	1
=		1	0	1	1	0	1	0	0
-		1	0	0	0	0	0	0	0
=		0	0	1	1	0	1	0	0

Eliminating the subtraction

	Step\Bit	7	6	5	4	3	2	1	0
Subtrahend	***1***		0	1	1	0	0	1	1
1's Complement	***2***		1	0	0	1	1	0	0
+	3								1
2's Complement	***4***		1	0	0	1	1	0	1
+ ***Minuend***	***5***		1	1	0	0	1	1	1
=	6	1	0	1	1	0	1	0	0
Drop MSB	***7***		0	1	1	0	1	0	0

Step 2 (the 1's complement) is obtained by simply flipping the 0s of the number to 1s and the 1s to 0s.

Step 7 could be obtained by simply ignoring bit #7 (the MSB).

“Voila” no more subtraction.

2's complement

In Binary systems, a slight variation of the 1's complement is used to avoid the problem discussed in example 2 of the ones complement section above, thus using all ones (1111) to represent -1. In this case -1 + 1 = 1111 + 0001 = 0000 (1 discarder). Positive numbers are still represented as in the sign magnitude method. For negative numbers, all the bits are then complemented as follows:

When using the Binary (when radix = 2), the Radix Complement is called the Two's complement. Using one bit only the 2's complement of 0 is 0 because (10-0 = 1 0 and the left most bit is dropped) and the complement of 1 is 1 (10-1 = 0 1 and the left most bit is dropped. This resembles a one position binary odometer where to get the 2's complement of a positive number you reverse the odometer 2* the value of the number. For 2 binary digits or bits, the 2's complement of 01(or the negative representation of 01) is calculated as 100-01= 11. This resembles a 2 bits binary odometer where to get the 2's complement of a positive number you reverse the odometer 2* the value of the number.

Since computers use the binary number system, the 2's complement is used rather than the 10's complement.

In Math terms the 2's complement of M is: $2^n - M$ where n is the number of bits representing M.

2's Complement of M = $(2^n - 1) - M + 1$, = 1's complement of M + 1

Example 1:

Compute the 2's complement of M = 0100_2
M = 0100_2 => 1's of M = 1011_2
2's complement of M = $1011_2 + 1 = 1100_2$

An easier way to obtain the 2's complement of a binary number is to parse that number starting with the LSB until you find the rightmost digit that is equal to 1. Leave that digit and all other digits to the right of it unchanged, and then complement all digits to the left of that one digit.

The 2's complement of a negative number represented in its 2's complement form is equal to its corresponding positive number.

The range of numbers using n bits is $-(2^{n-1})$ to $(2^{n-1} - 1)$

For 4 bits the range is: -8 to +7

	Bits	***Sign bit***	***2***	***1***	***0***
+ 7	=	0	1	1	1
+ 1	=	0	0	0	1
+ 0	=	0	0	0	0
- 1	=	1	1	1	1
- 8	=	1	0	0	0

Advantages:

It has a single representation of the zero: 0

Each positive number has a corresponding negative number that starts with a 1, except -8 which has no corresponding positive number.

Example 2: 5-5 is computed as follows in a 4-bit precision:

0101+ 1011 = 10000. The left most bit is lost since we are limited by 4 bit precision (corresponds to a 4 bit hardware inside the computer), so the answer is 0000 which is the only representation of zero.

The 2's complement simplifies the logic required for addition and subtraction, since can use the addition to add and subtract both negative and positive numbers the same way, hence reducing and optimizing the underlying computer circuitry.

Nowadays, almost all computer systems are based on the 2's complement representation.

Comparing the 3 representations for signed binary numbers using 4 bits.

Binary Sequence	***2's Complement***	***1's Complement***	***Sign Magnitude***
0111	7	7	7
0110	6	6	6
0101	5	5	5
0100	4	4	4
0011	3	3	3
0010	2	2	2
0001	1	1	1
0000	0	0	0
1111	-1	-0	-7
1110	-2	-1	-6
1101	-3	-2	-5
1100	-4	-3	-4
1011	-5	-4	-3
1010	-6	-5	-2
1001	-7	-6	-1
1000	-8	-7	-0

