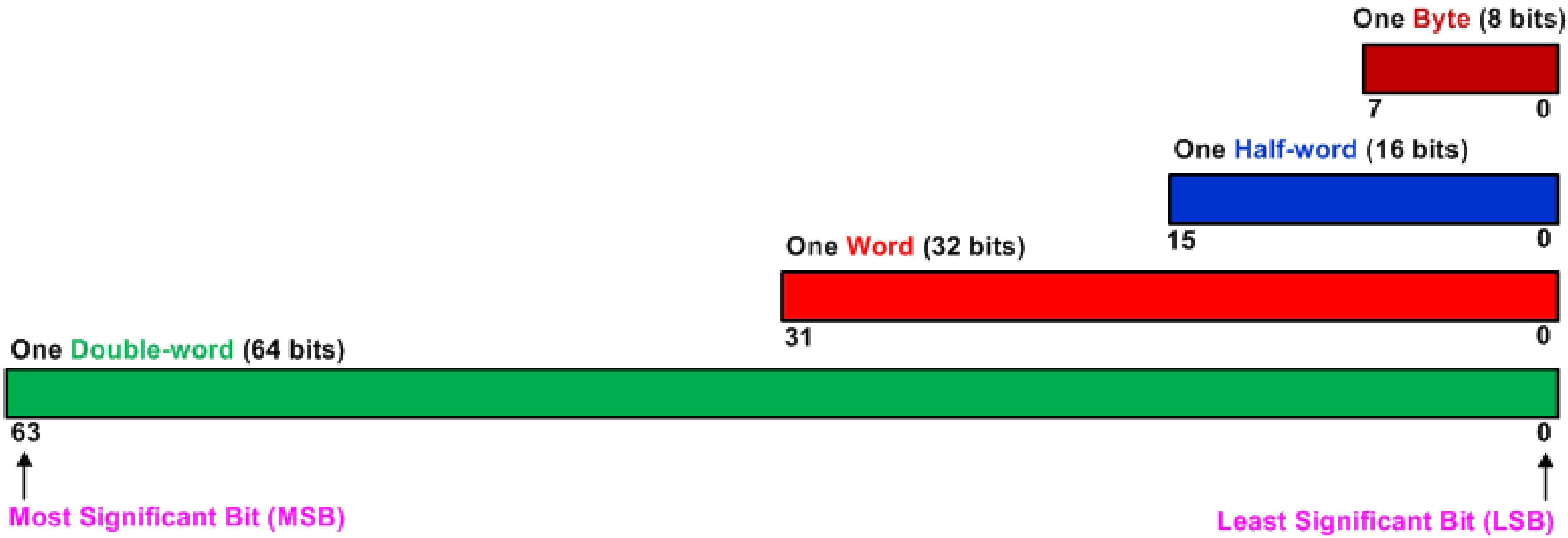
Embedded Systems with ARM Cortex-M3 Microcontrollers in Assembly Language and C

Chapter 2
Data Representation

Dr. Yifeng Zhu
Electrical and Computer Engineering
University of Maine

Spring 2015

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



Binary, Octal, Decimal and Hex

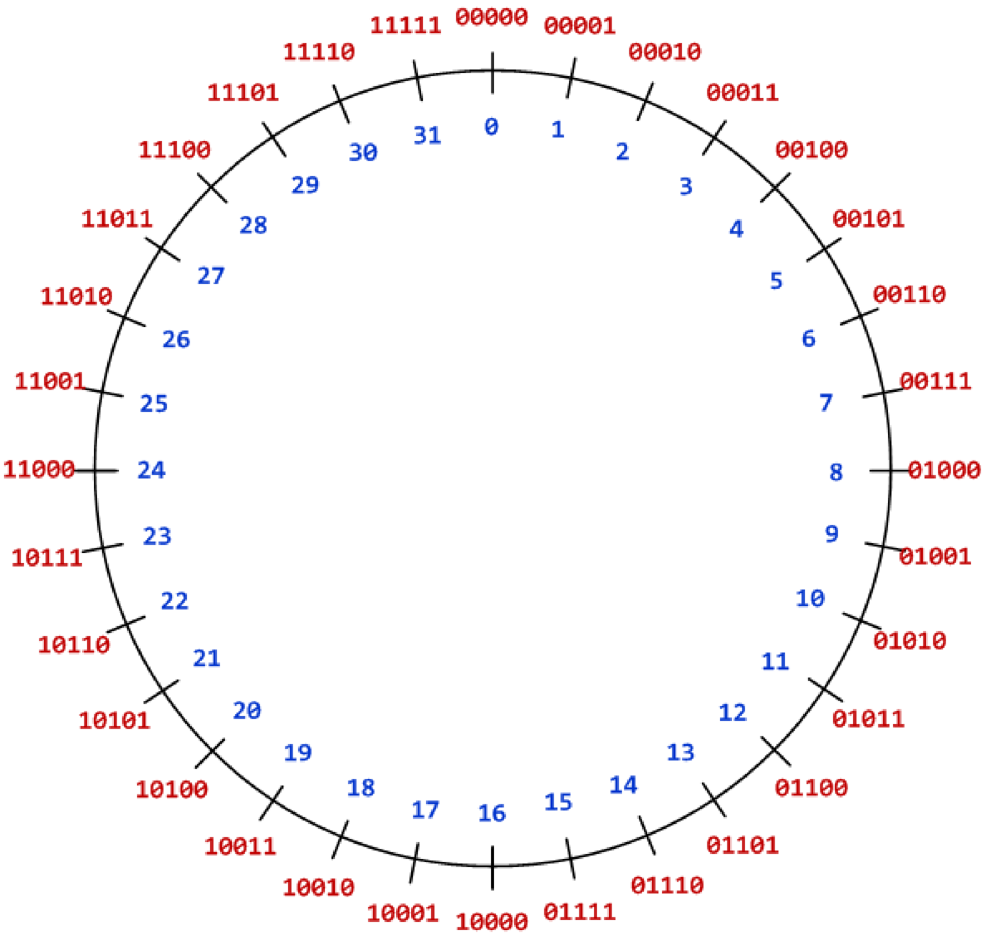
Decimal	Binary	Octal	Hex
0	0000	00	0x0
1	0001	01	0x1
2	0010	02	0x2
3	0011	03	0x3
4	0100	04	0x4
5	0101	05	0x5
6	0110	06	0x6
7	0111	07	0x7
8	1000	010	0x8
9	1001	011	0x9
10	1010	012	0xA
11	1011	013	0xB
12	1100	014	0xC
13	1101	015	0xD
14	1110	016	0xE
15	1111	017	0xF

