

# Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

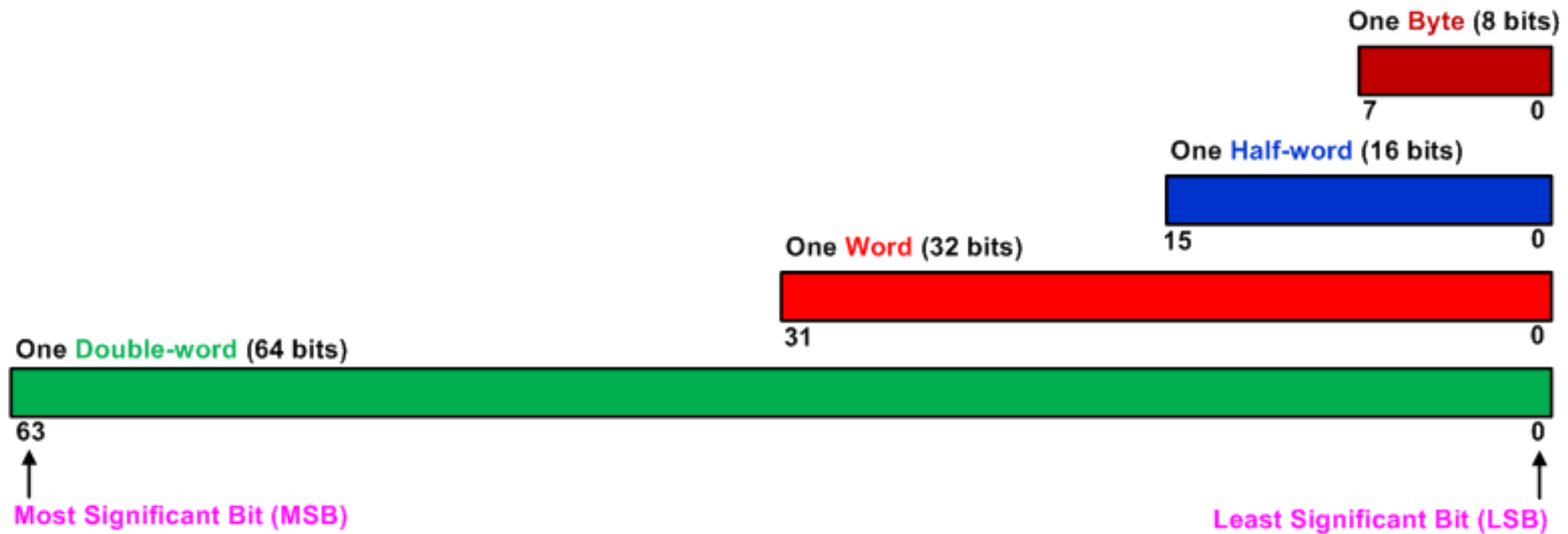
## Chapter 2 Data Representation

Zonghua Gu

Fall 2025

# Bit, Byte, Half-word, Word, Double-Word

---



# Binary, Octal, Decimal and Hex

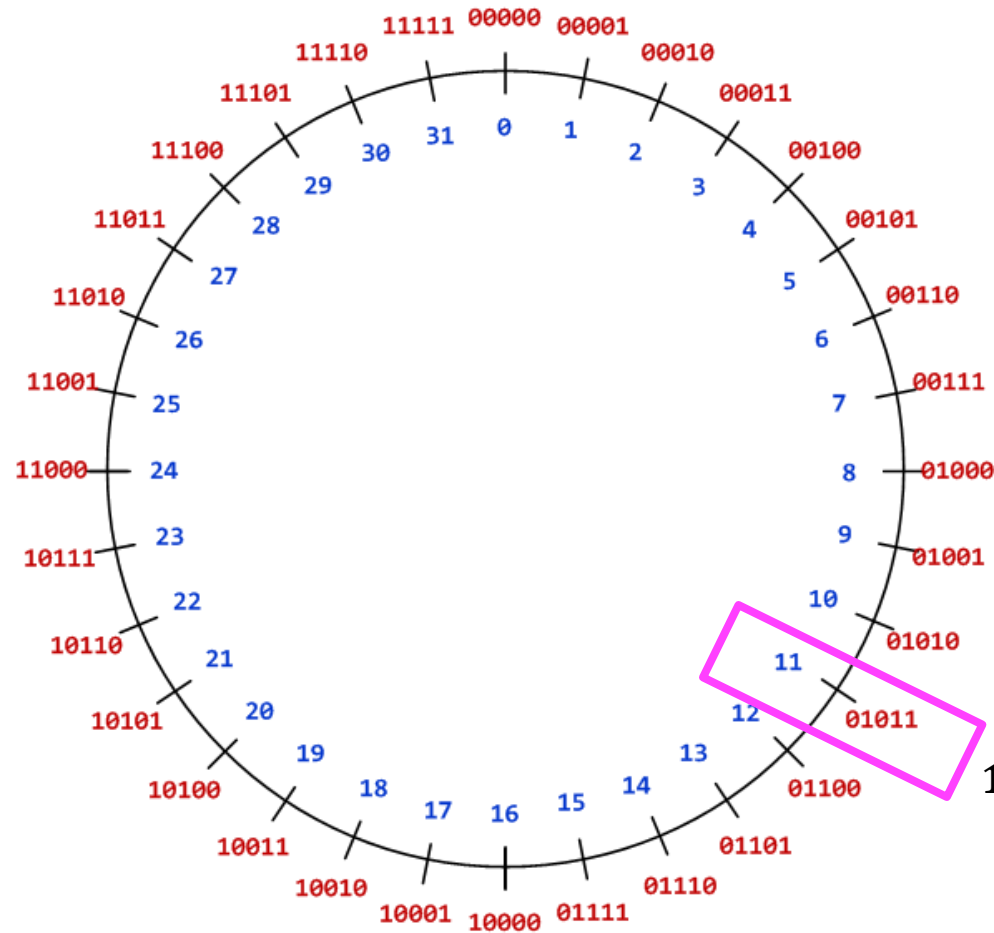
---

Decimal	Binary	Octal	Hex
0	0000	00	0x0
1	0001	01	0x1
2	0010	02	0x2
3	0011	03	0x3
4	0100	04	0x4
5	0101	05	0x5
6	0110	06	0x6
7	0111	07	0x7
8	1000	010	0x8
9	1001	011	0x9
10	1010	012	0xA
11	1011	013	0xB
12	1100	014	0xC
13	1101	015	0xD
14	1110	016	0xE
15	1111	017	0xF