Subtractor

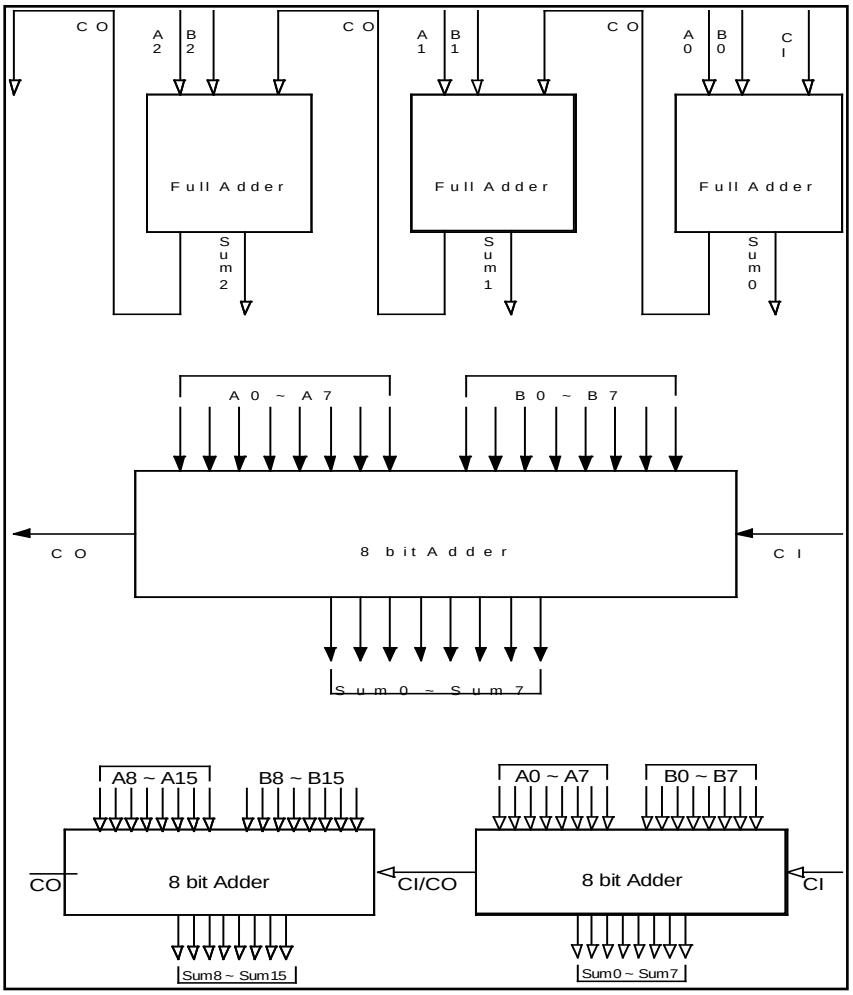
This section adds subtraction to an 8 bit adder. Adding subtraction to the full adder described so far requires us to analyze what happens when subtraction is performed.

Subtraction as performed by humans is very difficult to explain to a group of logical gates. The back and forth borrowing mechanism used by humans to perform subtraction has to be replaced by a simpler operation.

Since the 8 bit adder can already perform binary addition it is only natural to attempt to convert the subtraction into an addition of a positive number added to a negative number as in:

3 – 2 would be performed as ***3 + (-2).***

Below is a schematic representing how several full adders can be connected to build an 8 bit adder. A 16 bit adder can be build following a similar process using two 8 bit adders.



Eliminating the borrowing

Consider the following example:

779 – 89 = 690								
borrow						6	17	
Minuend						~~7~~	~~7~~	9
Subtrahend							8	9
Difference						6	9	0

The above example produced a borrow of a 10 from the 7 in the 3d column and added to the 7 in column number 2. This way the subtraction could be performed without any complication on a column by column basis and thus the answer 690 could be obtained. It is precisely this process that is difficult to explain to the adder.

Now consider the following alternative:

$$779 - 89 = 779 - 89 + 999 + 1 - 1000$$

Here a thousand (999+1) was added and then subtracted, hence, keeping the same result.
We can also permute the operands without changing the end result:

$$779 - 89 = \left(\left(779 + \left(999 - 89\right)\right) + 1\right) - 1000$$

$$779 - 89 = \left(\left(\left(999 - 89\right) + 1\right) + 779\right) - 1000$$

The new order of operations eliminates the need to borrowing!

		9	9	9
-			8	9
=		9	1	0
+				1
=		9	1	1
+		7	7	9
=	1	6	9	0
-	1	0	0	0
=		6	9	0

The same process can be applied to binary subtraction.

Binary subtraction follows the same procedure. Consider the following subtraction of two binary numbers: 1100111 – 0110011 = 110100

Binary Subtraction								
borrow								
		0	10					
			~~0~~	10				
Minuend		1	~~1~~	~~0~~	0	1	1	1
Subtrahend	-	0	1	1	0	0	1	1
Difference		0	1	1	0	1	0	0

In other words:

Binary Subtraction								
borrow								
Minuend		0	10	10	0	1	1	1
Subtrahend	-	0	1	1	0	0	1	1
Difference		0	1	1	0	1	0	0

Now consider the following alternative:
110 0111 – 011 0011 = 110 0111 – 011 0011 + 111 1111+ 1 – 1000 0000
Here a 1000 000 (111 1111 + 1) was added and then subtracted, hence, keeping the same result.
We can also permute the operands without changing the end result:
110 0111 – 011 0011 = (((111 1111 – 011 0011) + 1) + 110 0111) – 1000 000

The new order of operations eliminates the need to borrowing!

		7	***6***	***5***	***4***	***3***	***2***	***1***	***0***
			1	1	1	1	1	1	1
-			0	1	1	0	0	1	1
=			1	0	0	1	1	0	0

		7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
+									1
=			1	0	0	1	1	0	1
+			1	1	0	0	1	1	1
=		1	0	1	1	0	1	0	0
-		1	0	0	0	0	0	0	0
=		0	0	1	1	0	1	0	0

Eliminating Subtraction all together

	Step\Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Subtrahend	***1***		0	1	1	0	0	1	1
1's Complement	***2***		1	0	0	1	1	0	0
+	3								1
2's Complement	***4***		1	0	0	1	1	0	1
+ ***Minuend***	***5***		1	1	0	0	1	1	1
=	6	1	0	1	1	0	1	0	0
Drop MSB	**7**		0	1	1	0	1	0	0

Step 2 (the 1's complement) is obtained by simply flipping the 0s of the number to 1s and the 1s to 0s.

Step 7 could be obtained by simply ignoring bit #7 (the MSB).