Logic Design

Number Representation and Arithmetic Circuits

McMaster University

Number representation

- In a binary number the right-most bit is called the least-significant bit (LSB) and the left-most bit is called the most significant bit (MSB)
- A group of 4 bits is called a nibble
- A group of 8 bits is called a byte

Number representation

- Numbers that are positive only are called unsigned
- Numbers that can be positive or negative are called signed
- Numbers could be integer or real
- Simplest: unsigned integer
- A decimal integer:

$$D = d_{n-1}d_{n-2}...d_1d_0$$
$$V(D) = d_{n-1}\times 10^{n-1} + d_{n-2}\times 10^{n-2} + .. + d_1\times 10^1 + d_0\times 10^0$$

Number representation

- Conversion from decimal to binary: successively divide by 2
- In each step the remainder is the next binary digit
- The process continue until the quotient becomes zero

$$V = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + .. + b_1\times 2^1 + b_0\times 2^0$$
$$\frac{V}{2} = b_{n-1}\times 2^{n-2} + b_{n-2}\times 2^{n-3} + .. + b_1\times 2^0 + \frac{b_0}{2}$$

Number representation

- Binary numbers:

$$B = b_{n-1}b_{n-2}...b_1b_0$$
$$V(B) = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + .. + b_1\times 2^1 + b_0\times 2^0$$

1101

$$V = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 13$$
$$(1101)_2 = (13)_{10}$$

Number representation

Convert $(857)_{10}$

			Remainder	
857 ÷ 2	=	428	1	LSB
428 ÷ 2	=	214	0	
214 ÷ 2	=	107	0	
107 ÷ 2	=	53	1	
53 ÷ 2	=	26	1	
26 ÷ 2	=	13	0	
13 ÷ 2	=	6	1	
6 ÷ 2	=	3	0	
3 ÷ 2	=	1	1	
1 ÷ 2	=	0	1	MSB

Result is $(1101011001)_2$

Number representation

- The most common bases in addition to decimal are:
- base 2 (binary) { 0, 1 }
- base 8 (octal) { 0, 1, … 7}
- base 16 (hexadecimal) { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }
- Reason for using octal and hexadecimal systems: useful shorthand notation for binary numbers

Addition of Unsigned Numbers

$X = x_4x_3x_2x_1x_0$	0 1 1 1 1	$(15)_{10}$
$+\ Y = y_4y_3y_2y_1y_0$	0 1 0 1 0	$(10)_{10}$
	1 1 1 0 ← Generated carries	
$S = s_4s_3s_2s_1s_0$	1 1 0 0 1	$(25)_{10}$

Number representation

- One octal digit represents three bits
- Conversion from binary to octal: starting from the LSB replace every group of three digits with their corresponding octal digit
- Conversion from binary to hexadecimal: starting from the LSB replace every group of four digits with their corresponding hexadecimal digit
- Conversion from octal to binary: substitute each octal digit by corresponding three bits
- Conversion from hexadecimal to binary: substitute each hex digit by four bits

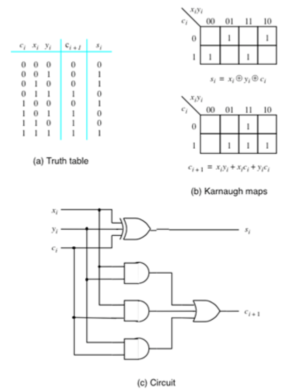
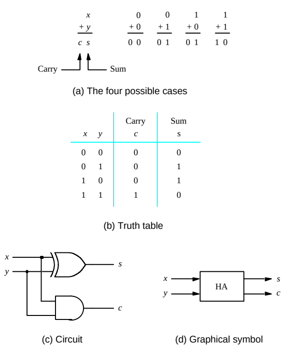
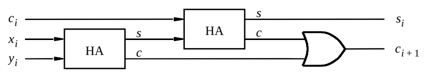


Figure 5.4. Full-adder.