0x: Hex

0: Octal

# Unsigned Integers

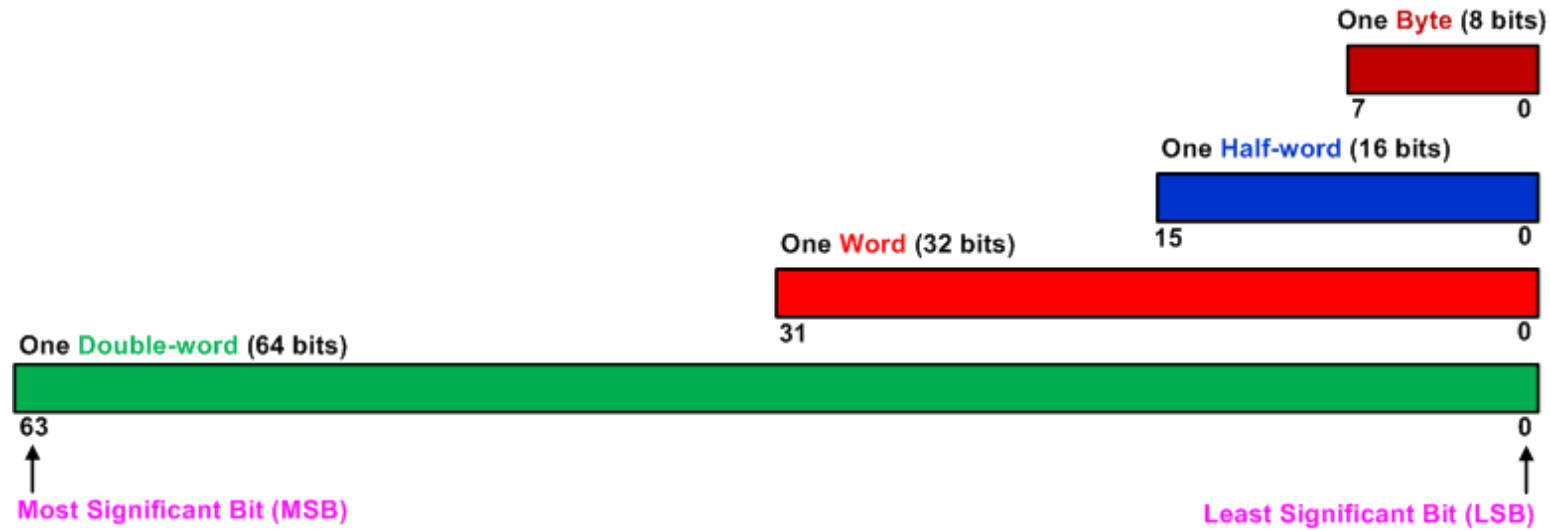


Five-bit binary integers

**Convert from Binary to Decimal:**

$$\begin{aligned} 1011_2 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 8 + 2 + 1 \\ &= 11 \end{aligned}$$

# Range of Unsigned Integers

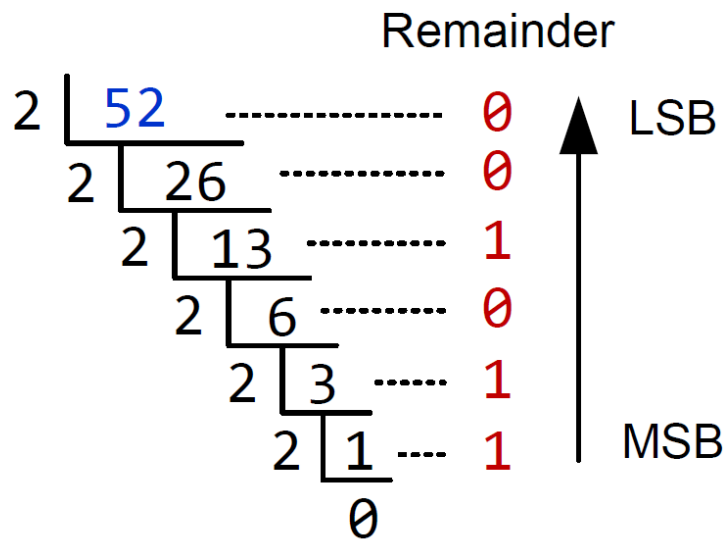


Storage Size	Range	Powers of 2
Unsigned Byte	0 to 255	0 to $2^8-1$
Unsigned Halfword	0 to 65,535	0 to $2^{16}-1$
Unsigned Word	0 to 4,294,967,295	0 to $2^{32}-1$
Unsigned Double-word	0 to 18,446,744,073,709,551,615	0 to $2^{64}-1$

# Unsigned Integers

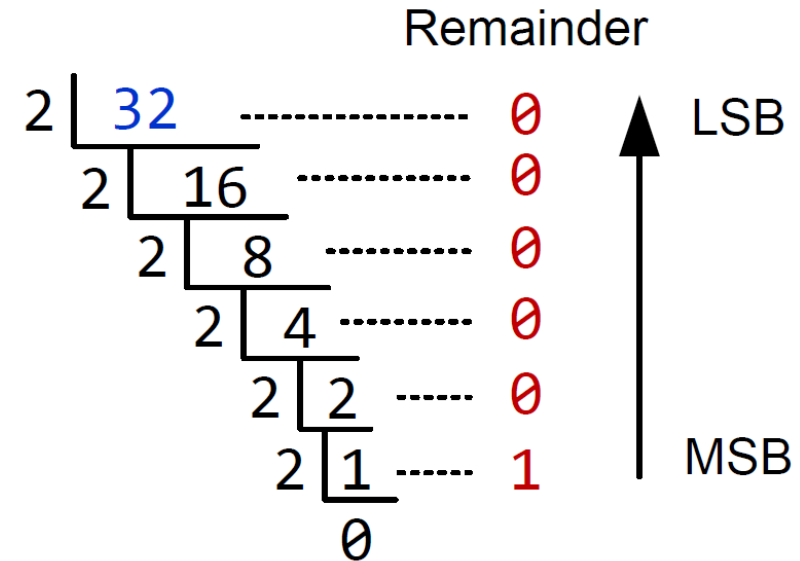
## Convert Decimal to Binary

### Example 1



$$52_{10} = 110100_2$$

### Example 2



$$32_{10} = 100000_2$$

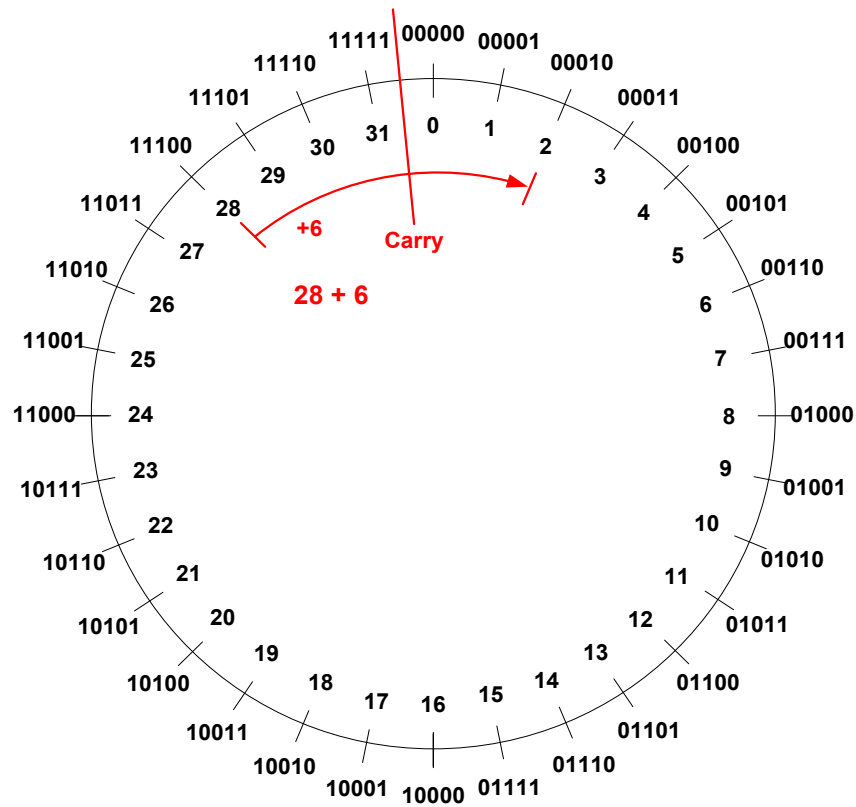
# Carry/borrow flag bit for unsigned numbers

