School of Engineering and Computer Science

Numerical Applied Mathematics

(Signed Magnitude Binary Numbers)

Kamel ATTAR
attar.kamel@gmail.com

Week #10 — 11 ✦ 5/JAN/2024 ✦

1 How to represent negative numbers in binary?
- Sign and Magnitude Method
- Complement Number System Method
  - Car Odometer
  - **1′**s complement
  - **2′**s complement
- Summary

2 Sign Extension

3 Overflow condition
- Definition
- Signed subtraction
  - Signed subtraction with no overflow
  - Signed subtraction with overflow correction
- Unsigned Addition/Subtraction
  - Unsigned Addition

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary
2 Sign Extension
3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Signed Magnitude Number System ★

How to represent negative numbers in binary?

- In mathematics, positive numbers (including zero) are represented as unsigned numbers. That is we do not put the +ve sign in front of them to show that they are positive numbers. But when dealing with negative numbers we do use a −ve sign in front of the number to show that the number is negative in value and different from a positive unsigned value, and the same is true with **signed binary numbers**.

★ Signed Magnitude Number System ★

How to represent negative numbers in binary?

- In mathematics, positive numbers (including zero) are represented as unsigned numbers. That is we do not put the +ve sign in front of them to show that they are positive numbers. But when dealing with negative numbers we do use a −ve sign in front of the number to show that the number is negative in value and different from a positive unsigned value, and the same is true with **signed binary numbers**.
- However, in digital circuits there is no provision made to put a plus or even a minus sign to a number, since digital systems operate with binary numbers that are represented in terms of “0’s” and “1’s”.

★ Signed Magnitude Number System ★

How to represent negative numbers in binary?

- In mathematics, positive numbers (including zero) are represented as unsigned numbers. That is we do not put the +ve sign in front of them to show that they are positive numbers. But when dealing with negative numbers we do use a −ve sign in front of the number to show that the number is negative in value and different from a positive unsigned value, and the same is true with **signed binary numbers**.
- However, in digital circuits there is no provision made to put a plus or even a minus sign to a number, since digital systems operate with binary numbers that are represented in terms of "0's" and "1's".
- The symbols "+" and "−" were used to represent the sign of a decimal number. But how do we represent signed binary numbers if all we have is a bunch of **1**'s and **0**'s.

$$-50_{10} \underset{?}{=} \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare.$$

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary
2 Sign Extension
3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Signed Magnitude Number System ★

Sign and Magnitude Method

- An alternative is to use an **extra digit** to represent positive and negative instead of introducing a new symbol. The most significant bit (**MSB**) is used as the sign bit.
  - If the sign bit is "**0**", this means the number is positive in value.
  - If the sign bit is "**1**", then the number is negative in value.

  The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.

★ Signed Magnitude Number System ★

Sign and Magnitude Method

- An alternative is to use an **extra digit** to represent positive and negative instead of introducing a new symbol. The most significant bit (**MSB**) is used as the sign bit.
  - If the sign bit is "**0**", this means the number is positive in value.
  - If the sign bit is "**1**", then the number is negative in value.

  The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.
- Thus, if a computer is capable of handling numbers of **8**-bit numbers:

$$-53_{10} = \underbrace{\underbrace{\blacksquare}_{\text{sign bit}}\ \underbrace{\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare}_{\text{Magnitude bits}}}_{\text{8—bit word}} \qquad +53_{10} = \underbrace{\underbrace{\blacksquare}_{\text{sign bit}}\ \underbrace{\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare}_{\text{Magnitude bits}}}_{\text{8—bit word}}.$$

★ Signed Magnitude Number System ★

- An alternative is to use an **extra digit** to represent positive and negative instead of introducing a new symbol. The most significant bit (**MSB**) is used as the sign bit.
  - If the sign bit is "**0**", this means the number is positive in value.
  - If the sign bit is "**1**", then the number is negative in value.

  The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.
- Thus, if a computer is capable of handling numbers of **8**-bit numbers:

$$-53_{10} = \underbrace{\underbrace{1}_{\text{sign bit}}\ \underbrace{0\,1\,1\,0\,1\,0\,1}_{\text{Magnitude bits}}}_{8-\text{bit word}} \qquad +53_{10} = \underbrace{\underbrace{0}_{\text{sign bit}}\ \underbrace{0\,1\,1\,0\,1\,0\,1}_{\text{Magnitude bits}}}_{8-\text{bit word}}.$$

Example

$$(1\,1111111)_2 = -255 = -7F_{16}\,, \qquad (0\,1111111)_2 = +255 = +7F_{16}\,.$$

✎ **Exercise 1.**

a. $(01101101)_2 = (\cdots)_{10}$ b. $(11101101)_2 = (\cdots)_{10}$ c. $(00101011)_2 = (\cdots)_{10}$
d. $(10101011)_2 = (\cdots)_{10}$.

★ Signed Magnitude Number System ★

Sign and Magnitude Method

- The disadvantage here is that whereas before we had a full range n-bit unsigned binary number, we now have an $n-1$ bit signed binary number giving a reduced range of digits

$$\text{from } -(2^{n-1}-1) \quad \text{to} \quad +(2^{n-1}-1)$$

Example

If we have **4** bits to represent a signed binary number, (**1**-bit for the Sign bit and **3**-bits for the Magnitude bits), then the actual range of numbers we can represent in sign-magnitude notation would be:

$$-(2^{4-1}-1) \text{ to } +(2^{4-1}-1) \iff -7 \text{ to } +7$$

1111	1110	1101	1100	1011	1010	1001	1000	0000	0001	0010	0011	0100	0101	0110	0111
−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7