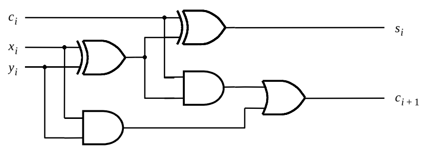
Addition of Unsigned Numbers



Decomposed Full Adder

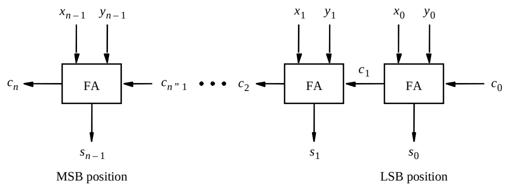


(a) Block diagram



(b) Detailed diagram

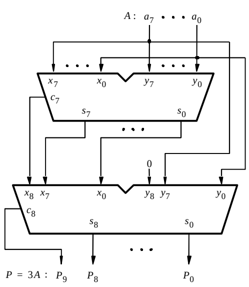
Ripple Carry Adder



Ripple Carry Adder

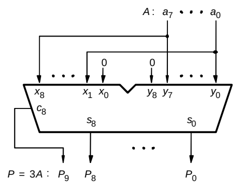
- When operands X and Y are applied as inputs to the adder, it takes some time before output sum S is valid.
- Each full-adder has a delay before its s_i and c_{i+1} are valid
- If this delay is Δt the complete sum will be valid after a delay of $n\Delta t$
- Because of the way the carry signal "ripple" through the full-adder, this circuit is called a ripple-carry adder

Example



(a) Naive approach

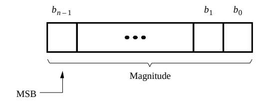
Example



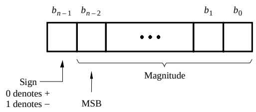
(b) Efficient design

Signed Numbers

- One of the bits (usually the left-most bit) is reserved for the sign of the number.
- Usually a 1 indicates *negative* and 0 indicates *positive.*



(a) Unsigned number



(b) Signed number

Signed Numbers

- Extending the 'natural' binary representation of positive integers to negative integers can be done in at least 3 different schemes: sign-magnitude, one's complement and two's complement.
- Sign-and-magnitude: The most significant bit (MSB) is reserved to the sign, 0 is positive, 1 is negative. All other bits are used to store the magnitude in the natural representation.
- Addition and subtraction are complicated.
- There are two representations for zero!

Signed Numbers

- <u>One's complement</u> Positive integers are like in the natural representation, negative numbers are obtained by complementing each bit of the corresponding positive number (i.e. the absolute value).
- There are two representations for zero! Bitwise addition of N and -N gives -0.
- Positive integers still have MSB = 0, and negative integers have MSB=1.
- 1's complement of an n-bit negative number K is obtained by subtracting its equivalent positive number P from 2^n-1
- $K_1=(2^n-1)-P$

Table: Signed Binary Integers (word length *n* = 4)

+N	Positive Integers (all systems)	−N	Negative Integers: Sign and Magnitude	Negative Integers: 2's Complement N^*	Negative Integers: 1's Complement $\bar{N}$
+0	0000	−0	1000	——	1111
+1	0001	−1	1001	1111	1110
+2	0010	−2	1010	1110	1101
+3	0011	−3	1011	1101	1100
+4	0100	−4	1100	1100	1011
+5	0101	−5	1101	1011	1010
+6	0110	−6	1110	1010	1001
+7	0111	−7	1111	1001	1000
		−8	——	1000	——

Signed Numbers

- <u>Two's complement</u> Like one's complement, but negative numbers are having 1 added after complementation.
- Bitwise addition of N and -N gives 0 if you ignore the carry out of the MSB.
- Positive integers still have MSB = 0, and negative integers have MSB=1. <u>Only one representation for zero!</u>
- 2's complement of an n-bit negative number K is obtained by subtracting its equivalent positive number P from 2^n
- $K_2=2^n-P$

2's complement signed numbers

$B=b_{n-1}b_{n-2}\ldots b_1b_0$

$$V=(-b_{n-1}\times 2^{n-1})+b_{n-2}\times 2^{n-2}+..+b_1\times 2^1+b_0\times 2^0$$