---

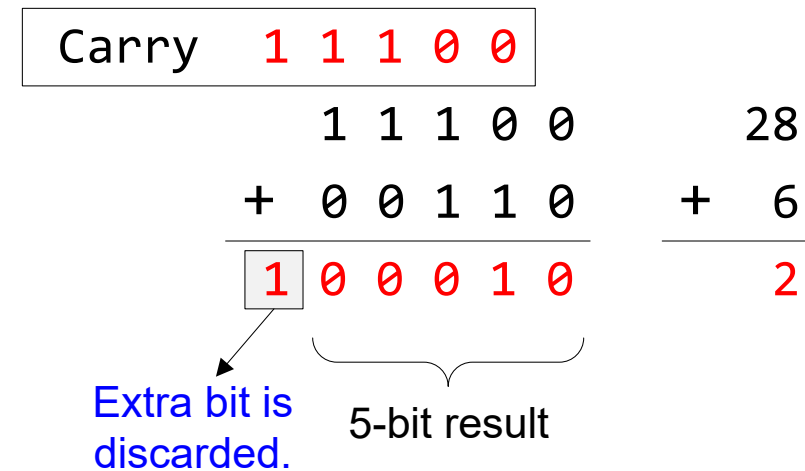
- Given **unsigned** integers  $a$  and  $b$
- $c = a + b$ 
  - Carry happens if  **$c$  is too big to fit in  $n$  bits** (i.e.,  $c > 2^n - 1$ ).
- $c = a - b$ 
  - Borrow happens if  $c < 0$ .
- On ARM Cortex-M processors, the carry flag and the borrow flag are physically the same flag bit in the status register.
  - **For an unsigned subtraction, Carry = NOT Borrow**

# Carry/borrow flag bit for unsigned 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

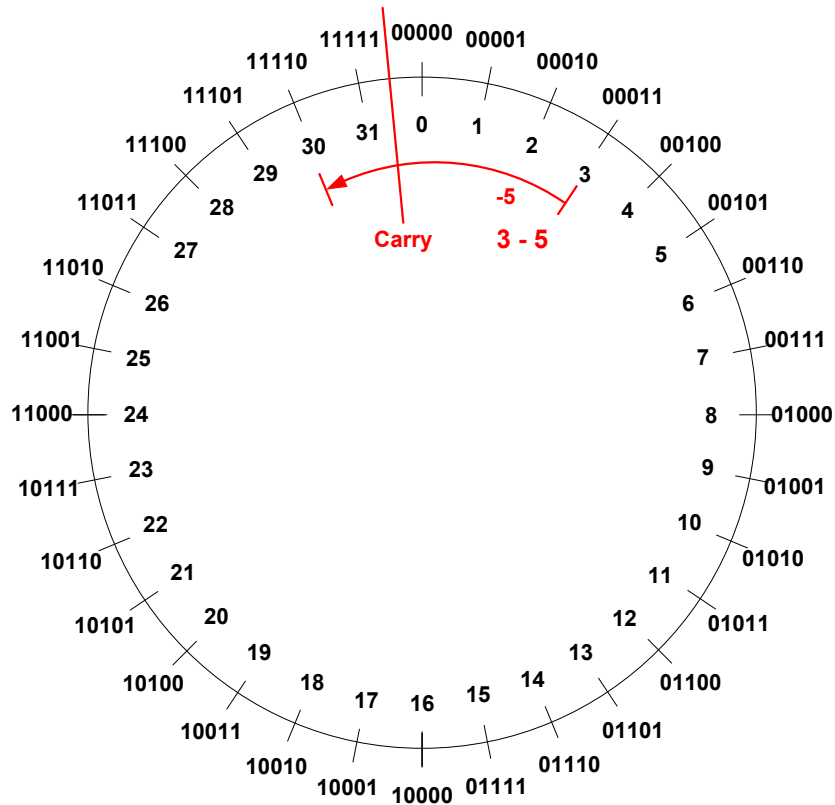


- Carry flag = 1, indicating carry has occurred on unsigned addition.
- Carry flag is 1 because the result crosses the boundary between 31 and 0.



# Carry/borrow flag bit for unsigned 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.

# Signed Integer Representation

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

	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 ( $\pm 0$ )	Two zeroes ( $\pm 0$ )	One zero
Unique Numbers	$2^n - 1$	$2^n - 1$	$2^n$

