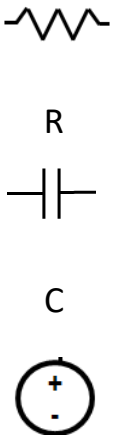
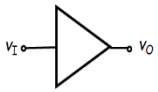
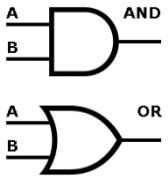
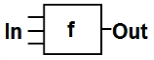
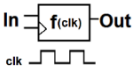
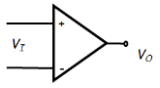
What is Engineering?

- The Purposeful Use of Science
- Providing an understanding of natural phenomenon
  - How? By **Abstraction** !!!

ECE Layers by Abstraction

Natures	Physics laws	Lumped circuit Abst.	Amplifier		Logic gates	Combinational logic	Clocked Digital Abst.	Instruction set Architecture (ISA)	Computer language	**SW Abst. starts**
Measuring using probes V I 3 0.1 6 0.2 9 0.3	V= IR Maxwell Eqs.	 Voltage source	 More interesting components without considering MEs	Digital	 (elementary bldg. blocks to inverters)	Define functional blocks 		Machine language, i.e.,X86 Instruction set (to build up micro processors)	JAVA C C++ ….	Operating systems i.e., Linux Windows
				Analog	Op-amp	Analog system components	Fun devices			Useful to human beings
						Oscillator Filter Power supply	Toasters, Power plants …..			Video games, Space ships …

☺$$

☺ $

ABSTRACTION!!!! From natures, all the way to devices
Bigger things, complicate behavior inside, but simple to describe

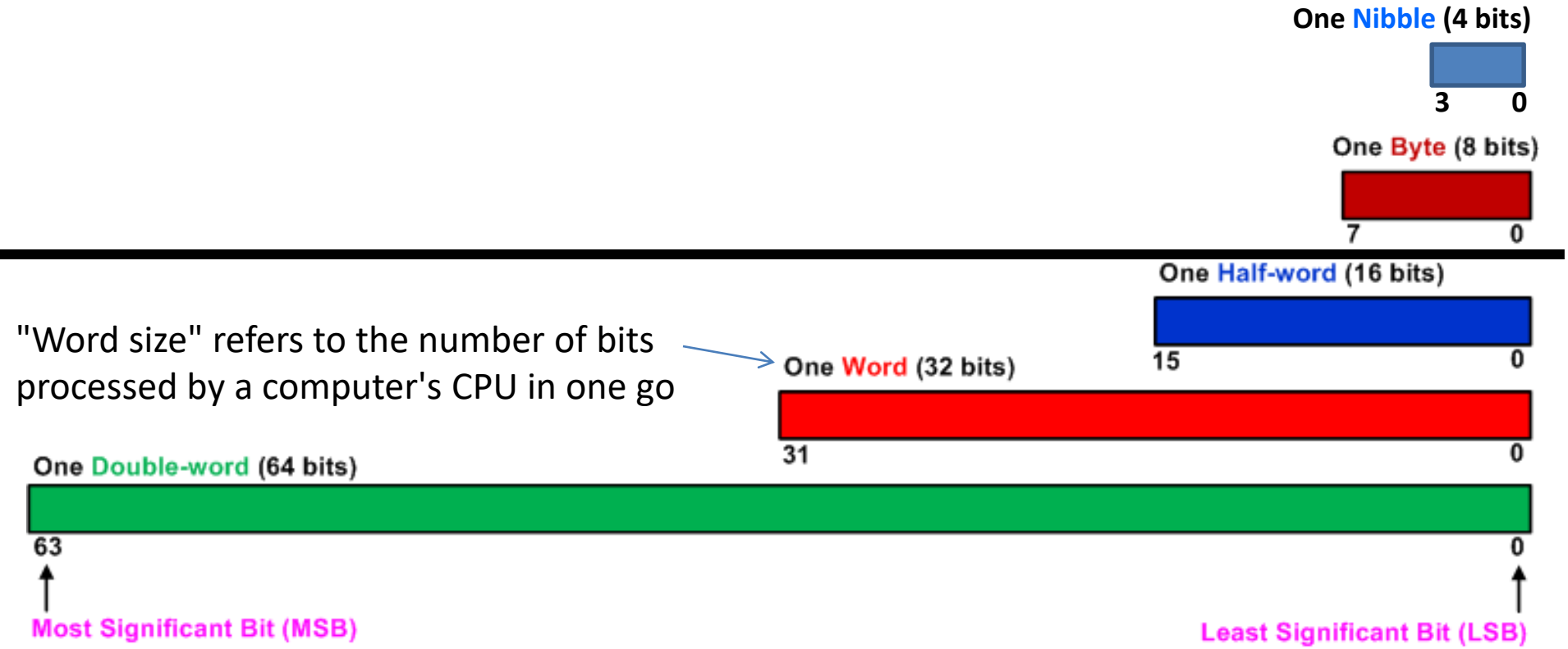
Data Representation in computers

Dr. Noori Kim
(noori.kim@digipen.edu)