Largest negative number: -2^{n-1}

Largest positive number: $2^{n-1}-1$

Signed Numbers

- Relationship between 2's complement and 1's complement
- $K_2=K_1+1$
- A simple way of finding the 2's complement is to find 1's complement and add 1
- Rule for finding 2's complement:
  - Given signed number $B=b_{n-1}b_{n-2}\ldots b_1b_0$
  - 2's complement: $K=k_{n-1}k_{n-2}\ldots k_1k_0$
  - Examine bits of B from right to left, copy all bits of B that are 0 and the first bit that is 1, then complement the rest of the bits

1's complement addition

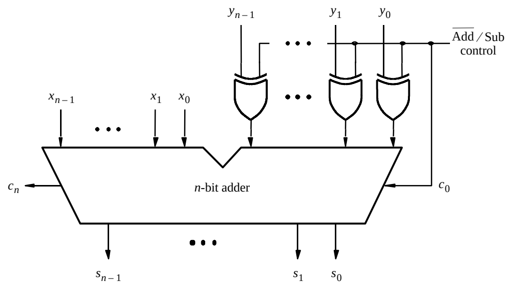
```
  (+5)     0101        (-5)     1010
+ (+2)   + 0010      + (+2)   + 0010
  ----     ----        ----     ----
  (+7)     0111        (-3)     1100

  (+5)     0101        (−5)     1010
+ (−2)   + 1101      + (−2)   + 1101
  ----    -----        ----    -----
  (+3)    10010        (−7)    10111
              1                    1
          -----                -----
           0011                 1000
```

Addition and Subtraction

- Addition of 1's complement numbers might need a correction
- Time needed to add two 1's complement numbers may be twice as long as time needed to add two unsigned numbers

Adder and Subtractor Unit



2's complement addition

(+ 5) + (+ 2) = (+ 7)	0101 + 0010 = 0111	(–5) + (+ 2) = (–3)	1011 + 0010 = 1101
(+ 5) + (–2) = (+ 3)	0101 + 1110 = 10011 (ignore carry)	(–5) + (–2) = (–7)	1011 + 1110 = 11001 (ignore carry)

Radix-complement schemes

- Complements – general theory
- The r's complement of an n-digit number N in base r is:
  $K_r = r^n - N$ for $N \neq 0$
  (0 for N=0)
- The (r-1)'s complement, K_{r-1} is defined as:
  $K_r = (r^n - 1) - N$
- The concept of subtracting a number by adding its radix-complement is general

2's complement subtraction

(+ 5) – (+ 2) = (+ 3)	0101 – 0010	⇒	0101 + 1110 = 10011 (ignore carry)
(–5) – (+ 2) = (–7)	1011 – 0010	⇒	1011 + 1110 = 11001 (ignore carry)
(+ 5) – (–2) = (+ 7)	0101 – 1110	⇒	0101 + 0010 = 0111
(–5) – (–2) = (–3)	1011 – 1110	⇒	1011 + 0010 = 1101

Arithmetic Overflow

- If n bits are used to represent signed numbers, result must be in the range -2^{n-1} to $2^{n-1}-1$
- If the result does not fit in this range, we say that arithmetic overflow has happened
- We should be able to detect overflow
- The key to determining the overflow is carry-out from MSB position and carry-out from the sign bit
- If they are the same no overflow has happened.

$$overflow = c_{n-1} \oplus c_n$$

Arithmetic Overflow

(+7)	0111	(−7)	1001
+ (+2)	+ 0010	+ (+2)	+ 0010
(+9)	1001	(−5)	1011
	$c_4 = 0$ $c_3 = 1$		$c_4 = 0$ $c_3 = 0$
(+7)	0111	(−7)	1001
+ (−2)	+ 1110	+ (−2)	+ 1110
(+5)	10101	(−9)	10111
	$c_4 = 1$ $c_3 = 1$		$c_4 = 1$ $c_3 = 0$

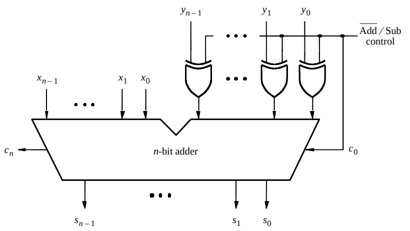
Fast Adders

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$
$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$
$$c_{i+1} = g_i + p_i c_i$$
$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_2 p_1 g_0 + p_i p_{i-1} \ldots p_1 p_0 c_0$$

Performance Issue

- Speed of any circuit is limited by the longest delay along the paths through the circuit
- This is called the critical path delay
- Critical path for the ripple adder is from input y, through the XOR gate and through the carry circuit of each stage.