Magic 32-bit Numbers

- Used as a special pattern for debug
- Used as a special pattern of memory values during allocation and de-allocation

`0xDEADBEEF`	Dead Beef
`0xBADDCAFE`	Bad Cafe
`0xFEE1DEAD`	Feel Dead
`0x8BADF00D`	Ate Bad Food
`0xBAADF00D`	Bad Food
`0xDEADC0DE`	Dead Code
`0xFACEB00C`	Facebook
`0xDEADD00D`	Deade Dude

Unsigned Integers



Five-bit binary code

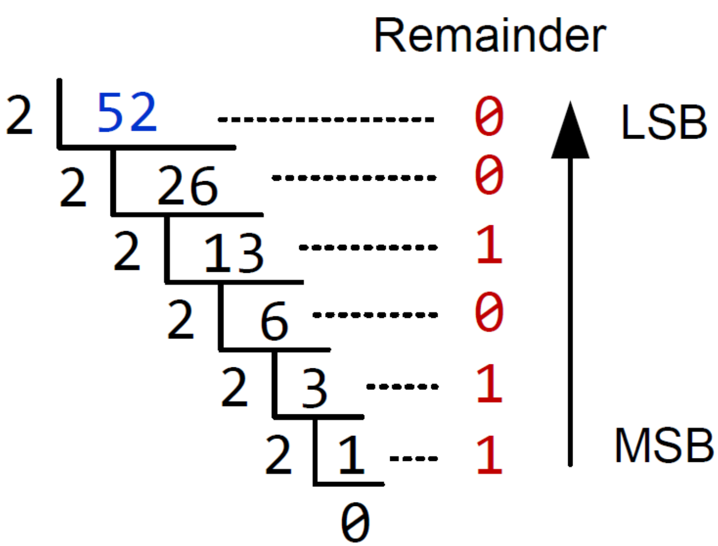
Convert from Binary to Decimal:

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

Unsigned Integers

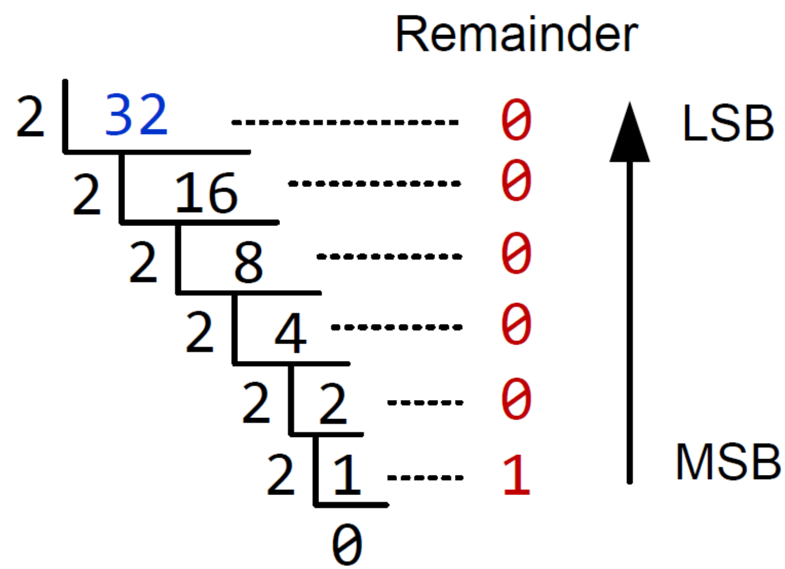
Convert Decimal to Binary

Example 1



$52_{10} = 110100_2$

Example 2



$32_{10} = 100000_2$

Carry/borrow flag bit for unsigned numbers

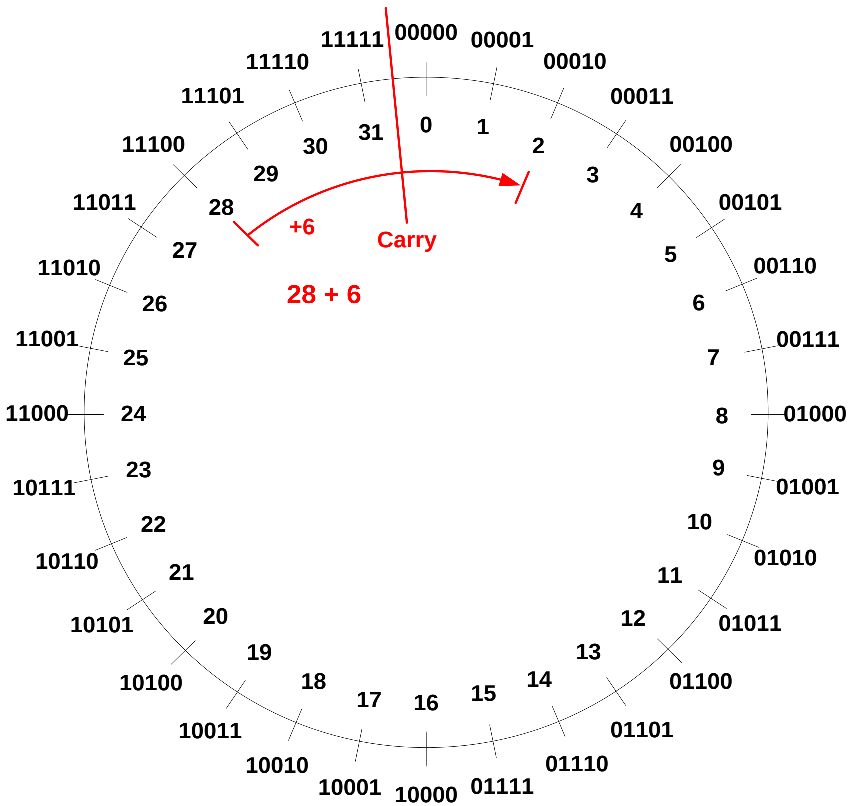
- When adding two unsigned numbers in an *n*-bit system, a carry occurs if the result is larger than the maximum unsigned integer that can be represented (*i*.e. $2^n - 1$).