# Signed Integers

## Method 1: Signed Magnitude

### 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

▶  $+7_{10} = 00111_2$

▶  $-7_{10} = 10111_2$

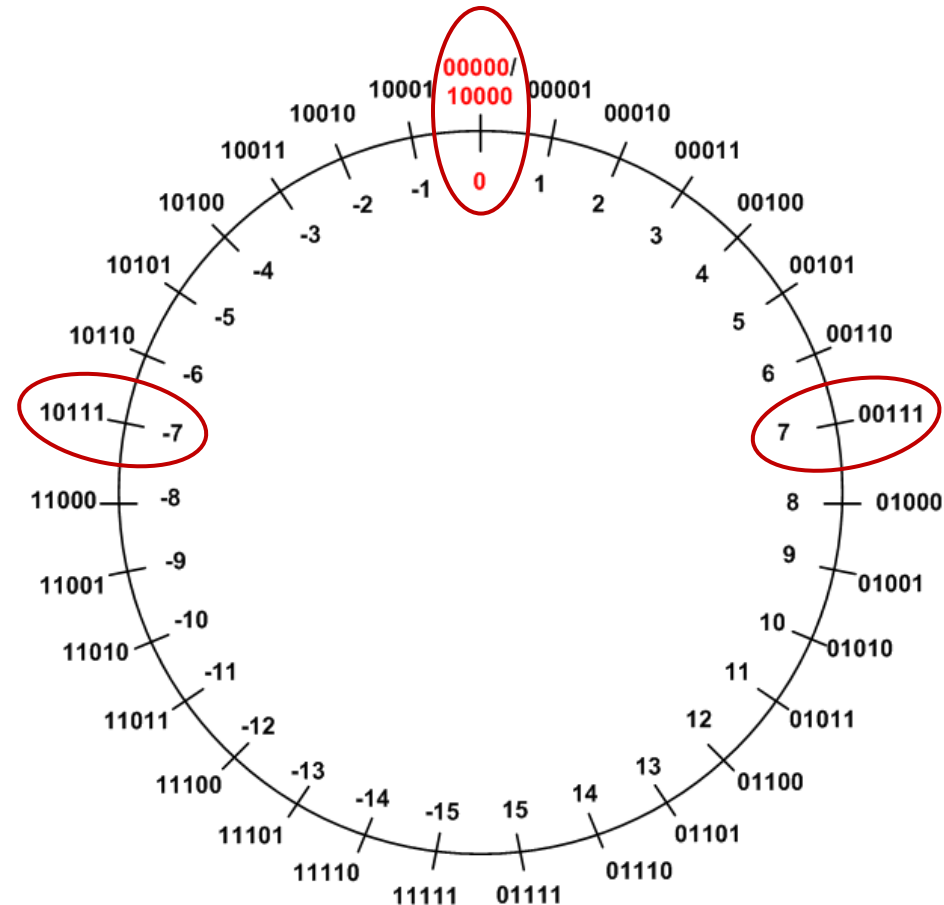
#### ▶ Two ways to represent zero

▶  $+0_{10} = 00000_2$

▶  $-0_{10} = 10000_2$

#### ▶ Not used in modern systems

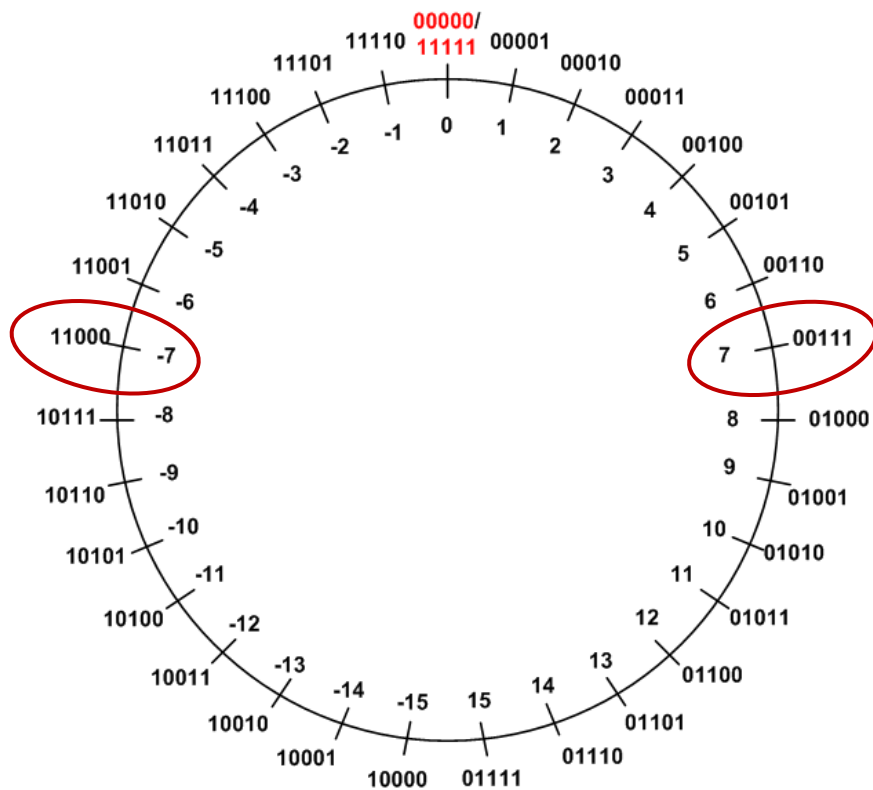
- ▶ Hardware complexity
- ▶ Two zeros



# Signed Integers

## Method 2: One's Complement

**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

$$+7_{10} = 00111_2$$

$$-7_{10} = 11000_2$$

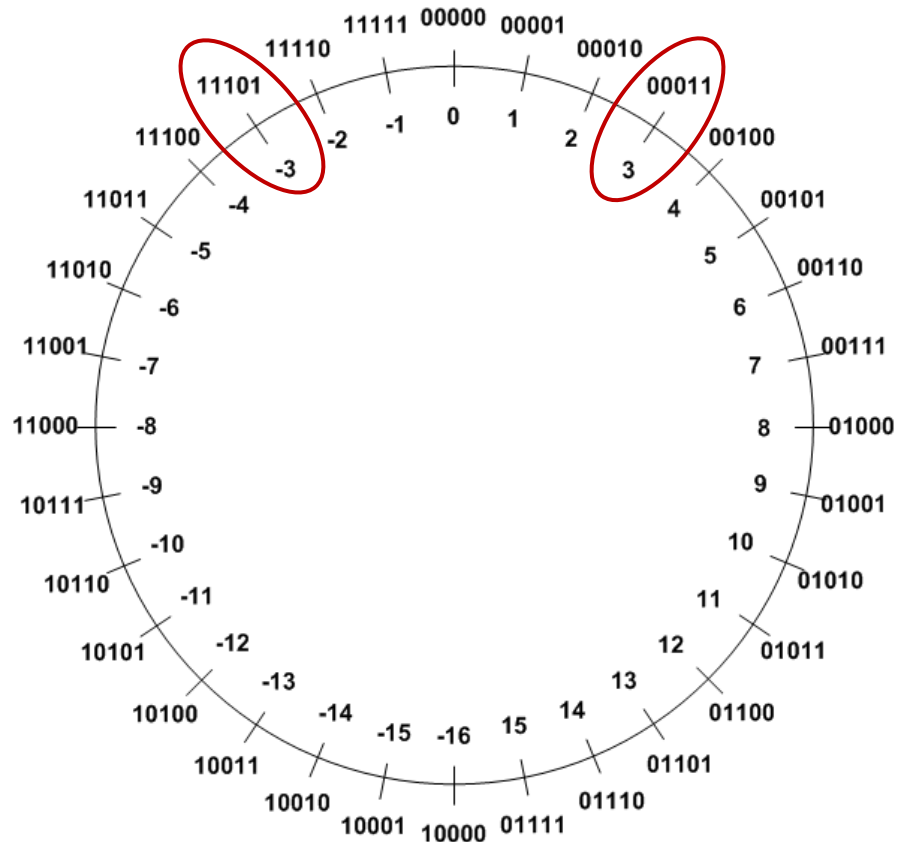
$$\begin{aligned} +7_{10} + (-7_{10}) &= 00111_2 + 11000_2 \\ &= 11111_2 \\ &= 2^5 - 1 \end{aligned}$$