Agenda

1. Negative numbers (signed integers) in computers
2. Four special condition flags with Two's complement implementation

Recap: Data size and naming



1. Negative numbers (signed integers) in computers

Negative numbers in computers

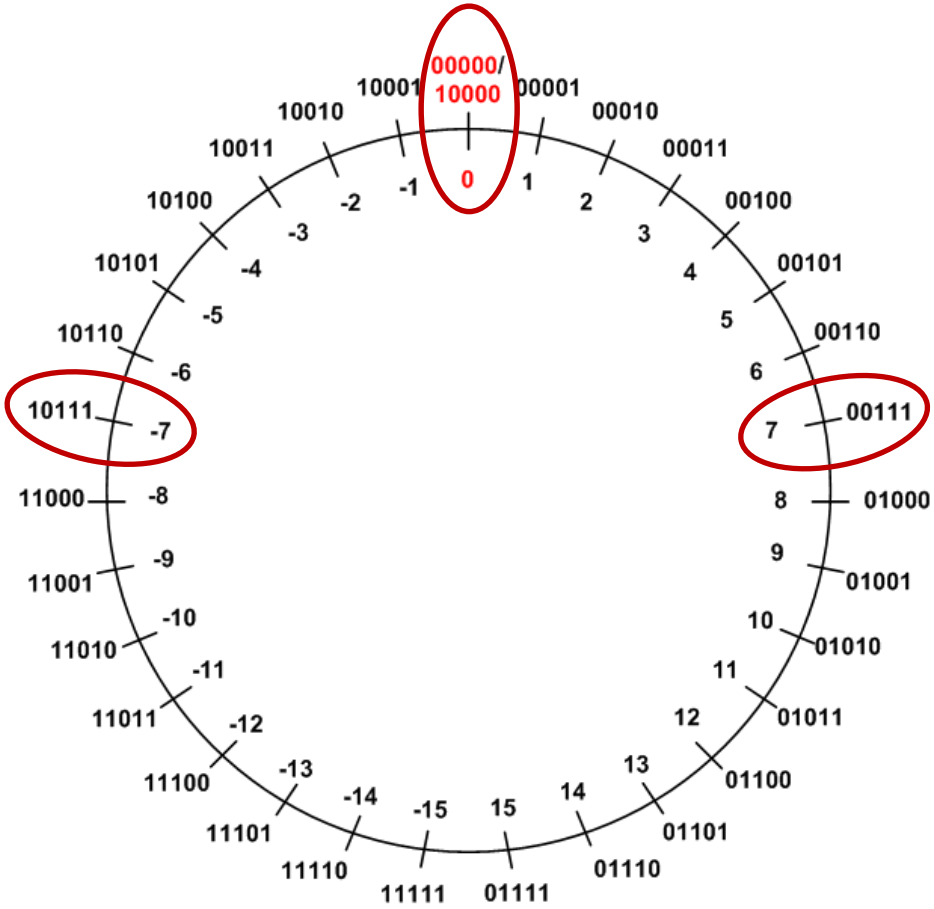
1. Sign magnitude method: a sign bit method
   - Ex) 1001: -1, 0001: 1
   - Problem: +0 (0000) and -0 (1000)

Sign-and-Magnitude:

$$value = (-1)^{sign} \times Magnitude$$

- The most significant bit is the sign.
- The rest bits are magnitude.

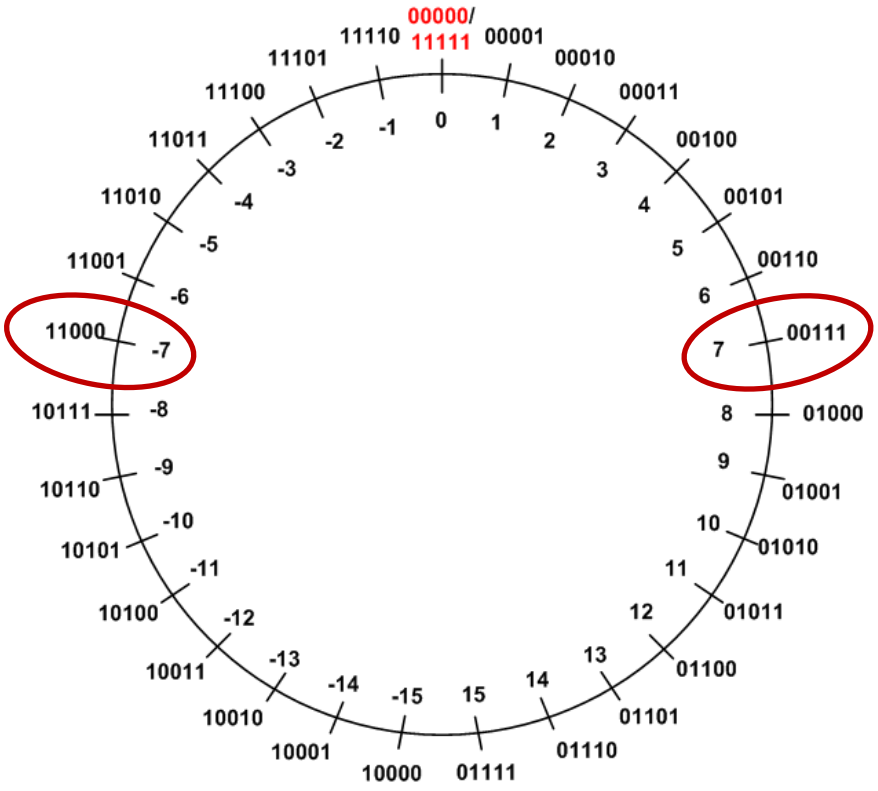
- Example: in a 5-bit system
  - $+7_{10}$ = 00111_2
  - -7_{10} = 10111_2
- Two ways to represent zero
  - $+0_{10}$ = 00000_2
  - -0_{10} = 10000_2
- Not used in modern systems
  - Hardware complexity
  - Two zeros



2. 1’s complement method
   - Ex) 1000: -7, 0111: 7
   - Problem: +0 (0000) and -0 (1111)

One's Complement ($\tilde{\alpha}$):

$$\alpha + \tilde{\alpha} = 2^n - 1$$



The one's complement representation of a negative binary number is the bitwise NOT of its positive counterpart.

Example: in a 5-bit system

$+7_{10} = 00111_2$

$-7_{10} = 11000_2$

$+7_{10} + (-7_{10}) = 00111_2 + 11000_2$

$= 11111_2$

$= 2^5 - 1$

Seldom used in modern systems

- Two zeros

3. 2's complement method
   - Most computer nowadays
   - Ex) 0111:7, 1001:-7
   - Only one type of zero
   - 2's complement operation: inverting bits and adding 1
     - Ex) compute two's complement of 0111 => 1001
     - Ex) compute a decimal value of a signed integer 1000
       - $1*(-2^3)=-8$
       - (two's complement) 0111+1=1000=8, therefore -8

Complement?

Complement (set theory)

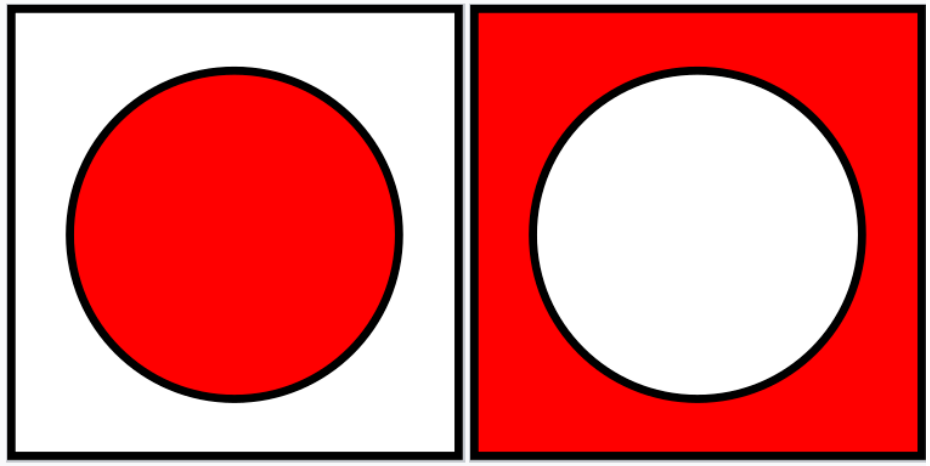
From Wikipedia, the free encyclopedia

In set theory, the **complement** of a set A refers to elements not in A.