Whereas before, the range of an unsigned **4**-bit binary number would have been from **0** to **15**, or **0** to ***F*** in hexadecimal, we now have a reduced range of -7 to $+7$. Thus an unsigned binary number does not have a single sign-bit, and therefore can have a larger binary range as the most significant bit (MSB) is just an extra bit or digit rather than a used sign bit

★ Signed Magnitude Number System ★

Sign and Magnitude Method

- The disadvantage here is that whereas before we had a full range n-bit unsigned binary number, we now have an $n-1$ bit signed binary number giving a reduced range of digits

$$\text{from } -(2^{n-1}-1) \quad \text{to} \quad +(2^{n-1}-1)$$

Example

If we have **4** bits to represent a signed binary number, (**1**-bit for the Sign bit and **3**-bits for the Magnitude bits), then the actual range of numbers we can represent in sign-magnitude notation would be:

$$-(2^{4-1}-1) \text{ to } +(2^{4-1}-1) \iff -7 \text{ to } +7$$

1111	1110	1101	1100	1011	1010	1001	1000	0000	0001	0010	0011	0100	0101	0110	0111
−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7

Whereas before, the range of an unsigned **4**-bit binary number would have been from **0** to **15**, or **0** to ***F*** in hexadecimal, we now have a reduced range of -7 to $+7$. Thus an unsigned binary number does not have a single sign-bit, and therefore can have a larger binary range as the most significant bit (MSB) is just an extra bit or digit rather than a used sign bit

- Another disadvantage here of the sign-magnitude form is that we can have a positive result for zero, $+0$ or 0000_2, and a negative result for zero, -0 or 1000_2. Both are valid but which one is correct.

Note that for a **4**-bit, **6**-bit, **8**-bit, **16**-bit or **32**-bit signed binary number all the bits MUST have a value, therefore "**0**'s" are used to fill the spaces between the leftmost sign bit and the first or highest value "**1**".

Example

$-\mathbf{13}_{10}$ as a **8**-bit number is **10001101** and $+\mathbf{13}_{10}$ as a **8**-bit number is **00001101**

✎ **Exercise 2.** Convert the following decimal values into signed binary numbers using the sign-magnitude format:

a. $-\mathbf{15}_{10}$ as a **6**-bit number
b. $+\mathbf{23}_{10}$ as a **7**-bit number
c. $-\mathbf{56}_{10}$ as a **8**-bit number
d. $+\mathbf{85}_{10}$ as a **8**-bit number
e. $-\mathbf{127}_{10}$ as a **8**-bit number

Note that for a **4**-bit, **6**-bit, **8**-bit, **16**-bit or **32**-bit signed binary number all the bits MUST have a value, therefore "**0**'s" are used to fill the spaces between the leftmost sign bit and the first or highest value "**1**".

Example

$-\mathbf{13}_{10}$ as a **8**-bit number is **10001101** and $+\mathbf{13}_{10}$ as a **8**-bit number is **00001101**

Can we use sign and magnitude representation to do subtraction?

Note that for a **4**-bit, **6**-bit, **8**-bit, **16**-bit or **32**-bit signed binary number all the bits MUST have a value, therefore "**0**'s" are used to fill the spaces between the leftmost sign bit and the first or highest value "**1**".

Example

$-\mathbf{13}_{10}$ as a **8**-bit number is **10001101** and $+\mathbf{13}_{10}$ as a **8**-bit number is **00001101**

Can we use sign and magnitude representation to do subtraction?

$$\mathbf{1-6=-5} \iff +$$

Borrow				
	0	**0**	**0**	**1**
	1	**1**	**1**	**0**
Result				

Note that for a **4**-bit, **6**-bit, **8**-bit, **16**-bit or **32**-bit signed binary number all the bits MUST have a value, therefore "**0**'s" are used to fill the spaces between the leftmost sign bit and the first or highest value "**1**".

Example

$-\mathbf{13}_{10}$ as a **8**-bit number is **10001101** and $+\mathbf{13}_{10}$ as a **8**-bit number is **00001101**

Can we use sign and magnitude representation to do subtraction?

$$1 - 6 = -5 \iff +$$

Borrow				
	0	0	0	1
	1	1	1	0
Result	1	1	1	1

????

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary
2 Sign Extension
3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Signed Magnitude Number System ★

Car Odometer

$$
\begin{array}{r|ccccc}
 & & & & & \\
\hline
 & 9 & 9 & 9 & 9 & 9 \\
+ & 0 & 0 & 0 & 0 & 1 \\
\hline
 & & & & & \\
\end{array}
$$

★ Signed Magnitude Number System ★

Car Odometer

$$
\begin{array}{r|ccccc}
 & & & & & \\
\hline
 & 9 & 9 & 9 & 9 & 9 \\
+ & 0 & 0 & 0 & 0 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$\Longrightarrow 99999 = -1$$

Example (Finding the opposite of **328769**)

$$
\begin{array}{r|cccccc}
 & & & & & & \\
\hline
 & 3 & 2 & 8 & 7 & 6 & 9 \\
+ & & & & & & \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

★ Signed Magnitude Number System ★

Car Odometer

$$
\begin{array}{r|ccccc}
 & & & & & \\
\hline
 & 9 & 9 & 9 & 9 & 9 \\
+ & 0 & 0 & 0 & 0 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$\Longrightarrow 99999 = -1$$

Example (Finding the opposite of **328769**)

$$
\begin{array}{r|cccccc}
 & & & & & & \\
\hline
 & 3 & 2 & 8 & 7 & 6 & 9 \\
+ & 6 & 7 & 1 & 2 & 3 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\qquad \Longrightarrow 671231 = -328769
$$

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

	+26	0	0	0	1	1	0	1	0
+		?	?	?	?	?	?	?	?
		0	0	0	0	0	0	0	0

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

0	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**+26**

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

	0	**0**	**0**	**1**	**1**	**0**	**1**	**0**	+26
+	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	Invert bits
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