# Signed Integers

## Method 3: Two's Complement (TC)

**Two's Complement ( $\bar{\alpha}$ ):**

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



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

**Example 1: TC(3)**

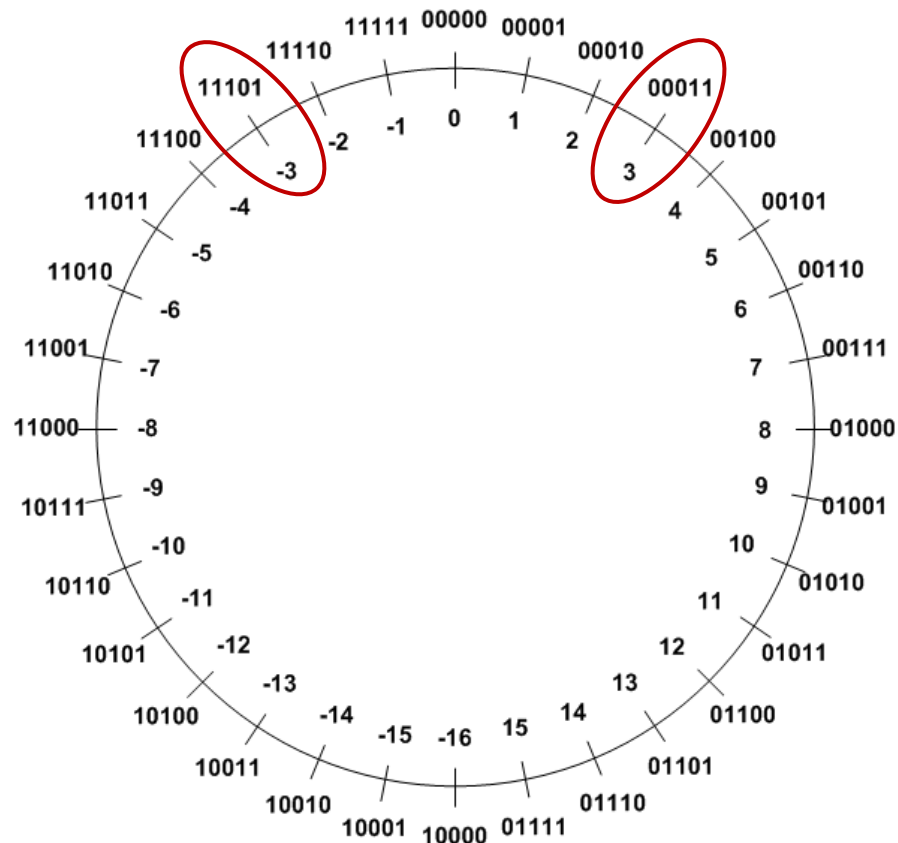
	Binary	Decimal
Original number	0b00011	3
Step 1: Invert every bit	0b11100	
Step 2: Add 1	+ 0b00001	
Two's complement	0b11101	-3

# Signed Integers

## Method 3: Two's Complement (TC)

### Two's Complement (TC)

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

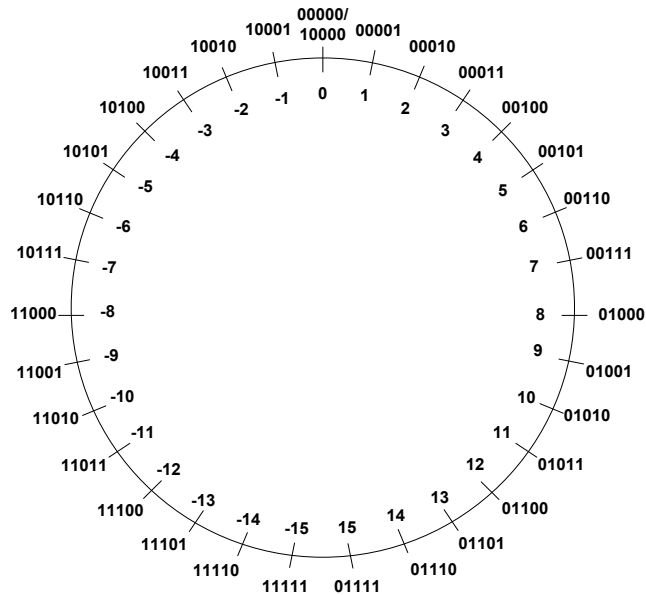


**TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.**

Example 2: **TC(-3)**

	Binary	Decimal
Original number	0b11101	-3
Step 1: Invert every bit	0b00010	
Step 2: Add 1	+ 0b00001	
Two's complement	0b00011	3