When all sets under consideration are considered to be subsets of a given set U, the **absolute complement** of A is the set of elements in U but not in A.

The **relative complement** of A with respect to a set B, also termed the **difference** of sets A and B, written $B \setminus A$, is the set of elements in B but not in A.

Contents [hide]

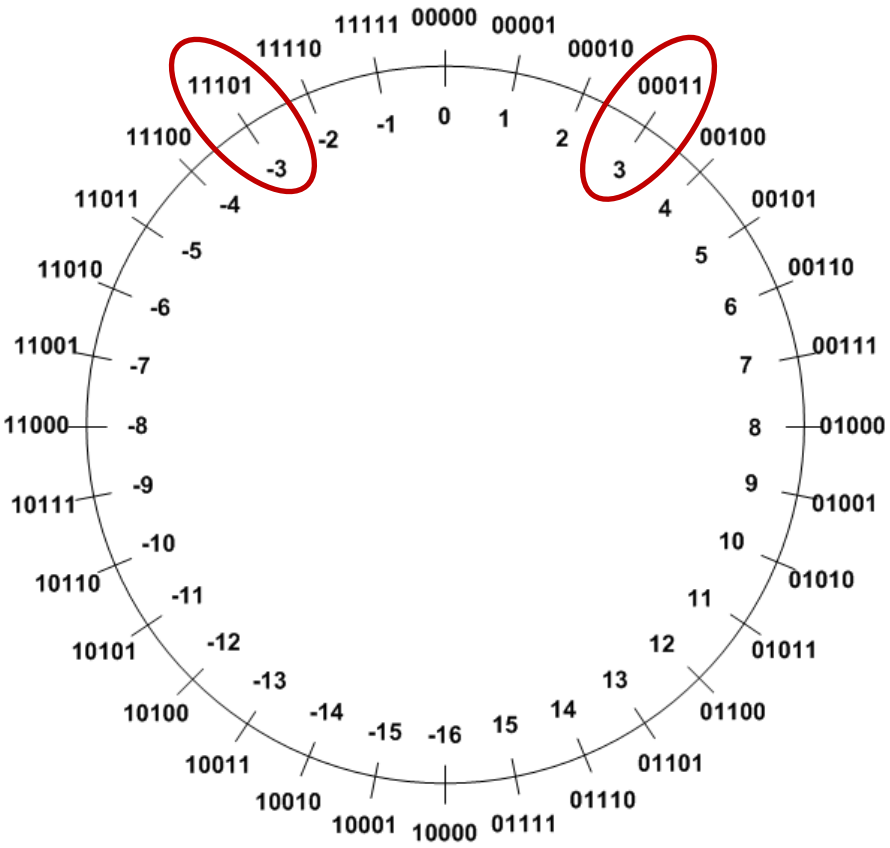


If A is the area colored red in this image... ... then the complement of A is everything else.

- Two’s complement of 0000 and 1000: themselves
- Number pairs with Two’s complement relationship: Sum of them are 0

Two's Complement, TC, $(\overline{\alpha})$:

$$\alpha + \overline{\alpha} = 2^n$$



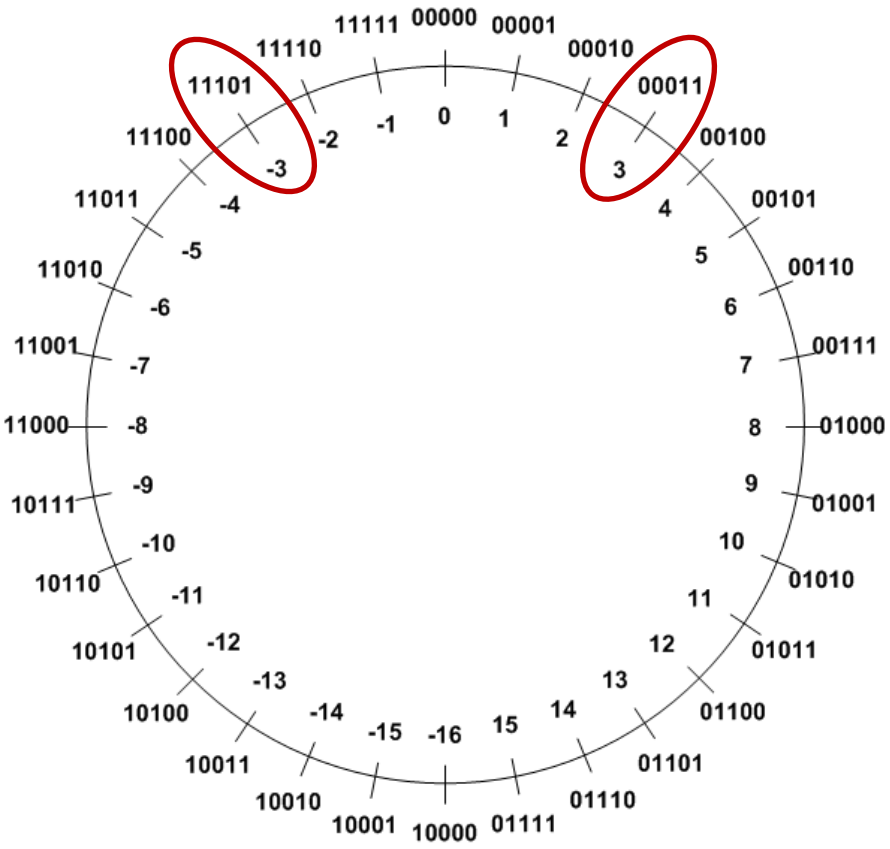
TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.