	0	**0**	**0**	**1**	**1**	**0**	**1**	**0**	+26
+	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	Invert bits
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
+	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	Add one
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

	0	**0**	**0**	**1**	**1**	**0**	**1**	**0**	$+26$
$+$	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	Invert bits
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
$+$	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	Add one
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	

Thus

$$-26 = \underbrace{\underbrace{11100101}_{1's \text{ complement of}(+26)} + \underbrace{00000001}_{\text{add one}}}_{2's \text{ complement of}(+26)} = 11100110$$

★ Signed Magnitude Number System ★

Complement Number System Method

Given and **8** bit signed binary number

	0	**0**	**0**	**1**	**1**	**0**	**1**	**0**	$+26$
$+$	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	Invert bits
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
$+$	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	Add one
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	

Thus

$$-26 = \underbrace{\underbrace{\mathbf{1\,1\,1\,0\,0\,1\,0\,1}}_{\mathbf{1's}\text{ complement of}(+26)} + \underbrace{\mathbf{0\,0\,0\,0\,0\,0\,0\,1}}_{\text{add one}}}_{\mathbf{2's}\text{ complement of}(+26)} = \mathbf{1\,1\,1\,0\,0\,1\,1\,0}$$

$$\mathbf{1\,1\,1\,0\,0\,1\,1\,0} = -128 + 64 + 32 + 4 + 2 = -26$$

★ Complement Number System ★

1′s complement

One's Complement or **1's Complement** as it is also termed, is another method which we can use to represent negative binary numbers in a signed binary number system.

- In **1**'s complement, positive numbers (also known as non-complements) remain unchanged as before with the sign-magnitude numbers.
- Negative numbers however, are represented by taking the one's complement (inversion, negation) of the signed positive number.
- The **1**'s complement of a number is found by changing all **1**'s to **0**'s and all **0**'s to **1**'s.

Original Value	One's Complement
0	1
1	0
1010	0101
1111	0000
11110000	00001111
10100011	01011100

- There are various uses of **1**'s complement of Binary numbers, mainly in signed Binary number representation and various arithmetic operations (additions, subtractions, etc).

★ Complement Number System ★

1's complement

Example

Let's see how to find the 1's complement of $-$**33**, $-$**127** and $-$**1** in **8** bit (**1** byte)?

- **33** is represented as $(100001)_2$

  In **8** bit notation, it is represented as $(0010\ 0001)_2$

  Now, $-$**33** is represented in one's complement as $(1101\ 1110)_2$
- In **8** bit notation, **127** is represented as $(0111\ 1111)_2$

  Now, $-$**127** is represented in one's complement as $(1000\ 0000)_2$
- **1** is represented as $(01)_2$

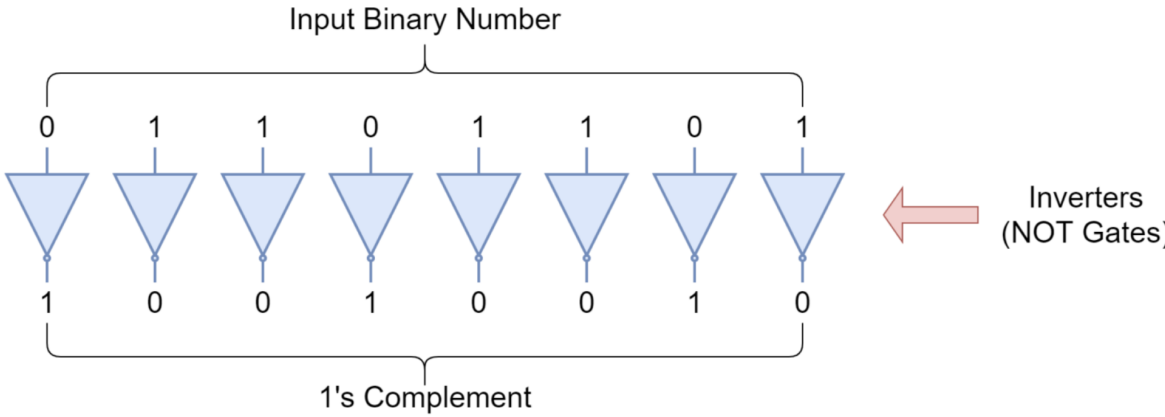
  In **8** bit notation, it is represented as $(0000\ 0001)_2$

  Now, $-$**1** is represented in one's complement as $(1111\ 1110)_2$

★ Complement Number System ★

1's complement

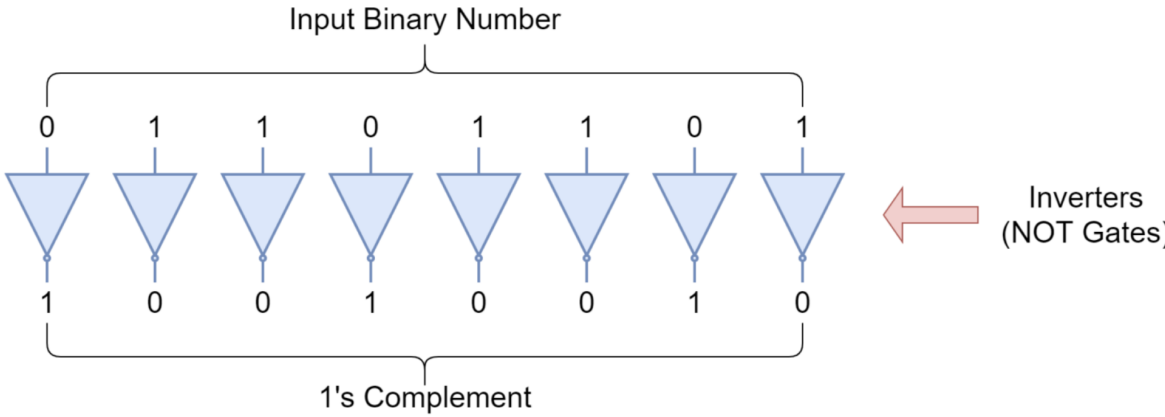
- The easiest way to find the one's complement of a signed binary number when building digital arithmetic or logic decoder circuits is to use Inverters.