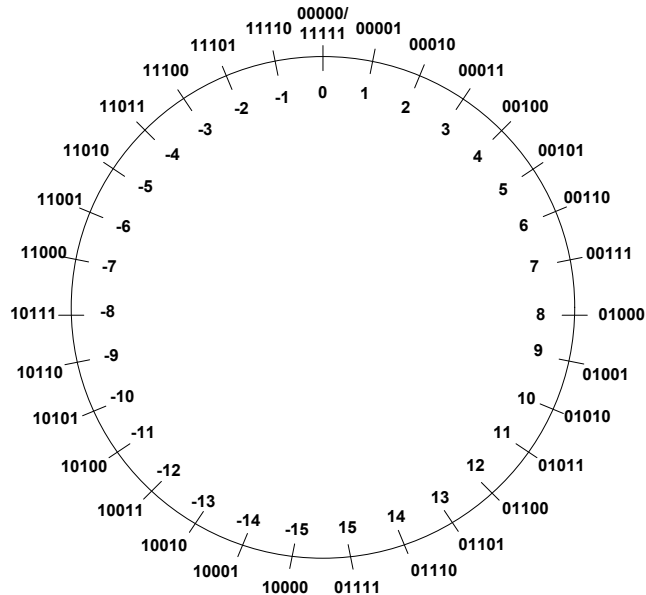
# Comparison



Signed magnitude  
representation

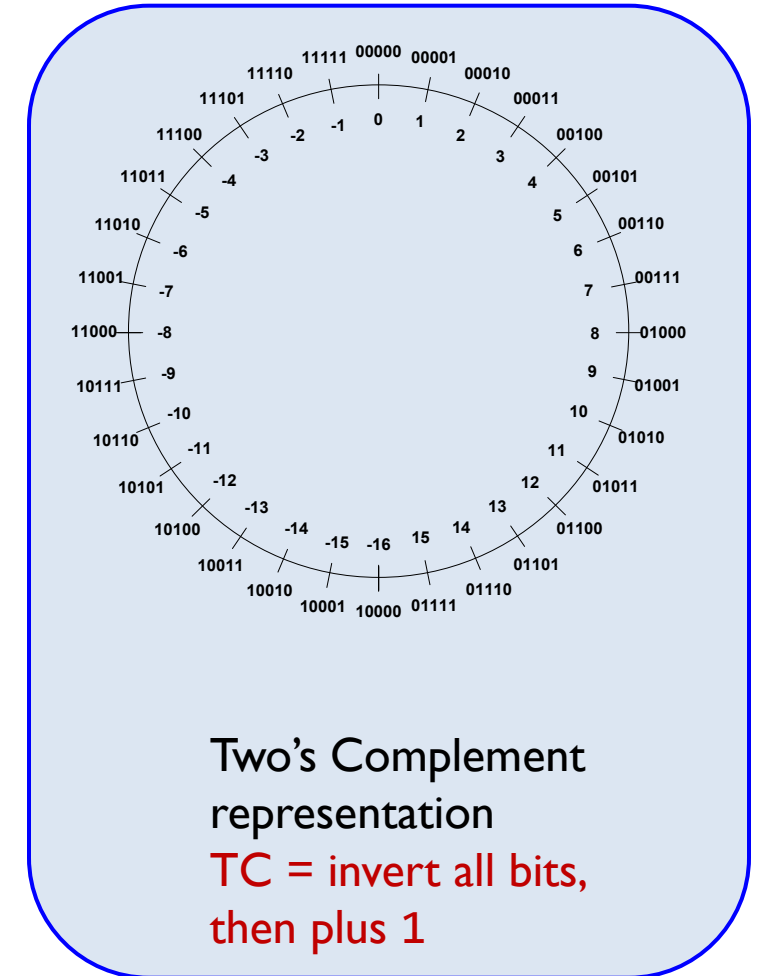
0 = positive

1 = negative



One's complement  
representation

Negative = invert all  
bits of a positive

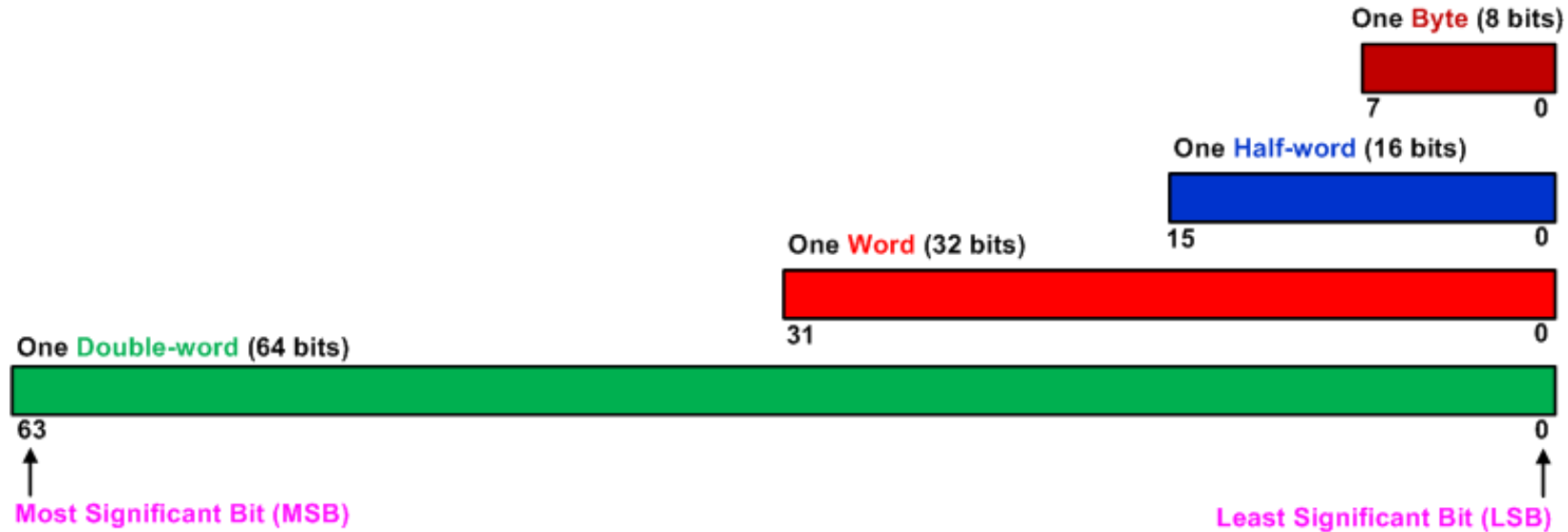


Two's Complement  
representation

TC = invert all bits,  
then plus 1

Used in modern computers!

# Range of Signed Integers (Two's Complement)



Storage Size	Range	Powers of 2
Signed Byte	-128 to +127	$-2^7$ to $2^7-1$
Signed Halfword	-32,768 to +32,767	$-2^{15}$ to $2^{15}-1$
Signed Word	-2,147,483,648 to +2,147,483,647	$-2^{31}$ to $2^{31}-1$
Signed Double-word	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	$-2^{63}$ to $2^{63}-1$

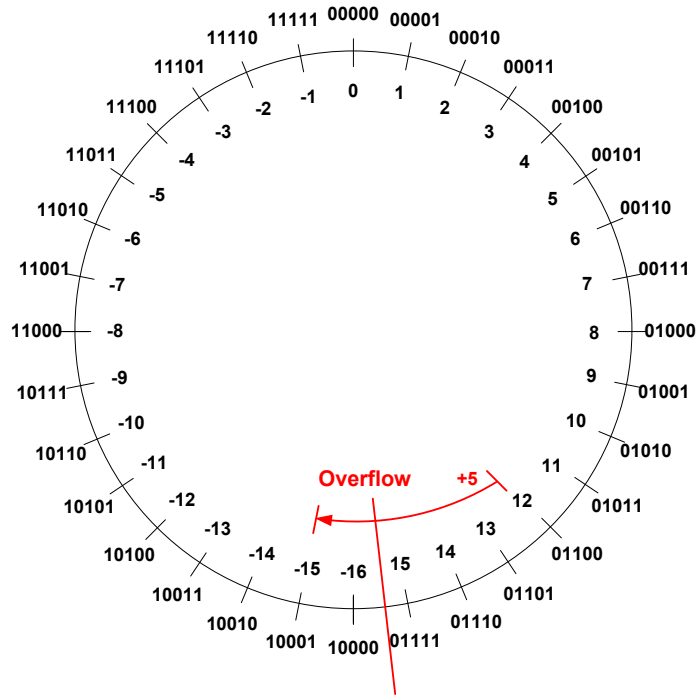


# Overflow for Signed Add/Sub

---

- ▶ 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.

# Overflow for Signed Add

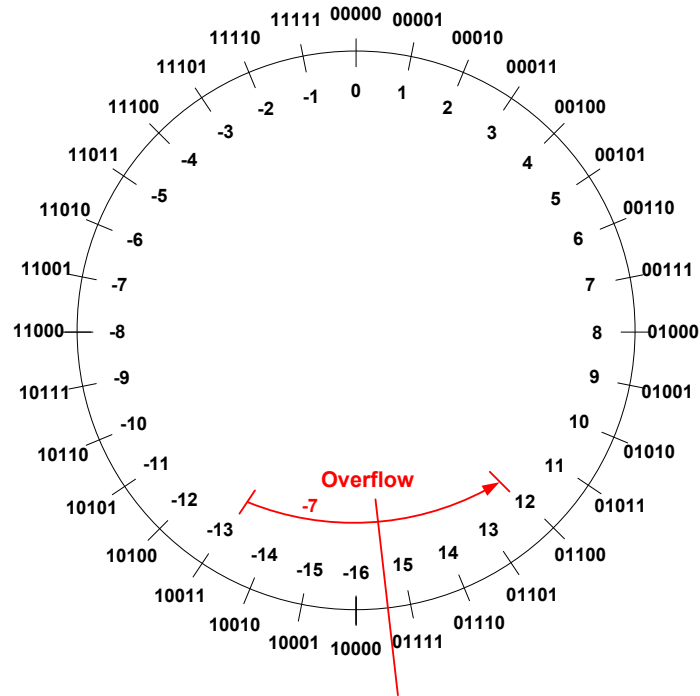


Overflow occurs when adding two positive integers but getting a negative result.

0 1 1 0 0	12
+ 0 0 1 0 1	+ 5
<hr/>	<hr/>
1 0 0 0 1	-15
<hr/>	
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.

# Overflow for Signed Add



Overflow occurs when adding two negative integers but 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.