Example **1**: TC(3)

	Binary	Decimal
Original number	00011	3
Step 1: Invert every bit	11100	
Step 2: Add 1	+ 00001	
Two's complement	11101	-3

Two's Complement (TC)

$$\alpha + \overline{\alpha} = 2^n$$

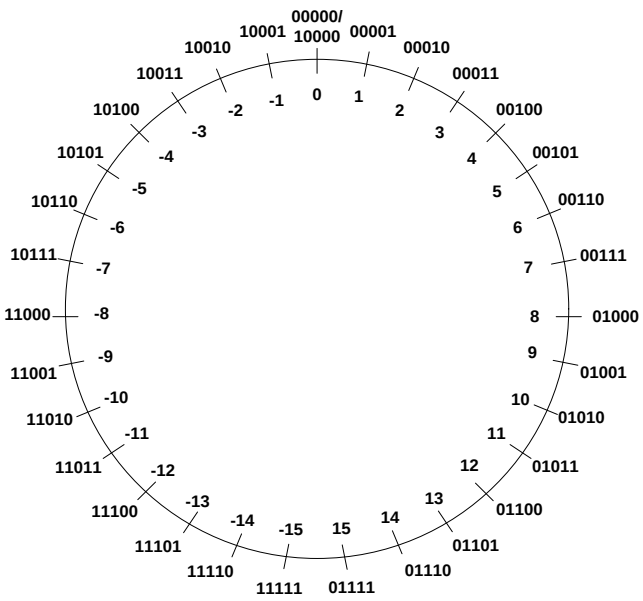


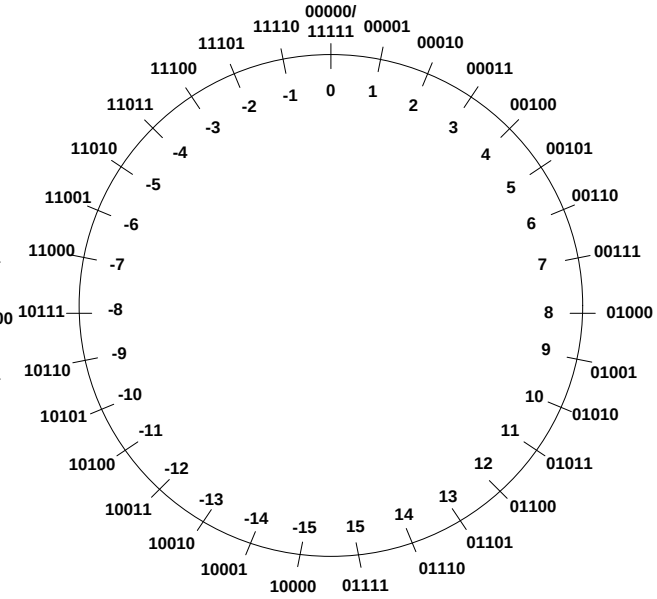
TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.

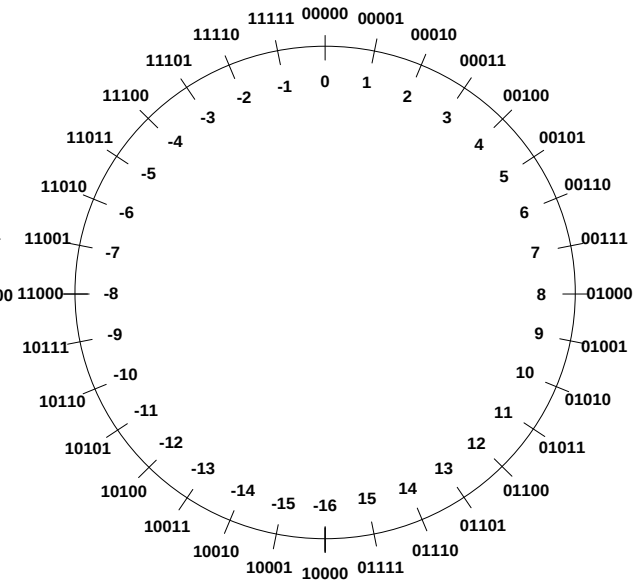
Example 2: TC(-3)

	Binary	Decimal
Original number	11101	-3
Step 1: Invert every bit	00010	
Step 2: Add 1	+ 00001	
Two's complement	00011	3

Comparison







Signed magnitude representation

0 = positive

1 = negative

One's complement representation

Negative = invert all bits of a positive

Two's Complement representation

TC = invert all bits, then plus 1

In summary,

negative numbers

- Three ways to represent signed binary integers:
  - Signed magnitude
    - $\boldsymbol{value} = (-\mathbf{1})^{\boldsymbol{sign}} \times \boldsymbol{Magnitude}$
  - One's complement ($\tilde{\alpha}$)
    - $\alpha + \tilde{\alpha} = \mathbf{2}^{\boldsymbol{n}} - \mathbf{1}$
  - Two's complement ($\bar{\alpha}$)
    - $\alpha + \bar{\alpha} = \mathbf{2}^{\boldsymbol{n}}$

	Sign-and-Magnitude	One's Complement	Two's Complement
Range	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$
Zero	Two zeroes (± 0)	Two zeroes (± 0)	One zero

2. Four special condition flags with Two’s complement implementation

Four special conditions

Most of modern computers are using 2's complement method to represent signed integers,

- To operate the computers efficiently, they place 4 flags to indicate conditions of a number. What would be these four special conditions?

Condition Codes (with 4-bit examples)

Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

- C is set after an **unsigned** addition if the answer is "out of range"
  - i.e., $1000_2+1000_2 =? 0000_2$
- C is set after an subtract when borrow is not occurred
  - i.e., 1111_2-0111_2 ➔ $1111_2+1001_2 = 1000_2$: C bit sets
- V is set after a **signed** add. or subt. if the answer is wrong (just compare MSB)
  - i.e., 1111_2+1111_2 ➔ 1110_2: V bit clears (negative + negative must be negative, correct answer)
  - i.e., 1000_2-0111_2 ➔ $1000_2+1001_2 =? 0001_2$: V bit sets (negative - positive must be negative, wrong answer)

An interesting question

How does an assembler in a computer distinguish between a signed number and an unsigned number?