★ Complement Number System ★

1′s complement

- The easiest way to find the one's complement of a signed binary number when building digital arithmetic or logic decoder circuits is to use Inverters.

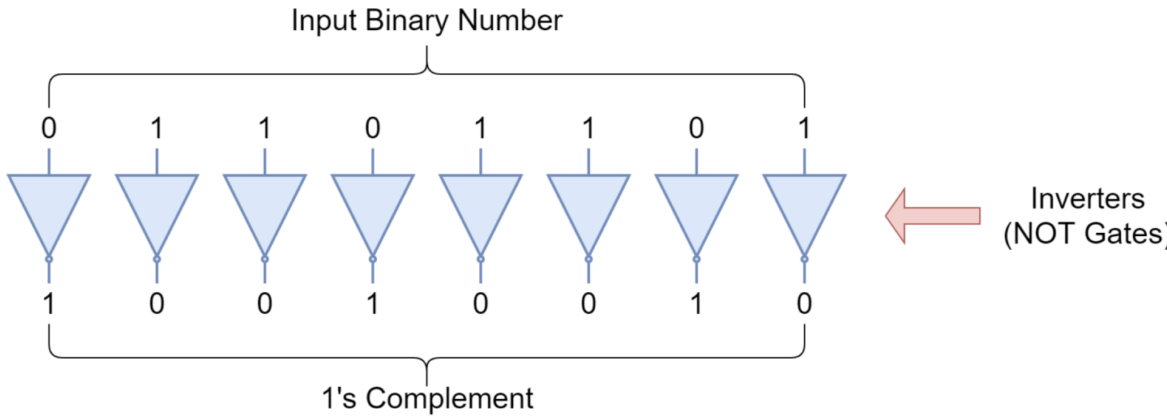


- Also just like the sign-magnitude representation, one's complement can also have n-bit notation to represent numbers in the range from: $-(2^{(n-1)} - 1)$ and $+2^{(n-1)} - 1$.

★ Complement Number System ★

1′s complement

- The easiest way to find the one's complement of a signed binary number when building digital arithmetic or logic decoder circuits is to use Inverters.



- Also just like the sign-magnitude representation, one's complement can also have n-bit notation to represent numbers in the range from: $-(2^{(n-1)} - 1)$ and $+2^{(n-1)} - 1$.

Example

A 4-bit representation in the one's complement format can be used to represent decimal numbers in the range from -7 to $+7$ with two representations of zero: **0000** ($+0$) and **1111** (-0) the same as before.

1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
-7	-6	-5	-4	-3	-2	-1	-0	$+0$	$+1$	$+2$	$+3$	$+4$	$+5$	$+6$	$+7$

★ Complement Number System ★

2′s complement

- In computers, subtraction is generally carried out by 2's complement to handle negative numbers.

★ Complement Number System ★

2′s complement

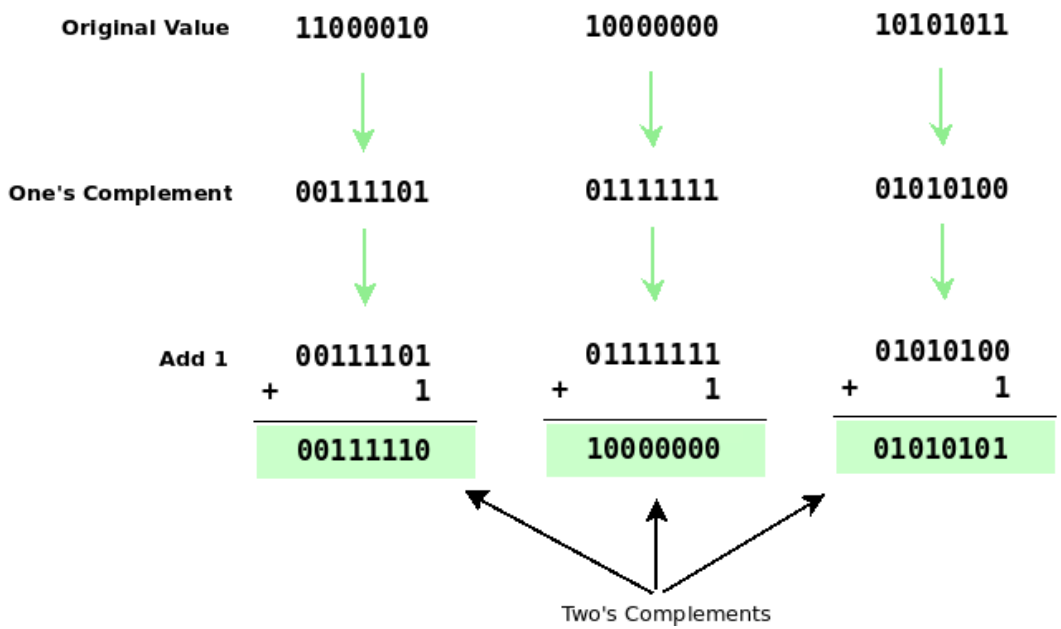
- In computers, subtraction is generally carried out by 2's complement to handle negative numbers.
- We build one adder and it can add positive and negative numbers alike. So instead of building an adder and a subtractor, you just build one adder and a really trivial negator.

★ Complement Number System ★

2′s complement

- In computers, subtraction is generally carried out by 2's complement to handle negative numbers.
- We build one adder and it can add positive and negative numbers alike. So instead of building an adder and a subtractor, you just build one adder and a really trivial negator.