Fast Adders

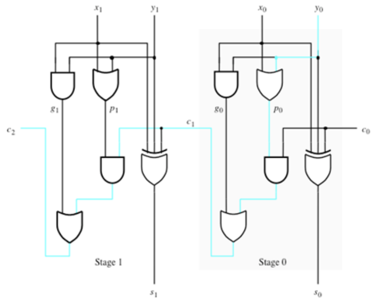
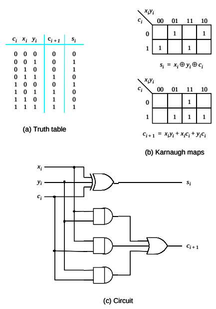


Figure 5.15. A ripple-carry adder based on expression 5.3.

Fast Adders



Fast Adders

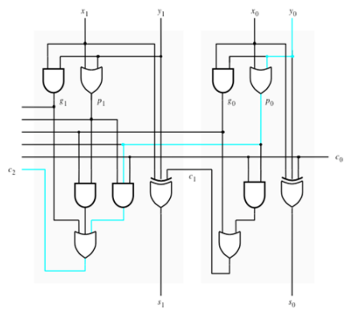


Figure 5.16. The first two stages of a carry-lookahead adder.

Fast Adders

- In an n-bit carry-look ahead adder the final carry-out signal would be produced after three gate delays
- The total delay in an n-bit carry-look ahead adder is four gate delays.
- Complexity of an n-bit carry look ahead adder increases rapidly as n becomes larger
- We can use a hierarchical approach in designing large adders.

Fast Adders

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$
$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0$$

$$c_8 = G_0 + P_0c_0$$

$$c_{16} = G_1 + P_1c_8 = G_1 + P_1G_0 + P_1P_0c_0$$

Fast Adders

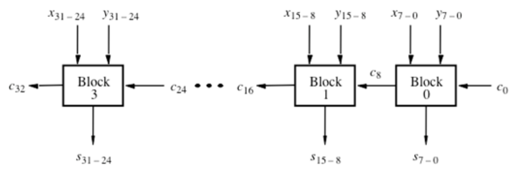
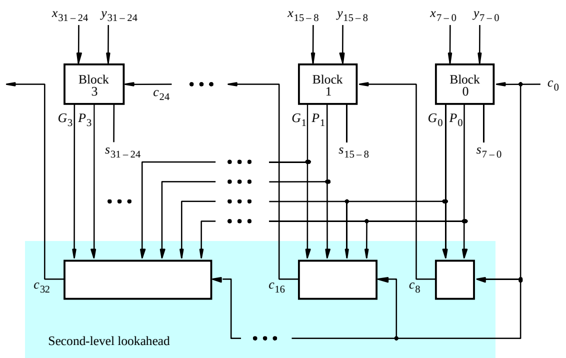


Figure 5.17. A hierarchical carry-lookahead adder with ripple-carry between blocks.

Fast Adders



Fast Adders

- A faster circuit can be designed in which a second-level carry-look-ahead is performed to produce quickly the carry signals between blocks.
- Instead of producing a carry-out signal from the most significant bit of the block, each block produces generate and propagate signals for the entire block

Technology Considerations

- So far we assumed gates with any number of inputs can be used
- Fan-in is limited to a small number
- More gates should be used to implement the logic
- Example: max fan-in is four

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

$$c_8 = (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [p_7p_6p_5p_4(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0$$

- Because fan-in limitation reduces the speed of carry-look-ahead adder, some devices with low fan-in include dedicated circuit for implementing fast adders
- Example: FPGA