Carry/borrow flag bit for numbers

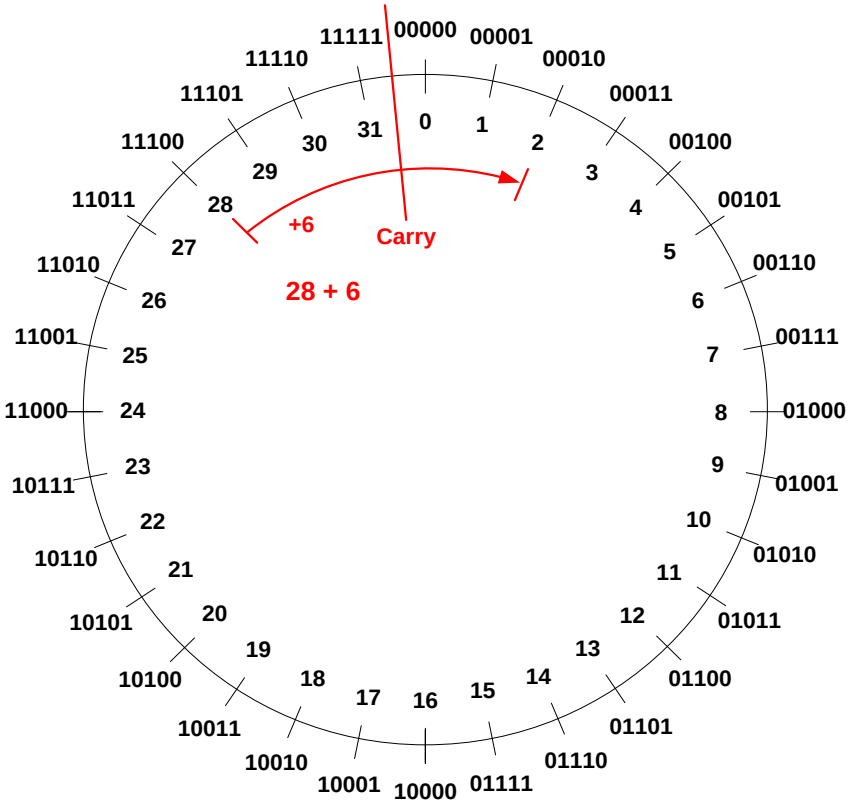
- When adding two numbers in an *n*-bit system, a carry occurs if the result is larger than the maximum unsigned integer that can be represented (*i.e.* $2^n - 1$).

- When subtracting two numbers, borrow occurs if the result is negative, smaller than the smallest unsigned integer that can be represented (*i.e.* 0).

- On ARM Cortex-M3 processors, the carry flag and the borrow flag are physically the same flag bit in the status register.
  - For an unsigned subtraction, Carry = NOT Borrow

Carry/borrow flag bit for numbers

If the traverse crosses the boundary between 0 and $2^n - 1$, the carry flag is set on addition and is cleared on subtraction.



A carry occurs when adding 28 and 6

Carry	1	1	1	0	0	
	1	1	1	0	0	28
+	0	0	1	1	0	+ 6
1	0	0	0	1	0	2

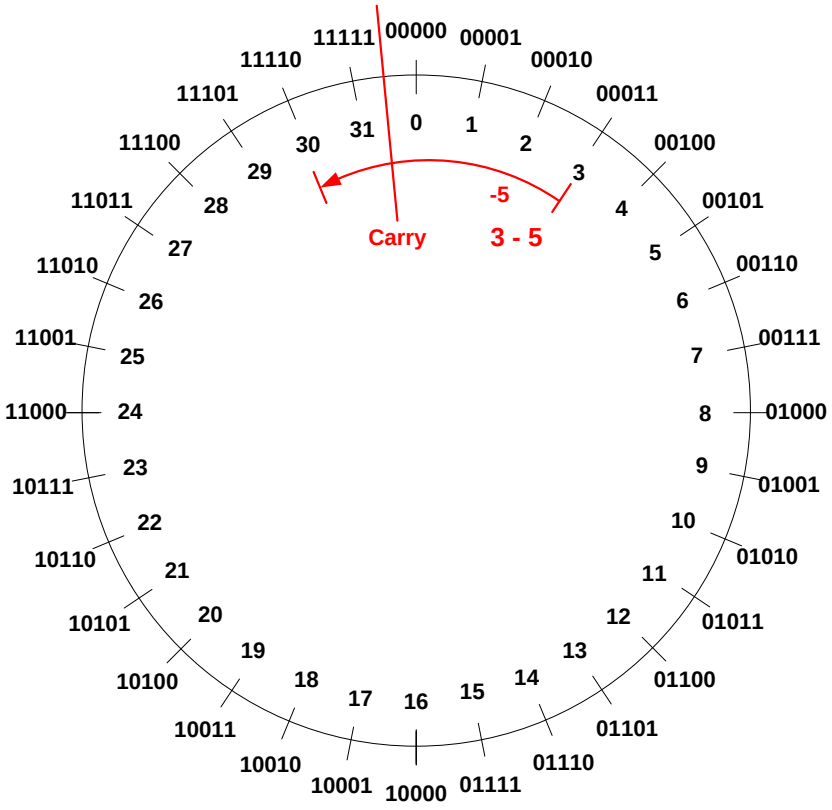
Extra bit is discarded.

5-bit result

- Carry flag is 1 because the result crosses the boundary between 31 and 0.

Carry/borrow flag bit for numbers

If the traverse crosses the boundary between 0 and $2^n - 1$, the carry flag is set on addition and is cleared on subtraction.



Borrow	1	1	1	0	0		
		0	0	0	1	1	3
-		0	0	1	0	1	- 5
		1	1	1	1	0	30

5-bit result

- Carry flag = 0, indicating borrow has occurred on unsigned subtraction.
- For subtraction, carry = NOT borrow.

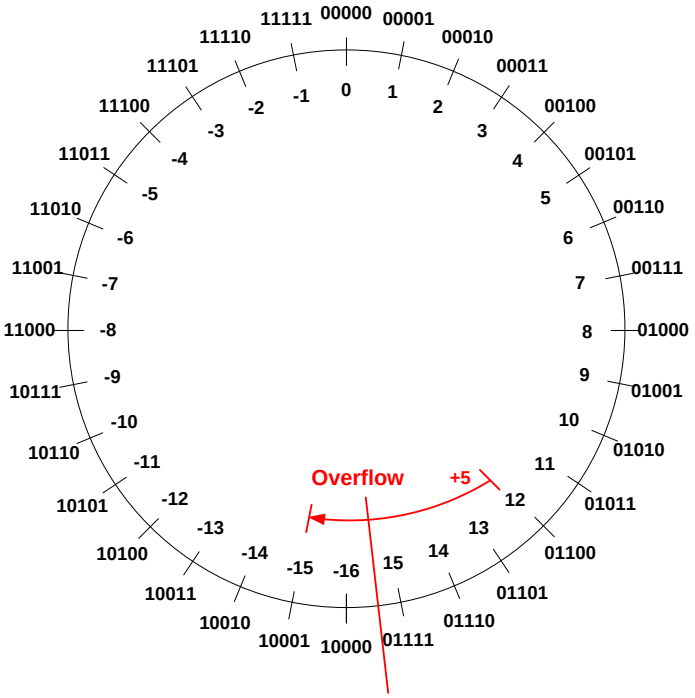
A borrow occurs when subtracting 5 from 3.