The **2**'s complement of binary number is obtained by adding **1** to the Least Significant Bit (LSB) of **1**'s complement of the number $\mathbf{2's = 1's + 1}$.



★ 2′s complement ★

- In two's complement representation, positive numbers are simply represented as themselves, and negative numbers are represented by the two's complement of their positive number.
  $A - B = A +$ (2's complement of B)

1001	1010	1011	1100	1101	1110	1111	0000	0000	0001	0010	0011	0100	0101	0110	0111
−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7

★ 2′s complement ★

- In two's complement representation, positive numbers are simply represented as themselves, and negative numbers are represented by the two's complement of their positive number.
  $A - B = A +$ (2's complement of B)

1001	1010	1011	1100	1101	1110	1111	0000	0000	0001	0010	0011	0100	0101	0110	0111
−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7

- The main advantage of two's complement over the previous one's complement is that there is no double-zero problem plus it is a lot easier to generate the two's complement of a signed binary number. Therefore, arithmetic operations are relatively easier to perform when the numbers are represented in the two's complement format.

★ 2′s complement ★

- In two's complement representation, positive numbers are simply represented as themselves, and negative numbers are represented by the two's complement of their positive number.
  $A - B = A +$ (2's complement of B)

1001	1010	1011	1100	1101	1110	1111	0000	0000	0001	0010	0011	0100	0101	0110	0111
−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7

- The main advantage of two's complement over the previous one's complement is that there is no double-zero problem plus it is a lot easier to generate the two's complement of a signed binary number. Therefore, arithmetic operations are relatively easier to perform when the numbers are represented in the two's complement format.
- The only disadvantage is that if we want to represent negative binary numbers in the signed binary number format, we must give up some of the range of the positive number we had before.

★ 2′s complement ★

Example

Let's look at the subtraction of our two **8**-bit numbers **115** and **27** from above using two's complement, and we remember from above that the binary equivalents are:

$$\mathbf{115_{10}} \text{ in binary is: } \mathbf{01110011_2} \text{ and } \mathbf{27_{10}} \text{ in binary is: } \mathbf{00011011_2}$$

Solution

Decimal	Binary	1's complement	2's complement
27	**00011011**	**11100100**	**11100101**

												Carry	**1**	**1**	**1**			**1**	**1**	**1**	
	115		**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**		(+115)		**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**
−	**27**		**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	⟺	(−27)		**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**
	82											Result		**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

As previously, the **9**th overflow bit is disregarded as we are only interested in the first **8**-bits, so the result is: $\mathbf{01011000_2}$ or $\mathbf{(64 + 16 + 8) = 88_{10}}$ in decimal the same as before.

★ 2's complement ★

Example

Let's look at the subtraction of our two **8**-bit numbers **9** and **14** from above using two's complement, and we remember from above that the binary equivalents are:

9_{10} in binary is: $\mathbf{0000\ 1001_2}$ and 14_{10} in binary is: $\mathbf{0000\ 1110_2}$

Solution

Decimal	Binary	1's complement	2's complement
14	**0000 1110**	**1111 0001**	**1111 0010**

$$
\begin{array}{|c|cccccccc|}
\hline
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
- & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\
\hline
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}
\quad \Longleftrightarrow \quad
\begin{array}{|c|cccccccc|}
\hline
\text{Carry} & & & & & & & & \\
+ & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\
\hline
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}
$$

9	(+9)
14	(−14)
−5	Result

There is no overflow bit, this says result Is still in two's complement form. the result of subtraction is negative and is obtained by writing the **2**'s complement of **1111 1011** which is equal to **0000 0101** and since the result is negative, the sign bit should be **1**. Hence the difference is **−0000 0101 = −5.**

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary
2 Sign Extension
3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Summary ★

- The signed binary number system is used to represent both positive and negative numbers in binary. The methods used to represent signed binary numbers are Sign-and-Magnitude, One's Complement, and Two's Complement.

−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7
Sign and Magnitude Method															
1111	1110	1101	1100	1011	1010	1001	1000	0000	0001	0010	0011	0100	0101	0110	0111
1's complement Method															
1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
2's complement Method															
1001	1010	1011	1100	1101	1110	1111	0000	0000	0001	0010	0011	0100	0101	0110	0111

- The Sign-and-Magnitude uses the most significant bit (MSB) as a sign bit. The zero (0) shows a positive magnitude and a one shows a negative magnitude of a binary number. The range of numbers is split into $-\mathbf{127}$ to $-\mathbf{0}$ and $+\mathbf{0}$ to $+\mathbf{127}$. The disadvantage involves double zeros i.e. $\mathbf{0000}_2$ and $\mathbf{1000}_2$.
- In One's Complement method, the positive numbers remain unchanged while the negative numbers are obtained by inverting each bit of its relevant counterpart positive. The range of numbers is split into $-\mathbf{127}$ to $-\mathbf{0}$ and $+\mathbf{0}$ to $+\mathbf{127}$. Similar to Sign-and-Magnitude, the One's Complement has the disadvantage of having double zero in its range of numbers. The addition or adder can be used along with one's complement to perform the subtraction of two binary numbers.

−7	−6	−5	−4	−3	−2	−1	−0	+0	+1	+2	+3	+4	+5	+6	+7
1111	1110	1101	1100	1011	1010	1001	1000	0000	0001	0010	0011	0100	0101	0110	0111
1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
1001	1010	1011	1100	1101	1110	1111	0000	0000	0001	0010	0011	0100	0101	0110	0111

- The Two's Complement has unaltered positive numbers just like Sign-and-Magnitude and One's Complement methods. The negative number is obtained by first determining the one's complement of the relevant positive number and then adding "**1**" to it. The range of numbers is **−128** to **0** to **+127** and includes only one zero in the range.
- The two's complement is the most commonly used method to represent signed binary numbers compared to other presented methods. The usage of the two's complement method avoids the issue of double zero and the arithmetic operations become relatively much easier.