- When subtracting two unsigned 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 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   1 1 1 0 0
          1 1 1 0 0        28
        + 0 0 1 1 0      +  6
        -----------      ----
        1 0 0 0 1 0        32
```

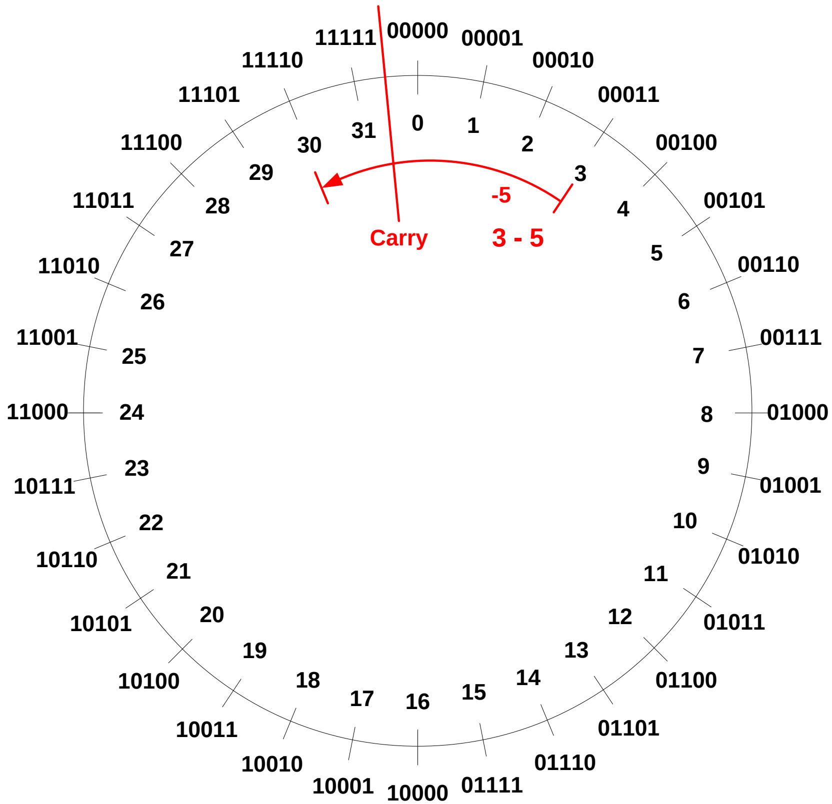
Extra bit is discarded.

5-bit result

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

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.



Barrow	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 Overview

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

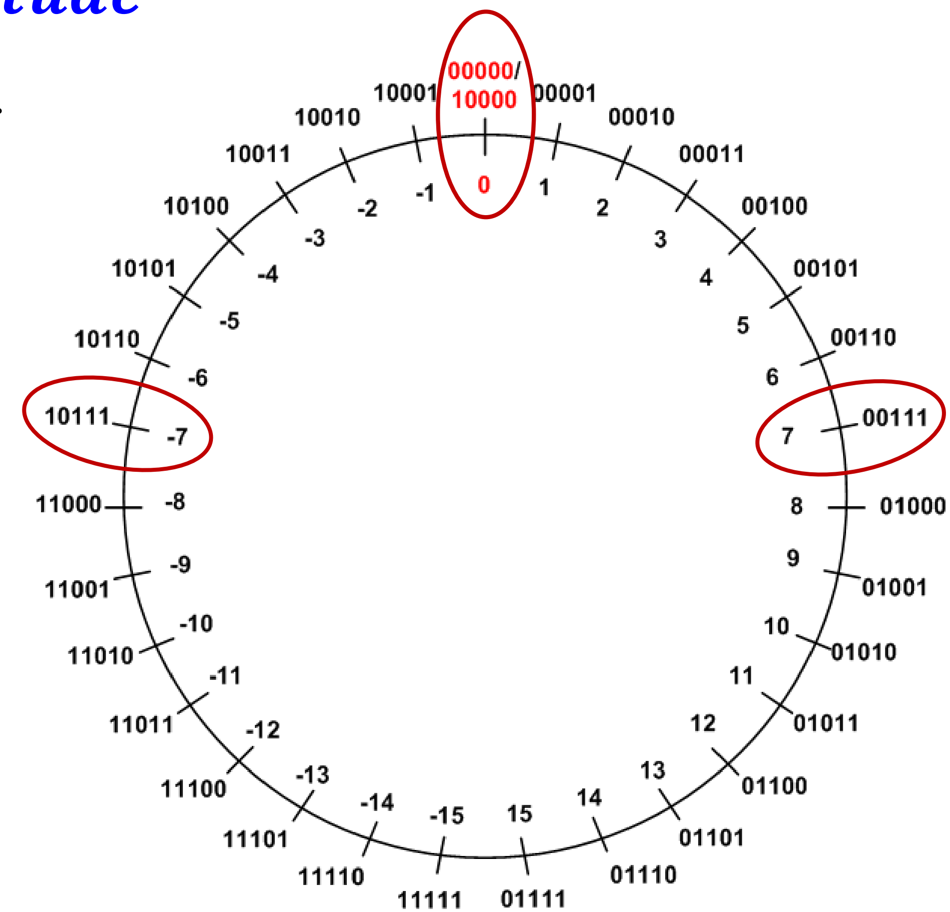
	Sign-and-Magnitude	One's Complement	Two's Complement
Range	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$
Zero	Two zeroes (± 0)	Two zeroes (± 0)	One zero
Unique Numbers	$2^n - 1$	$2^n - 1$	2^n

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.

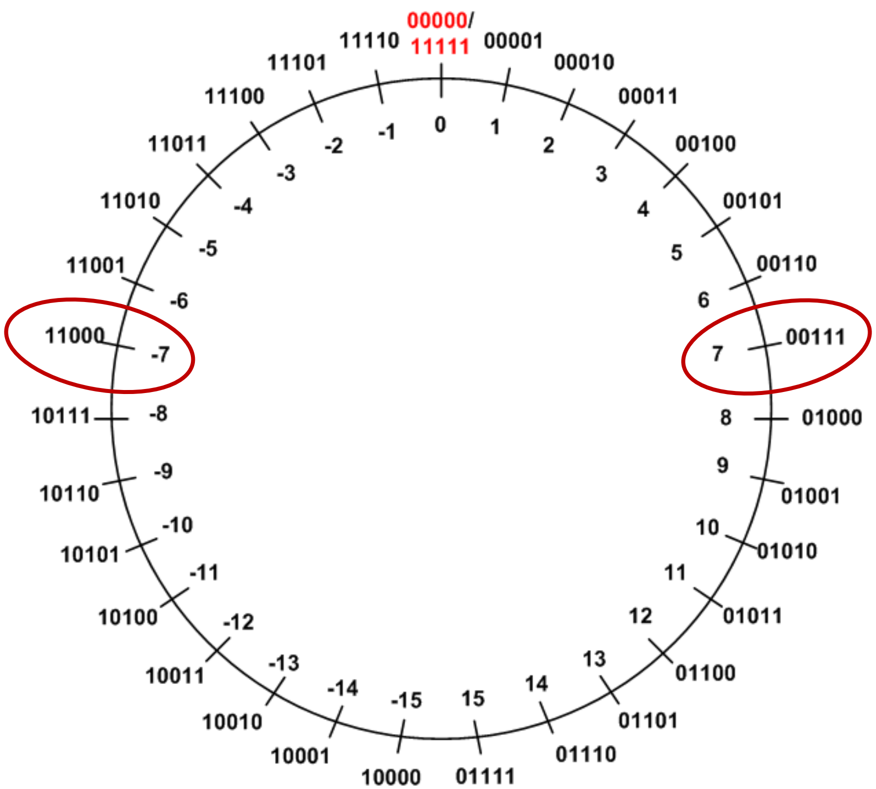


- Example: in a 5-bit system
  - $+7_{10} = 00111_2$