Overflow flag for signed numbers

- When adding signed numbers represented in two's complement, overflow occurs only in two scenarios:
  1. adding two positive numbers but getting a non-positive result, or
  2. adding two negative numbers but yielding a non-negative result.

- Similarly, when subtracting signed numbers, overflow occurs in two scenarios:
  1. subtracting a positive number from a negative number but getting a positive result, or
  2. subtracting a negative number from a positive number but producing a negative result.

- Overflow cannot occur when adding operands with different signs or when subtracting operands with the same signs.

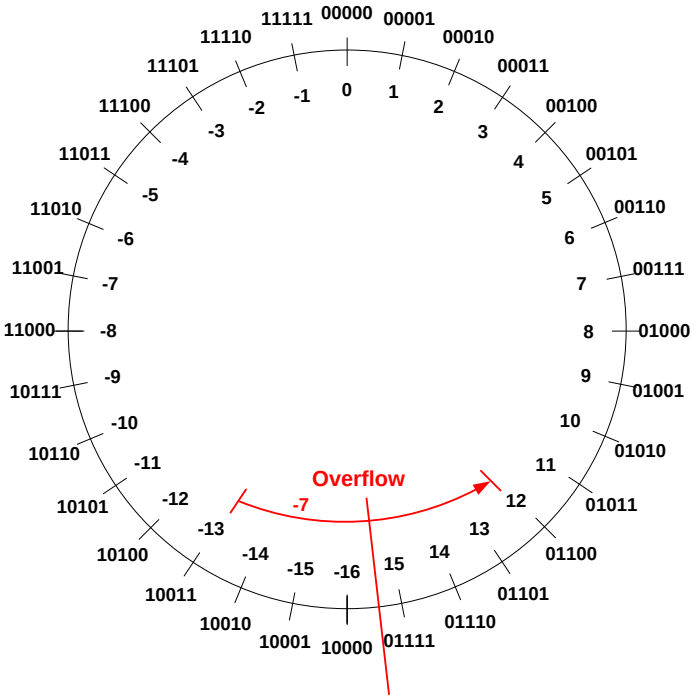


An overflow occurs when adding two positive numbers and getting a negative result.

```
  0 1 1 0 0      12
+ 0 0 1 0 1    + 5
-------------  ----
  1 0 0 0 1     -15
```

5-bit result

1. On addition, overflow occurs if $sum \geq 2^4$ when adding two positives.
2. Overflow never occurs when adding two numbers with different signs.



An overflow occurs when adding two negative numbers and getting a positive result.

```
  1 0 0 1 1      -13
+ 1 1 0 0 1    + -7
-------------  -----
1 0 1 1 0 0      12
```

Extra bit is discarded.

5-bit result

On addition, overflow occurs if $sum < -2^4$ when adding two negatives.

Exercise with various formats assuming Two's Complement hardware implementation

Assume a four-bit system:

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110			
0100 + 0110	1010			
1100 + 1110	1010			
1100 + 1010	0110			

Assume a four-bit system:

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
adds    r0, r1, r2
```

N = 0 The result is 0, which is considered positive

Z = 1 The result is 0, so the Z (zero) bit is set to 1.

C = 1 We lost some data because the result did not fit into 32 bits, so the processor indicates this by setting C (carry) to 1.

V = 0 From a two's complement signed-arithmetic viewpoint; (-1) + 1 = 0.
Negative + positive = can be positive (zero); check MSB