✎ **Exercise 3.** Complete the following table

Binary	**1**'s complement	**2**'s complement
1100 1001		
0000 1111		
0111 0011 0001 0000		

✎ **Exercise 4.** Calculate the **1**'s and **2**'s complements for the following numbers expressed in hexadecimal form. Do the calculation in binary then write down the answer in hex.

a. $(AA)_{16}$ b. $(FF)_{16}$ c. $(1248)_{16}$.

✎ **Exercise 5.** Convert these negative decimal values to negative binary using two's complement on **1** byte:

a. $-$ **192;** b. $-$ **16;** c. $-$ **1;** d. $-$ **0.**

✎ **Exercise 6.** Find the value of $C0_{16}$ in decimal number if:

a. it is an unsigned number?
b. if it is a signed number?

✎ **Exercise 7.** The following codes have a size of **16** bits (**2** bytes), they are signed and given in hexadecimal. Calculate their values and give the answer in decimal.

a. ***FFFF*** b. **8000** c. **7*FFF*** d. **00*FF***

✎ **Exercise 8.**

a. How to write $-$**255** in binary? How many bytes are needed at least to encode this value?
b. Can the value $-$**192** be encoded in a byte? justify your answer.
c. Write $-$**150** in binary and hexadecimal numbers?
d. What is 8001_{16} depending on whether this **2** byte code is signed or unsigned?

✎ **Exercise 9.** Convert the following **2**'s complement binary numbers to decimal.

a. **0110** b. **1101** c. **0110 1111** d. **1101 1011 0001 1100**

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary
2 Sign Extension
3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Sign Extension ★

Negative sign number

- Sign extension is the operation, in computer arithmetic, of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value.
- This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used.
- Sign-extending means copying the **sign bit** of the unextended value to *all bits on the left side* of the larger-size value.
- Moving **2**'s complement integers between data types of different sizes is *illegal* without the **sign extension** operation.
- When necessary, we **must** *sign-extend* integer to convert *signed value* to a larger size.

Example

For example, **8**-bit encoding of decimal number $-$**56** can be sign-extended as follows:

11001000 ← **8**-bit value of $-$ **56**

11111111 11001000 ← extended to **16**-bit value

11111111 11111111 11111111 11001000 ← extended to **32**-bit value

★ Sign Extension ★

Positive sign number

- The same way as leading zeroes do not affect values of a positive numbers, leading **1**s do not affect values of a negative numbers.
- Sign-extend operation is often abbreviated *SEXT*.
- Sign extending operation efficiently converts positive values as well, provided the sign bit is zero:

Example

For example, **8**-bit encoding of decimal number **+72** can be sign-extended as follows:

01001000 ← **8**-bit value of **+72**
00000000 01001000 ← extended to **16**-bit value
00000000 00000000 00000000 01001000 ← extended to **32**-bit value

✎ **Exercise 10.** Express the negative value **−22** as a **2**'s complement integer, using eight bits. Repeat it for **16** bits and **32** bits. What does this illustrate with respect to the properties of sign extension as they pertain to **2**'s complement representation?

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary

2 Sign Extension

3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary

2 Sign Extension

3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Overflow condition ★

- If an addition or a subtraction produces a result that exceeds the range of the number system (the data width allocated to the result), overflow is said to occur.

- For example, when the sum of two positive **8**-bit numbers exceeds **127**, we have an overflow.
- Similarly, if sum of two negative **8**-bit numbers is less than or equal to $-$**128**, we have an overflow.

★ Overflow condition ★

- If an addition or a subtraction produces a result that exceeds the range of the number system (the data width allocated to the result), overflow is said to occur.
- Overflow indicates that the result was too large or too small to fit in the original data type.

★ Overflow condition ★

- If an addition or a subtraction produces a result that exceeds the range of the number system (the data width allocated to the result), overflow is said to occur.
- Overflow indicates that the result was too large or too small to fit in the original data type.
- A simple rule exists for detecting overflow:
  1. Addition of two numbers with different signs can never produce overflow.
  2. When two signed (**2**'s complement) numbers are added, overflow is detected if:
     a. both operands are positive and the result is negative, or
     b. both operands are negative and the result is positive.

★ Overflow condition ★

- If an addition or a subtraction produces a result that exceeds the range of the number system (the data width allocated to the result), overflow is said to occur.
- Overflow indicates that the result was too large or too small to fit in the original data type.
- A simple rule exists for detecting overflow:
  1. Addition of two numbers with different signs can never produce overflow.
  2. When two signed (**2**'s complement) numbers are added, overflow is detected if:
     a. both operands are positive and the result is negative, or
     b. both operands are negative and the result is positive.

$A + B$			
Operand A	Operand B	Result	Overflow ?
≥ 0	≥ 0	< 0	Yes
< 0	< 0	≥ 0	Yes
≥ 0	≥ 0	≥ 0	No
< 0	< 0	< 0	No
≥ 0	< 0	< 0	No
≥ 0	< 0	≥ 0	No
< 0	≥ 0	< 0	No
< 0	≥ 0	≥ 0	No

★ Overflow condition ★

- If an addition or a subtraction produces a result that exceeds the range of the number system (the data width allocated to the result), overflow is said to occur.
- Overflow indicates that the result was too large or too small to fit in the original data type.
- A simple rule exists for detecting overflow:
  1. Addition of two numbers with different signs can never produce overflow.
  2. When two signed (**2**'s complement) numbers are added, overflow is detected if:
     a. both operands are positive and the result is negative, or
     b. both operands are negative and the result is positive.
  3. When two unsigned numbers are added, overflow occurrs if there is a carry out of the leftmost bit.