# Signed or Unsigned

---

$a = 0b10000$

$b = 0b10000$

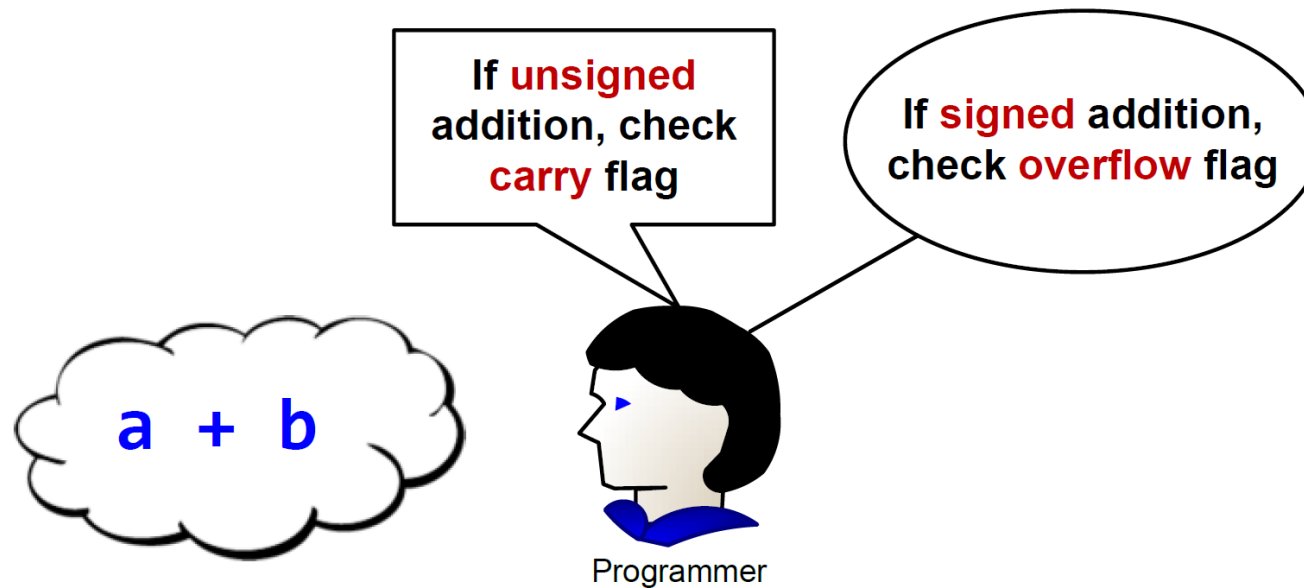
$c = a + b$

- ▶ Are  $a$  and  $b$  signed or unsigned numbers?
  - ▶ CPU does not know the answer at all.
  - ▶ Therefore, the hardware sets up both the carry flag and the overflow flag.
  - ▶ It is software's (programmers'/compilers') responsibility to interpret the flags.

# Signed or unsigned

---

- ▶ Whether the carry flag or the overflow flag should be used depends on the programmer's intention.



- ▶ When programming in high-level languages such as C, the compiler automatically chooses to use the carry or overflow flag based on how this integer is declared in source code ("**int**" or "**unsigned int**").

# Signed or Unsigned

---

$a = 0b10000$

$b = 0b10000$

$c = a + b$

- ▶ Are  $a$  and  $b$  signed or unsigned numbers?

```
uint a;  
uint b;  
...  
c = a + b  
...
```

C Program

Check the carry flag!

# Signed or Unsigned

---

$a = 0b10000$

$b = 0b10000$

$c = a + b$

- ▶ Are  $a$  and  $b$  signed or unsigned numbers?

```
int a;  
int b;  
...  
c = a + b  
...
```

C Program

Check the overflow flag!

# Signed Integer Representation

## Method 3: Two's Complement

---

Assume a four-bit system:

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



# Signed Integer Representation

## Method 3: Two's Complement

---

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

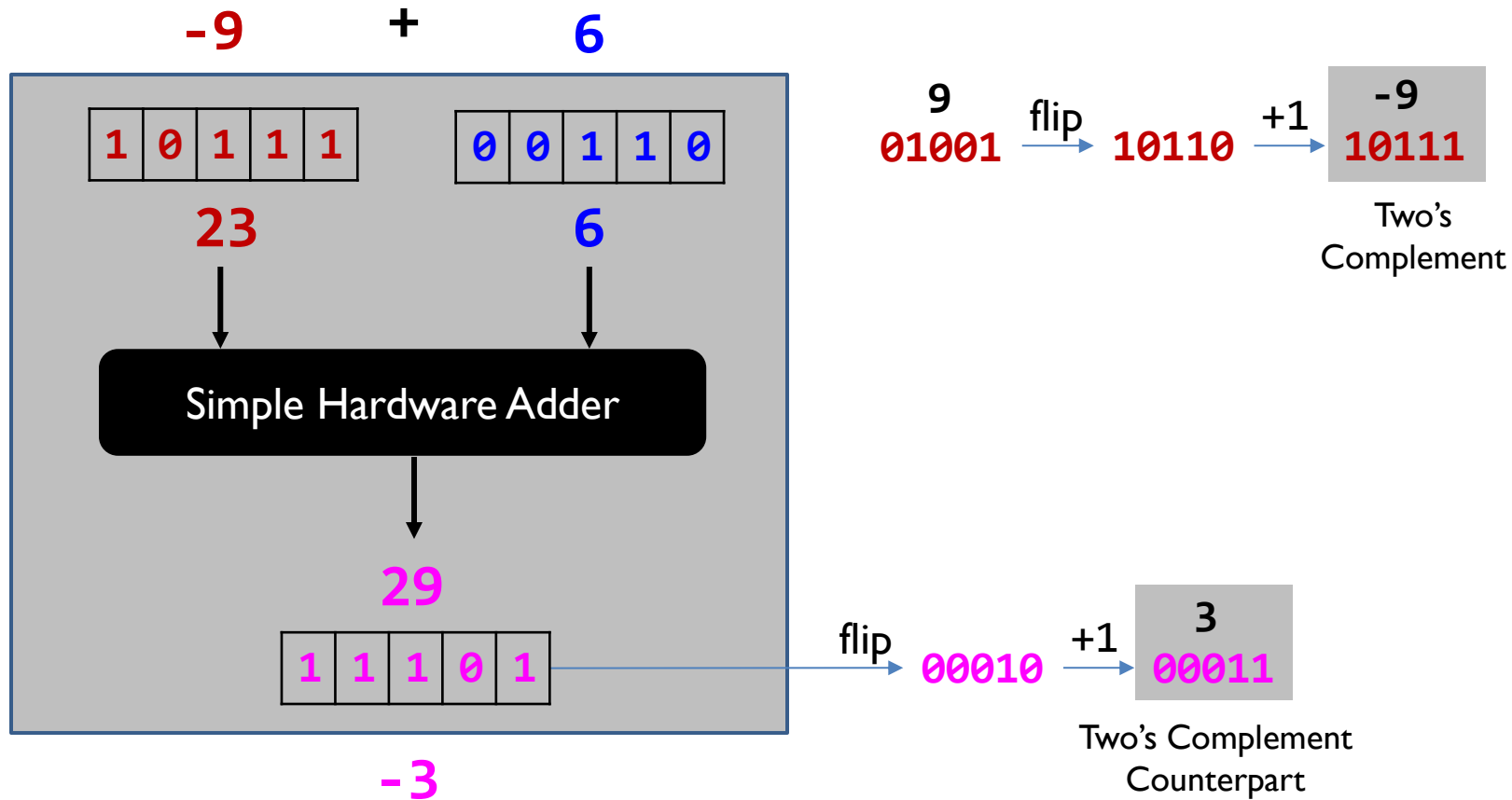
# Why use Two's Complement

---

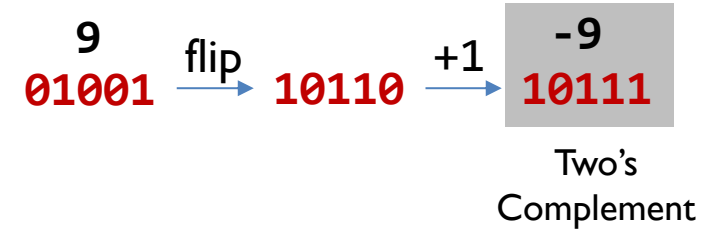
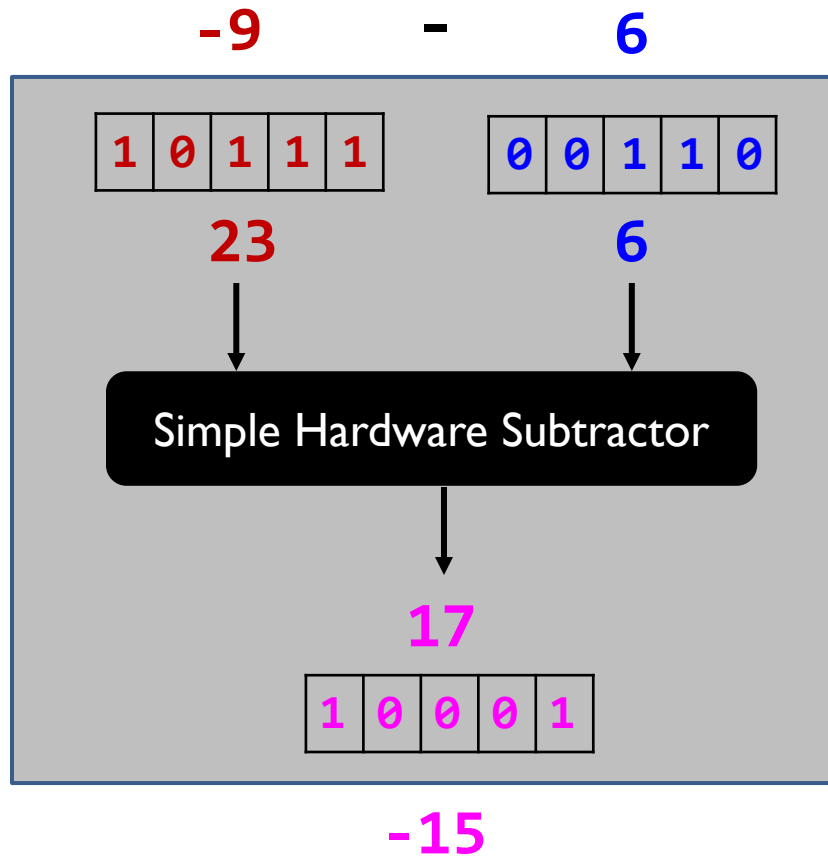
Two's complement simplifies hardware

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: $(-9) + 6$



# Subtracting two signed integers: $(-9) - 6$



## 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.

# Condition Codes

---

Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed arithmetic out of range
C	carry	signed arithmetic out of range

- C is set upon an **unsigned** addition if the answer is wrong
- C is cleared upon an **unsigned** subtract if the answer is wrong
- V is set upon a **signed** addition or subtraction if the answer is wrong

Why do we care about these bits?

# Carry and Overflow

$$c = a \pm b$$

The result is incorrect for the following cases:

	Carry (for unsigned)	Overflow (for signed)
Add	<i>Carry</i> = 1 if $c$ is too large to fit in.	<i>Overflow</i> = 1 if $c$ is too large or too small to fit in
Subtract	$Borrow = 1$ , i.e. <i>Carry</i> = 0 if $a < b$ .	

- ARM Cortex-M has no dedicated borrow flag, carry flag is reused.
- For unsigned subtract,  $Borrow = \overline{Carry}$

- Signed Subtraction is converted to sign addition
- $a - b = a + (-b)$

# Characters

**A**merican  
**S**tandard  
**C**ode for  
**I**nformation  
**I**nterchange

- Standard ASCII  
0 - 127
- Extended ASCII  
0 - 255
- ANSI  
0 - 255
- Unicode  
0 - 65535

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Standard ASCII: Encoding 128 characters



# Null-terminated String

---

```
char str[13] = "ARM Assembly";  
// The length has to be at least  
// 13 even though it has 12  
// letters. The NULL terminator  
// should be included.
```

or simply

```
char str[] = "ARM Assembly";
```

Memory Address	Memory Content	Letter
str + 12 →	0x00	\0
str + 11 →	0x79	y
str + 10 →	0x6C	l
str + 9 →	0x62	b
str + 8 →	0x6D	m
str + 7 →	0x65	e
str + 6 →	0x73	s
str + 5 →	0x73	s
str + 4 →	0x41	A
str + 3 →	0x20	space
str + 2 →	0x4D	M
str + 1 →	0x52	R
str →	0x41	A

# String Comparison

---

Strings are compared based on their ASCII values

- ▶ “j” < “jar” < “jargon” < “jargonize”
- ▶ “CAT” < “Cat” < “DOG” < “Dog” < “cat” < “dog”
- ▶ “12” < “123” < “2” < “AB” < “Ab” < “ab” < “abc”

# String Length

---

- ▶ Stings are terminated with a **null** character (NUL, ASCII value 0x00)

## Pointer dereference operator \*

```
int strlen (char *pStr){
    int i = 0;

    // loop until *pStr is NULL
    while( *pStr ) {
        i++;
        pStr++;
    }
    return i;
}
```

## Array subscript operator []

```
int strlen (char *pStr){
    int i = 0;

    // loop until pStr[i] is NULL
    while( pStr[i] )
        i++;

    return i;
}
```

# Convert to Upper Case

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	5A

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A

$$\text{'a'} - \text{'A'} = 0x61 - 0x41 = 0x20 = 32$$

## Pointer dereference operator \*

```
void toUpper(char *pStr){
    char *p;

    for(*p = pStr; *p; ++p){
        if(*p >= 'a' && *p <= 'z')
            *p -= 'a' - 'A';
            //or: *p -= 32;
    }
}
```

## Array subscript operator []

```
void toUpper(char *pStr){
    int i;
    char c = pStr[0];
    for(i = 0; c; i++, c = pStr[i];) {
        if(c >= 'a' && c <= 'z')
            pStr[i] -= 'a' - 'A';
            // or: pStr[i] -= 32;
    }
}
```