```
ldr     r1, =0x80000000
ldr     r2, =0x80000000
adds    r0, r1, r2
```

N = 0 The result is 0, which is considered positive

Z = 1 The result is 0, so the Z (zero) bit is set to 1.

C = 1 We lost some data because the result did not fit into 32 bits, so the processor indicates this by setting C (carry) to 1.

V = 1 From a two's complement signed-arithmetic viewpoint; the result says $(-2^{31}) + -2^{31} = 0$.
Negative + Negative ≠ positive (zero); check MSB

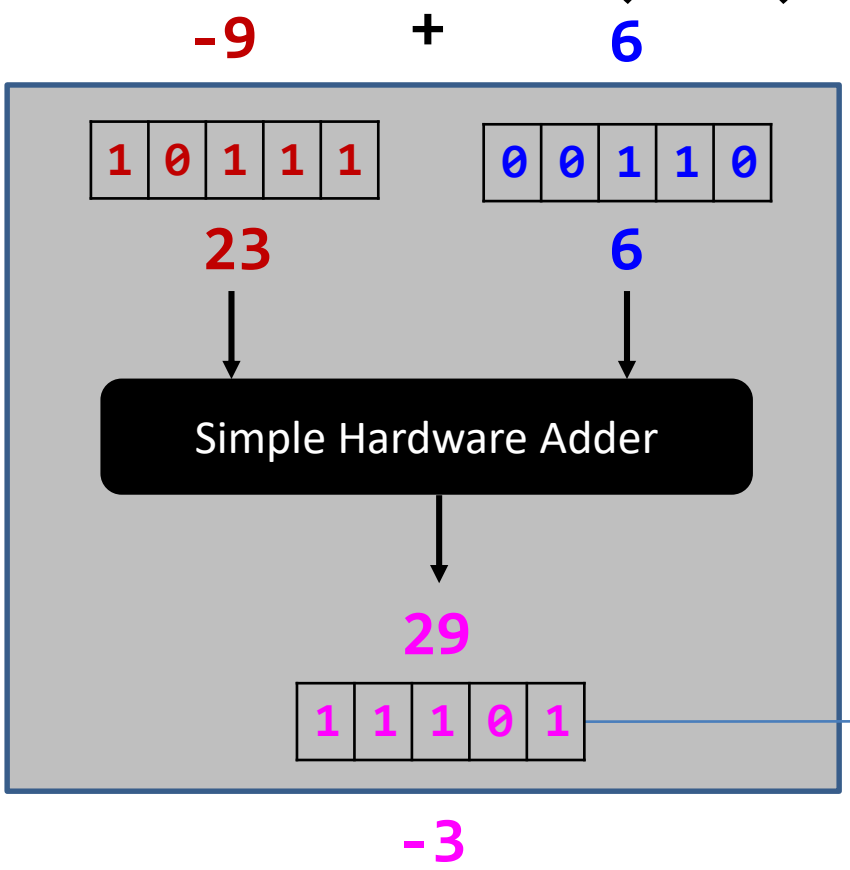
Why use Two's Complement

Two's complement simplifies hardware

Operation	Are signed and unsigned operations the same?
Addition	Yes
Subtraction	Yes
Multiplication	Yes if the product is required to keep the same number of bits as operands
Division	No

Adding two signed integers:

(-9) + 6



9
01001 → flip → 10110 → +1 → **-9** 10111

Two's Complement

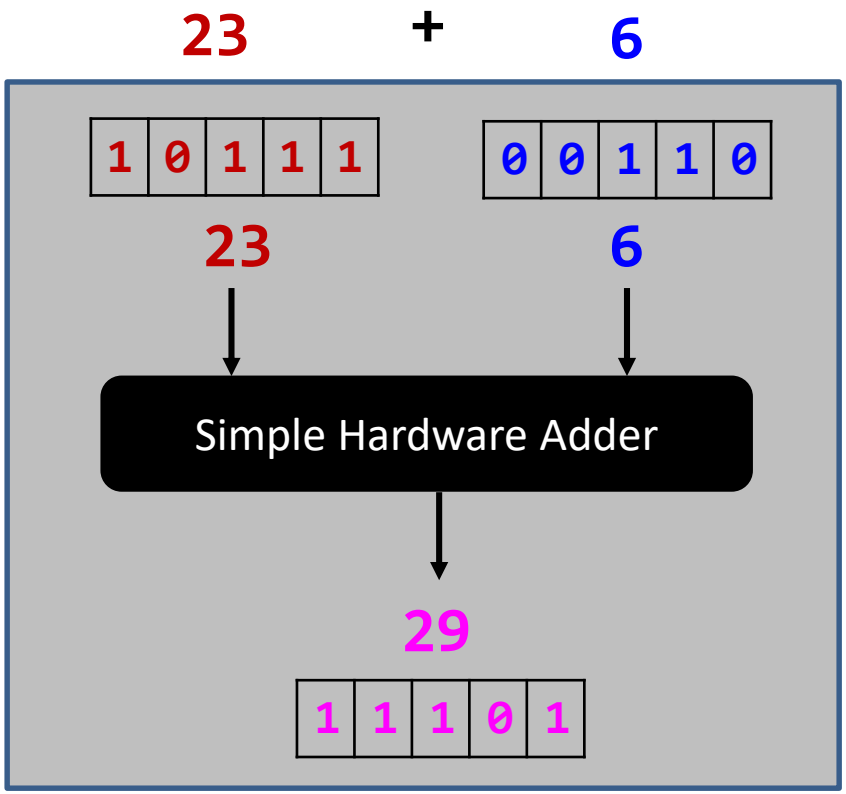
flip → 00010 → +1 → **3** 00011

Two's Complement Counterpart

As the sign bit (MSB) is 1 (N bit is set, N=1), this number (29) needs to 2's-complemented and minused before the number is printed out to screen

Adding two unsigned integers:

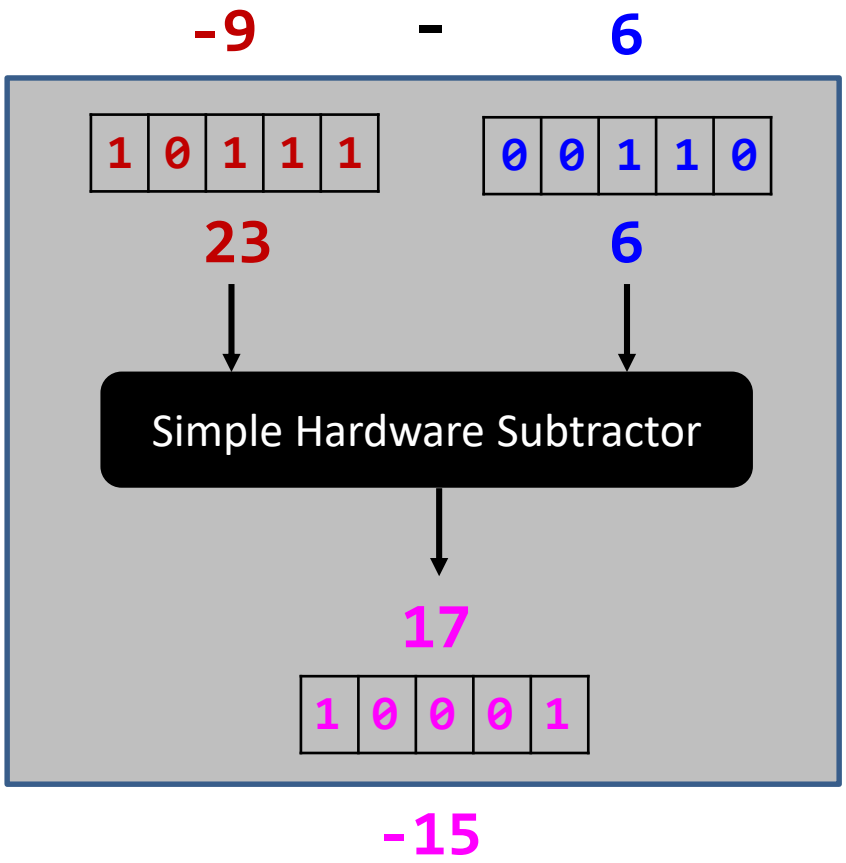
23 + 6



- Unsigned integers ➔ no sign bit.
- N bit is cleared (N=0, means not negative number)
- 29 is the answer

Subtracting two signed integers:

(-9) - 6



9
01001 flip → 10110 +1 → **-9** **10111**