★ Overflow condition ★

The following example illustrates overflowed computation for **4**-bit arithmetic. Keep in mind that the range of **4**-bit number can represent is from -8 to $+7$. Exceeding this range causes the overflow.

Example

	Carry	1			
+	(−3)		1	1	0	1
	(−6)		1	0	1	0
	Result	1	0	1	1	1

	Carry	1			
+	(+5)	0	1	0	1
	(+6)	0	1	1	0
	Result	1	0	1	0

	Carry	1			
+	(−8)		1	0	0	0
	(−8)		1	0	0	0
	Result	1	0	0	0	0

	Carry	1	1	1	
+	(+7)	0	1	1	1
	(+7)	0	1	1	1
	Result	1	1	1	0

All of the examples given above have the overflow error condition. What that means is that you cannot compute the given numbers with only four bits. You need more bit positions, if you wish to correct the error.

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary

2 Sign Extension

3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Overflow condition ★

Signed subtraction

- Signed subtraction in most computers is done by taking **2**'s complement of the subtrahend and then adding it to the minuend following the normal rules of addition.
- Overflow condition must be checked after the addition in order to obtain the correct computational result.
  - If no overflow condition is detected, the correct answer of the subtraction is obtained from the result by simply discarding the carry-out bit of the MSB if a carry-out bit exists.
  - If an overflow condition is detected, there are two ways of dealing with this error.
    * The first approach is simply reporting an error message that indicates the overflow condition. Most computers use this approach and leave the responsibility of handling the error to the user.
    * The second approach is modifying the result to a correct one by allocating more bits to the addends.
- Whenever an overflow occurs, only one more bit extension to operands is needed to express the overflowed number. However, due to the fixed data width of computers, the data width is usually extended twice of the data width, i.e., if a single precision computation is overflowed, a double precision (twice the data width) is used to correct the error.
- Notice that the positive number is extended by appending **0**'s, while the negative number is extended by appending **1**'s to the MSB of the number.

★ Overflow condition ★

Signed subtraction with no overflow

Example (Signed subtraction with no overflow)

Compute **0100** − **0011** =?

Solution

- Compute the **2**'s complement of **0011** $\Longrightarrow$

Binary	**1**'s complement	**2**'s complement
0011	**1100**	**1101**

- Add the complemented number to the minuend: $\Longrightarrow$

Carry	**1**	**1**			
(+4)		**0**	**1**	**0**	**0**
(−3)		**1**	**1**	**0**	**1**
+1	**1**	**0**	**0**	**0**	**1**

- Check overflow. Since the signs of the addends are different, there is no overflow. Simply discard the MSB carry-out bit and the correct answer of this computation is **0001**.

★ Overflow condition ★
Signed subtraction with no overflow

Example (Signed subtraction with no overflow)

Compute **0100** − **0111** =?

Solution

- Compute the **2**'s complement of **0111** $\Longrightarrow$

Binary	**1**'s complement	**2**'s complement
0111	**1000**	**1001**

- Add the complemented number to the minuend: $\Longrightarrow$

	Carry			
(+4)	**0**	**1**	**0**	**0**
(−7)	**1**	**0**	**0**	**1**
−3	**1**	**1**	**0**	**1**

- Check overflow. Since the signs of the addends are different, there is no overflow. Simply discard the MSB carry-out bit and the correct answer of this computation is **1101**.
- if we want to find it's representation in decimal number, we can take it's **2**'s complement or we can simply do the following $\mathbf{1101} = \mathbf{-8} + \mathbf{4} + \mathbf{1} = \mathbf{-3}$.

Overflow condition

Signed subtraction with overflow correction

Example (Signed subtraction with overflow correction)

Compute **0110** − **1101** =?

Solution

- Compute the **2**'s complement of **1101** $\Longrightarrow$

Binary	**1**'s complement	**2**'s complement
1101	**0010**	**0011**

- Add the complemented number to the minuend: $\Longrightarrow$

Carry	**1**	**1**		
(+6)	**0**	**1**	**1**	**0**
(+3)	**0**	**0**	**1**	**1**
+9	**1**	**0**	**0**	**1**

- Check overflow. Since the sign of sum is different from the sign of addends, overflow has occurred. Therefore, the computational result **1001** is incorrect. Report an overflow error message.
- If we wish to obtain a correct answer instead of just giving an overflow error message. We can extend it by appending **0**'s. In this example, we shall extend the computation to a double-precision (**8**-bit in this case) arithmetic.

Overflow condition

Signed subtraction with overflow correction

Example (Signed subtraction with overflow correction)

Compute **1101** − **0111** =?

Solution

- Compute the **2**'s complement of **0111** $\Longrightarrow$

Binary	**1**'s complement	**2**'s complement
0111	**1000**	**1001**

- Add the complemented number to the minuend: $\Longrightarrow$

Carry	**1**			**1**	
(−3)		**1**	**1**	**0**	**1**
(−7)		**1**	**0**	**0**	**1**
−10	**1**	**0**	**1**	**1**	**0**

- Since the final two addends are negative, the correct answer is obtained by appending ones to the MSB side until all the extended bits are filled Therefore, the correct answer is **1111 0110** $= -128 + 64 + 32 + 16 + 4 + 2 = -10$.

1 How to represent negative numbers in binary?
Sign and Magnitude Method
Complement Number System Method
Car Odometer
1's complement
2's complement
Summary

2 Sign Extension

3 Overflow condition
Definition
Signed subtraction
Signed subtraction with no overflow
Signed subtraction with overflow correction
Unsigned Addition/Subtraction
Unsigned Addition

★ Overflow condition ★

Unsigned Addition

- In an unsigned number system, all numbers are considered positive. For instance, four bits in binary represent positive numbers from $\mathbf{0}_{10}$ to $\mathbf{15}_{10}$.
- This approach uses the single bit assigned for sign representation as a part of the magnitude, and thus twice the magnitude of the signed representation is achieved.
- Since all numbers are positive in unsigned numbers, the two addends are always positive. Hence, an unsigned overflow condition occurs only if the computation produces a carry-out at the MSB of the allocated bit.
- The computation must be carried out using normal addition rules, but if an unsigned-overflow condition is detected, the correct answer is obtained by simply appending zeros to the MSB side of the extended bits.