  - $-7_{10} = 10111_2$
- Two ways to represent zero
  - $+0_{10} = 00000_2$
  - $-0_{10} = 10000_2$
- Not used in modern systems
  - Hardware complexity
  - Two zeros

Signed Integers
Method 2: One's Complement

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

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



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

Example: in a 5-bit system

$+7_{10} = 00111_2$

$-7_{10} = 11000_2$

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

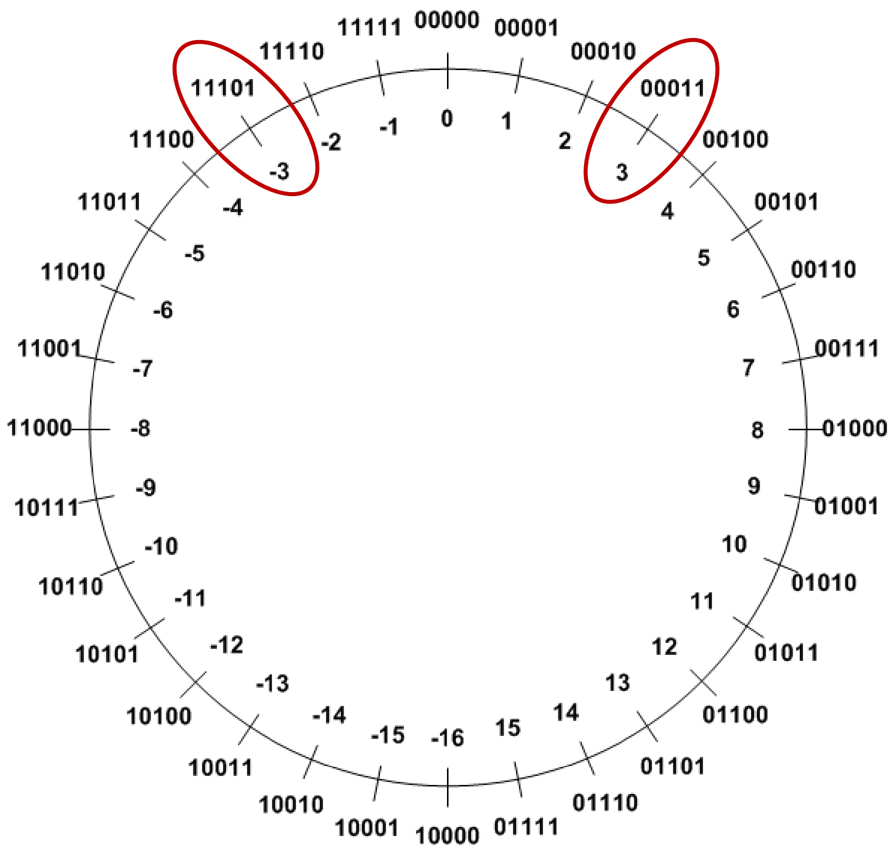
$= 11111_2$

$= 2^5 - 1$

Signed Integers
Method 3: Two's Complement

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

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



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

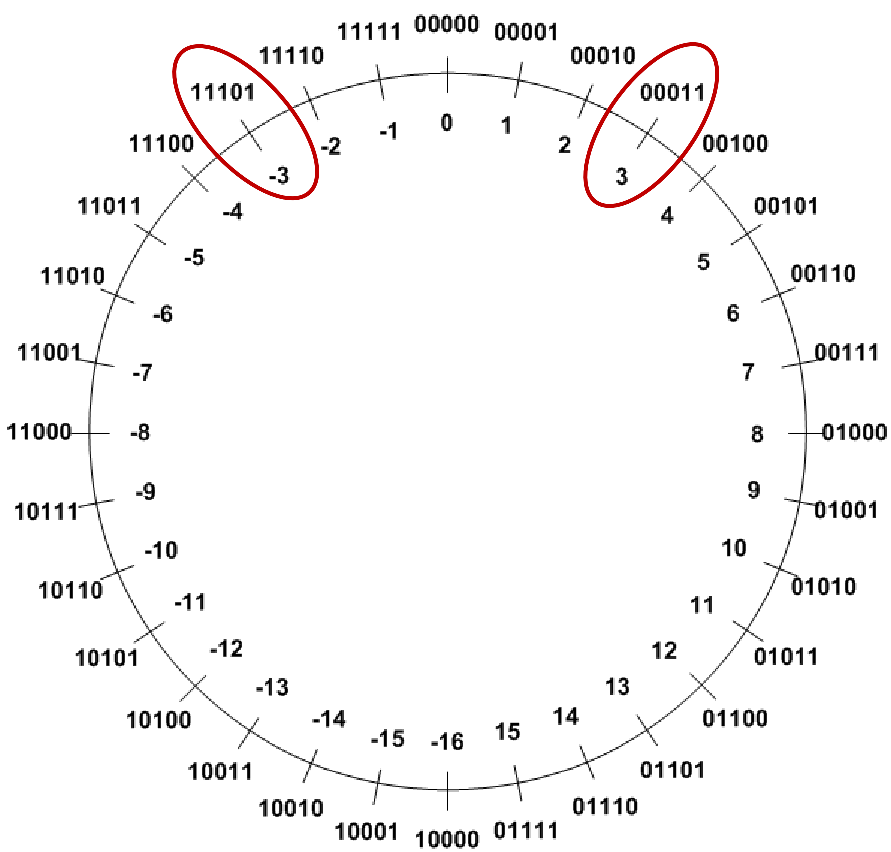
Example **1**: TC(3)

	Binary	Decimal
Original number	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

Two's Complement (TC)

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