Two's Complement

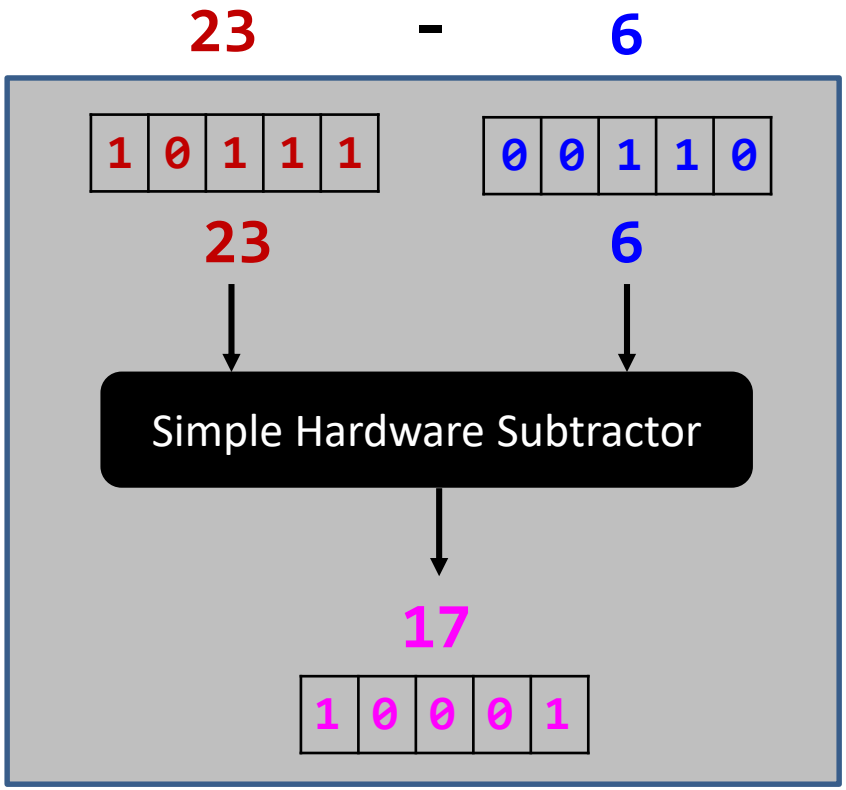
flip → 01110 +1 → **15** **01111**

Two's Complement Counterpart

As the sign bit (MSB) is 1 (N bit is set, N=1), this number (17) needs to 2's-complemented and minused before the number is printed out to screen

Subtracting two unsigned integers:

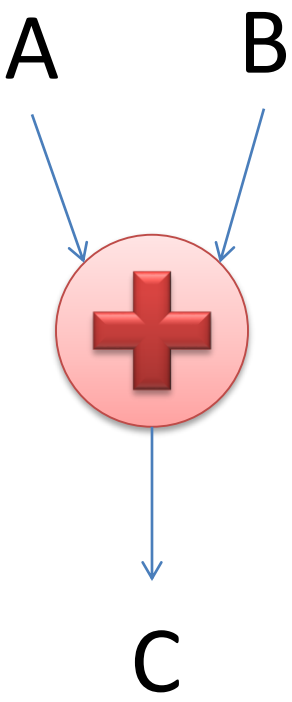
(23) - 6



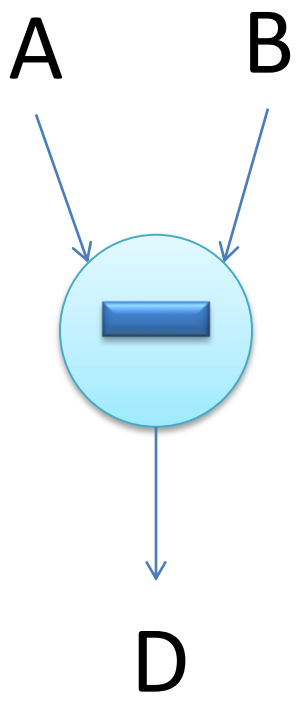
- Unsigned integers ➔ no sign bit.
- N bit is cleared (N=0, means not negative number)
- 17 is the answer

2-in-1 implementation (add/sub)

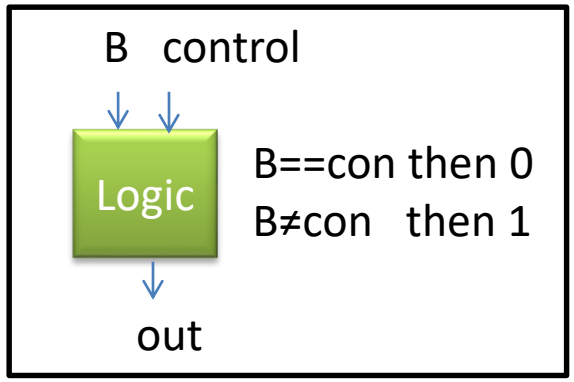
Addition (A + B = C)



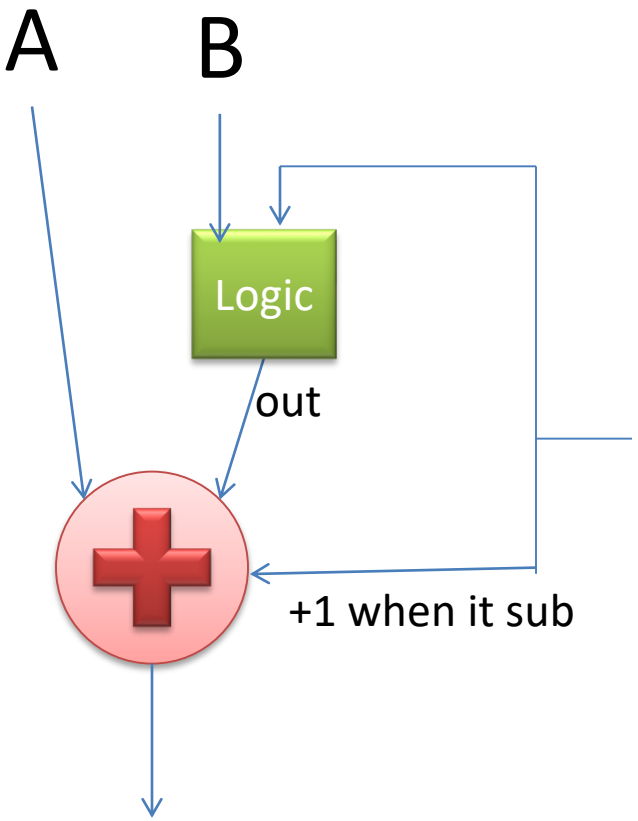
Subtraction (A - B = D)



2-in-1 implementation using 2's complement method



B	cont	out
0	0	
0	1	
1	0	
1	1	



A Control bit
(1 or 0)

- addition➔0
  ➔B is always itself (B)
- subtraction➔ 1
  ➔B is always negate ($\bar{B}$)

https://www.oldcalculatormuseum.com/comp241.html

Two's Complement Simplifies Hardware Implementation

- In two's complement, the same hardware works correctly for both signed and unsigned addition/subtraction.
  - If the product is required to keep the same number of bits as operands, unsigned multiplication hardware works correctly for signed numbers.
  - However, this is not true for division.
- Also one can implement an operator which can run add/sub in one setting.