★ Overflow condition ★

Unsigned Addition

Example

Carry	**1**			**1**	
		1	**1**	**0**	**1**
+		**1**	**0**	**0**	**1**
	1	**0**	**1**	**1**	**0**

Carry-out exists. An unsigned-overflow has occurred. The correct answer in double precision is **0001 0110**.

Example

Carry	**1**			**1**	
		0	**1**	**1**	**0**
+		**0**	**1**	**0**	**1**
	1	**0**	**1**	**1**	

No carry-out exists, so the result is correct. The correct answer is $\mathbf{1011}_2$ or $\mathbf{11}_{10}$ in decimal. However, notice that if it was signed computation, it generates an overflow error.

★ Overflow condition ★

Unsigned subtraction

- In unsigned subtraction, the minuend must be larger than the subtrahend. Otherwise, the result would become negative, which violates the definition of unsigned computation.
- If the subtracted result is actually negative, an occurrence of error should be indicated. In a computer implementation, this error condition is shown through a borrow bit.
- If the borrow bit is set, it means that the minuend is smaller than the subtrahend and indicates an error condition for unsigned computation.

★ Overflow condition ★
Hexadecimal number

Example

Imagine the following calculations made on **16**-bit numbers.

$$\begin{array}{rcccc} & F & F & F & F \\ + & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array} \qquad \begin{array}{rcccc} & 8 & 0 & 1 & 2 \\ + & F & 0 & 9 & 1 \\ \hline 1 & 7 & 0 & A & 3 \end{array}$$

$$\begin{array}{rcccc} & 5 & 1 & A & 4 \\ + & 6 & 2 & C & 1 \\ \hline & B & 3 & B & 0 \end{array} \qquad \begin{array}{rcccc} & 2 & 4 & 1 & C \\ + & 3 & 0 & A & 5 \\ \hline & 5 & 4 & C & 1 \end{array}$$

Determine whether there is any overflow. If there is an overflow, then discuss how to avoid it and find the values in Hexadecimal number:

1. If it is an unsigned number?
2. If it is a signed number?

Solution

a)
$$\begin{array}{cccccc} & & F & F & F & F \\ + & & 0 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 & 0 \end{array}$$
there is a carry that comes out of the **16** bits and the two numbers to be added have different signs.

① Carry-out exists. An unsigned-overflow has occurred. The correct answer in double precision **32** bits is $\mathbf{(0\,0\,0\,1\,0\,0\,0\,0)_{16}}$.
② The signs of the addends are different, there is no overflow. Simply discard the MSB carry-out bit and the correct answer of this computation is $\mathbf{(0\,0\,0\,0)_{16}}$.

b)
$$\begin{array}{cccccc} & & 8 & 0 & 1 & 2 \\ + & & F & 0 & 9 & 1 \\ \hline & 1 & 7 & 0 & A & 3 \end{array}$$
there is a carry that comes out of the **16** bits and the sign of sum is different from the sign of addends.

① Carry-out exists. An unsigned-overflow has occurred. The correct answer in double precision **32** bits is $\mathbf{(0\,0\,0\,1\,7\,0\,A\,3)_{16}}$.
② The sign of sum is different from the sign of addends, overflow has occurred. Therefore, the result is incorrect. Report an overflow error message. We can extend it to a double-precision **32** bits. $\mathbf{(1\,1\,1\,1\,7\,0\,A\,3)_{16}}$.

Solution

c)
$$\begin{array}{cccccc} & 5 & 1 & A & 4 \\ + & 6 & 2 & C & 1 \\ \hline & B & 3 & B & 0 \end{array}$$
there is no carry that comes out of the **16** bits and the sign of sum is different from the sign of addends.

① No carry-out exists. The correct answer is $(\boldsymbol{B\ 3\ B\ 0})_{16}$.
② The sign of sum is different from the sign of addends, overflow has occurred. Therefore, the result is incorrect. Report an overflow error message. We can extend it to a double-precision **32** bits. $(\boldsymbol{0\ 0\ 0\ 0\ B\ 3\ B\ 0})_{16}$

d)
$$\begin{array}{cccccc} & 2 & 4 & 1 & C \\ + & 3 & 0 & A & 5 \\ \hline & 5 & 4 & C & 1 \end{array}$$
there is no carry that comes out of the **16** bits and the sign of sum is the same as the sign of addends.

① No carry-out exists. The correct answer is $(\boldsymbol{5\ 4\ C\ 1})_{16}$.
② The sign of sum is the same as the sign of addends, there is no overflow. The correct answer is $(\boldsymbol{5\ 4\ C\ 1})_{16}$.

✎ **Exercise 11.** Perform the following binary addition in **5**-bit **2**'s complement arithmetic. Determine whether there is any overflow. If there is an overflow, then discuss how to avoid it.

a. $-7+10$ b. $-10+7$ c. $10+7$ d. $-10+-7$

✎ **Exercise 12.** Describe what conditions indicate overflow has occurred when two **2**'s complement numbers are added

a. When adding two numbers.

b. When the operands have differing leftmost bits.

✎ **Exercise 13.** The following binary numbers are **4**-bit **2**'s complement binary numbers. Which of the following operations generate overflow? Justify your answers by translating the operands and results into decimal.

a. $0011+1100$ b. $0111+1111$ c. $1110+1000$ d. $0110+0010$


Thank you! Questions?