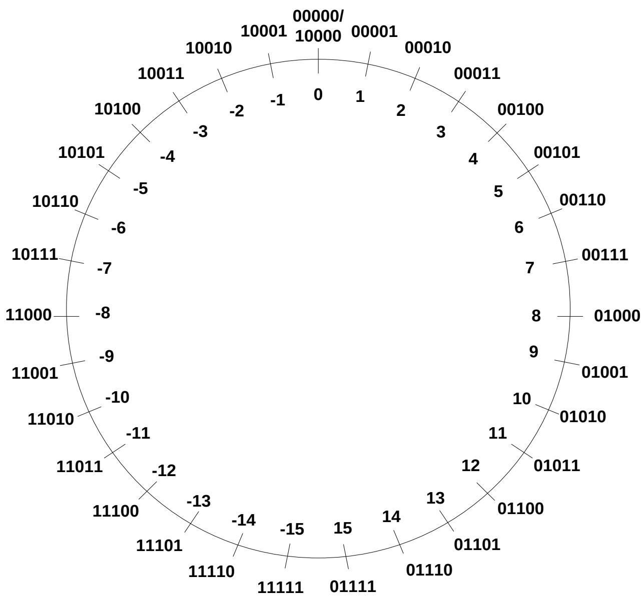


TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.

Example 2: TC(-3)

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

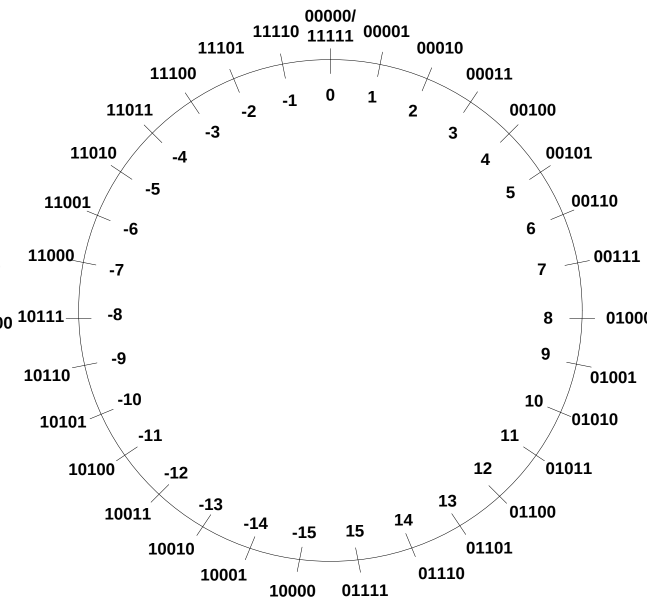
Comparison



Signed magnitude representation

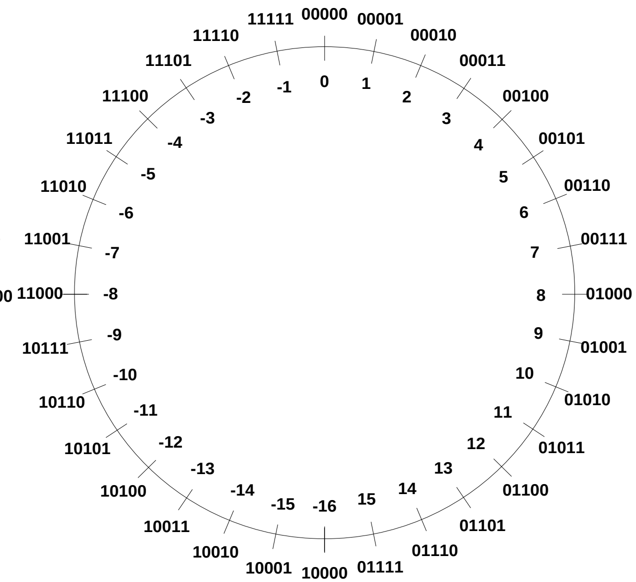
0 = positive

1 = negative



One’s complement representation

Negative = invert all bits of a positive



Two’s Complement representation

TC = invert all bits, then plus 1

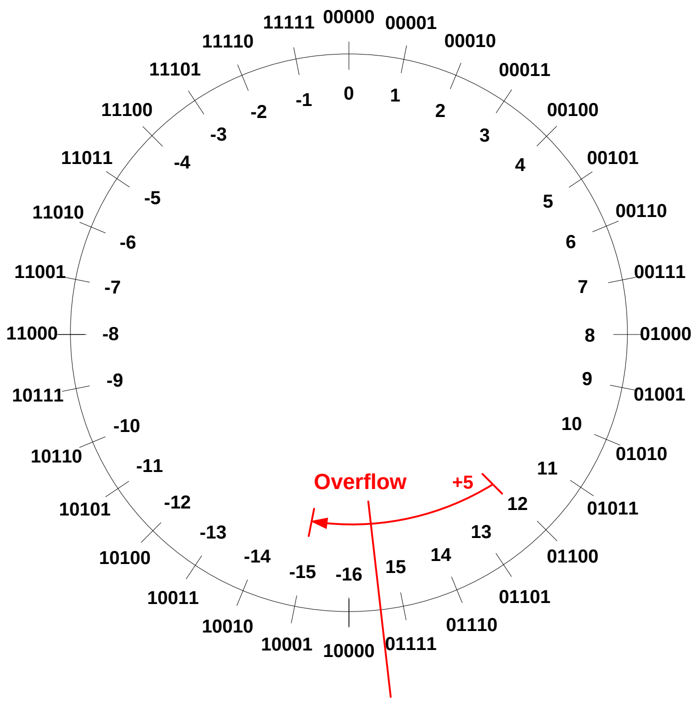
Overflow flag for signed numbers

- 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.

Overflow bit flag for signed numbers



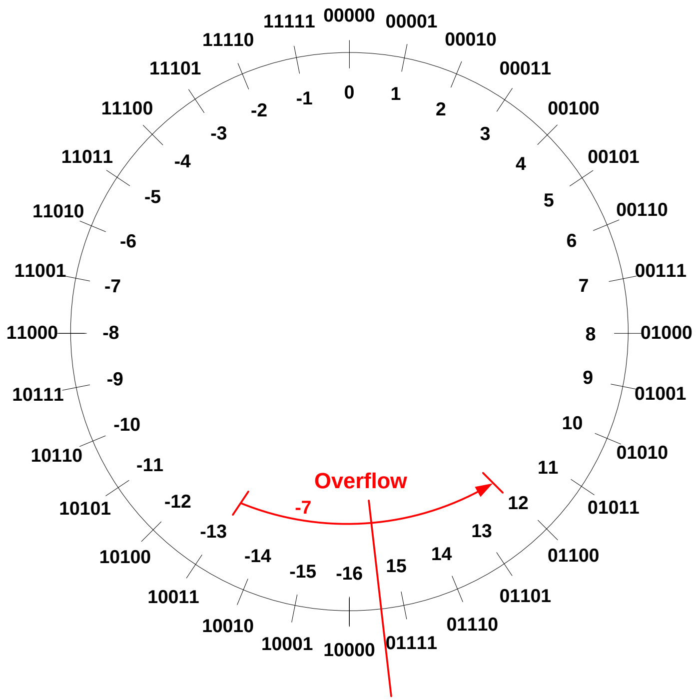
An overflow occurs when adding two positive numbers and getting a negative result.

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

$\underbrace{1\ 0\ 0\ 0\ 1}_{\text{5-bit result}}$

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 bit flag for signed numbers




```
    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.

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

Signed or Unsigned

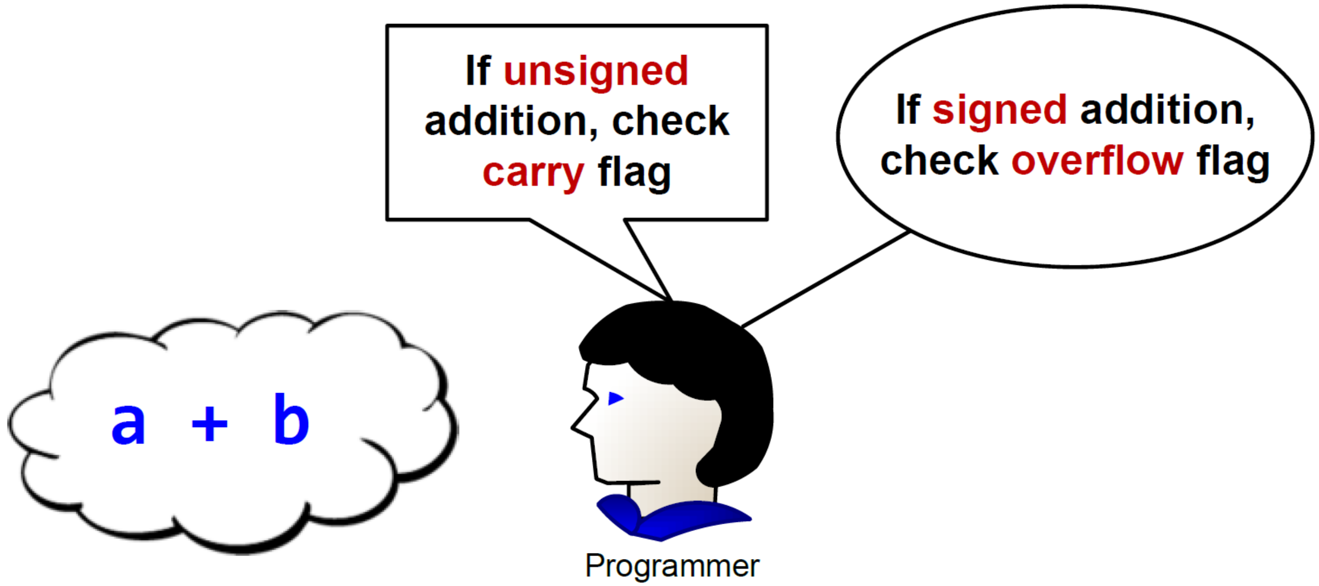
a = 0b10000
b = 0b10000
$c = a + b$

- 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**

Two's Complement Simplifies Hardware Implementation

The hardware adder designed for adding two unsigned numbers also works correctly for adding two signed numbers.

		Simple Addition (ignore sign)				
addend		1	0	1	1	1
+ addend	+	0	0	1	1	0
sum		1	1	1	0	1

Two's Complement Simplifies Hardware Implementation

The hardware adder designed for adding two unsigned numbers also works correctly for adding two signed numbers.

	Simple Addition (ignore sign)						Unsigned Addition	
addend		1	0	1	1	1		23
+ addend	+	0	0	1	1	0	+	6
sum		1	1	1	0	1		29

- If 11101 represents an unsigned number,
  $11101_2 = 29_{10}$

Two's Complement Simplifies Hardware Implementation

The hardware adder designed for adding two unsigned numbers also works correctly for adding two signed numbers.

	Simple Addition (ignore sign)						Unsigned Addition	Signed Addition
addend		1	0	1	1	1	23	-9
+ addend	+	0	0	1	1	0	+ 6	+ 6
sum		1	1	1	0	1	29	-3

- If 11101 represents an unsigned number,
  $11101_2 = 29_{10}$
- If 11101 represents a signed number,
  $11101_2 = -3_{10}$

Two's Complement Simplifies Hardware Implementation

The hardware adder designed for adding two unsigned numbers also works correctly for adding two signed numbers.

	Simple Addition (ignore sign)					Unsigned Addition	Signed Addition	
	1	0	1	1	1	23	-9	addend
+	0	0	1	1	0	+ 6	+ 6	+ addend
	1	1	1	0	1	29	-3	sum

- If 11101 represents an unsigned number,
  $11101_2 = 29_{10}$
- If 11101 represents a signed number,
  $11101_2 = -3_{10}$

Two's Complement Simplifies Hardware Implementation

The same subtraction hardware works correctly for both signed subtraction and unsigned subtraction.

	Simple Subtraction (ignore sign)						
minuend		1	0	1	1	1	
- subtrahend	-	0	0	1	1	0	
difference		1	0	0	0	1	

Two's Complement Simplifies Hardware Implementation

The same subtraction hardware works correctly for both signed subtraction and unsigned subtraction.

	Simple Subtraction (ignore sign)							Unsigned Subtraction	
minuend		1	0	1	1	1			23
- subtrahend	-	0	0	1	1	0		-	6
difference		1	0	0	0	1			17

- If 11101 represents an unsigned number,

  $10001_2 = 17_{10}$

Two's Complement Simplifies Hardware Implementation

The same subtraction hardware works correctly for both signed subtraction and unsigned subtraction.

	Simple Subtraction (ignore sign)						Unsigned Subtraction	Signed Subtraction
minuend		1	0	1	1	1	23	-9
- subtrahend	-	0	0	1	1	0	- 6	- 6
difference		1	0	0	0	1	17	-15

- If `11101` represents an unsigned number,
  $10001_2 = 17_{10}$
- If `11101` represents a signed number,
  $10001_2 = -15_{10}$

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 overflow
C	carry	unsigned overflow

C set after an **unsigned** addition if the answer is wrong
C clear after an **unsigned** subtract if the answer is wrong
V set after a **signed** addition or subtraction if the answer is wrong

Why do we care about these bits?

33

Formal Representation for Addition

R = X + Y

When adding two 32-bit integers X and Y, the flags are

- N = R_{31}
- Z is set if R is zero.
- C is set if the result is incorrect for an unsigned addition

$$C = X_{31} \& Y_{31} \parallel X_{31} \& \overline{R_{31}} \parallel Y_{31} \& \overline{R_{31}}$$

- V is set if the result is incorrect for a signed addition.

$$V = X_{31} \& Y_{31} \& \overline{R_{31}} \parallel \overline{X_{31}} \& \overline{Y_{31}} \& R_{31}$$

Formal Representation for Subtraction

R = X - Y

When subtracting two 32-bit integers X and Y, the flags are

- N = R_{31}
- Z is set if R is zero.
- C is ***clear*** if the result is incorrect for an unsigned subtraction

$$C = \overline{Y_{31} \& \ R_{31} \ \| \ \overline{X_{31}} \ \& \ R_{31} \ \| \ \overline{X_{31}} \ \& \ Y_{31}}$$

- V is clear if the result is incorrect for an signed subtraction.

$$V = X_{31} \& \overline{Y_{31}} \ \& \ \overline{R_{31}} \ \| \ \overline{X_{31}} \ \& \ Y_{31} \& \ R_{31}$$

ASCII

American
Standard
Code for
Information
Interchange

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

Encoding 128 characters

ASCII

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.

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"

Find out 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[i] is NULL
    while( pStr[i] )
        i++;

    return i;
}
```

Array subscript operator [ ]

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

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

‘a’ – ‘A’ = 0x61 – 0x41 = 0x20 = 32

Pointer dereference operator *

```
void toUpper(char *pStr){
  for(char *p = pStr; *p; ++p){
    if(*p >= ’a’ && *p <= ’z’)
      *p -= ‘a’ – ‘A’;
      //or: *p -= 32;
  }
}
```

Array subscript operator [ ]

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