Multiplication of unsigned numbers

Each multiplier bit is examined: if 1, a shifted version of the multiplicand is added to form the partial product; if zero nothing is added

```
Multiplicand M   (14)        1 1 1 0
Multiplier Q     (11)      × 1 0 1 1
                          ----------
                             1 1 1 0
                           1 1 1 0
                         0 0 0 0
                       1 1 1 0
                       -------------
Product P       (154)  1 0 0 1 1 0 1 0
```

(a) Multiplication by hand

Multiplication

- A number is multiplied by 2^k by shifting it left by k bit positions
- This is true both for unsigned and signed numbers
- Shifting to the right by k bit, is equivalent to dividing by 2^k
- For unsigned numbers the empty bit positions are filled with zero
- For signed numbers, in order to preserve the sign, the empty bit positions are filled with the sign bit

Multiplication of unsigned numbers

```
Multiplicand M     (11)        1 1 1 0
Multiplier Q       (14)      × 1 0 1 1
Partial product 0              1 1 1 0
                             + 1 1 1 0
Partial product 1            1 0 1 0 1
                           + 0 0 0 0
Partial product 2            0 1 0 1 0
                         + 1 1 1 0
Product P         (154)  1 0 0 1 1 0 1 0
```

(b) Multiplication for implementation in hardware

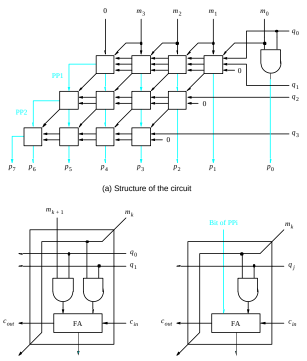
- B=011000=24
- B/2=001100=12
- B/4=000110=6

- B=101000=-24
- B/2=110100=-12
- B/4=111010=-6

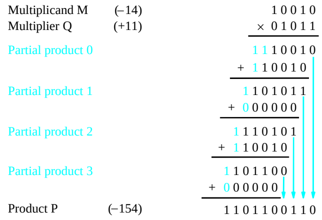
$$M = m_3 m_2 m_1 m_0$$
$$Q = q_3 q_2 q_1 q_0$$

$$PP0 = pp0_3\, pp0_2\, pp0_1\, pp0_0$$

$PP0$	0	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$
$+$	m_3q_1	m_2q_1	m_1q_1	m_0q_1	0
$PP1$	$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$



(a) Structure of the circuit
(b) A block in the top row
(c) A block in the bottom two rows



(b) Negative multiplicand

Multiplication of Signed Numbers

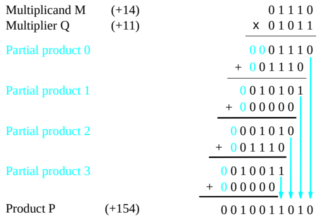
- If multiplier is positive essentially the same scheme as unsigned numbers can be used
- Since shifting the multiplicand to the left results in one of the operands having n+1 bits, the addition has to be performed using the second operand represented in n+1 bits
- An n bit signed number is represented as an n+1 bit number by replicating the sign bit
- Replication of the sign bit is called sign extension

Fixed point

- A fixed point number consists of integer and fraction parts.
- The position of radix point is fixed

$$B = b_{n-1}b_{n-2}\ldots b_1 b_0 .b_{-1}b_{-2}\ldots b_{-k}$$

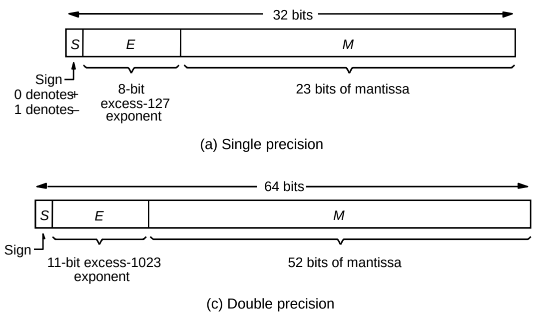
$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$



(a) Positive multiplicand

Floating point

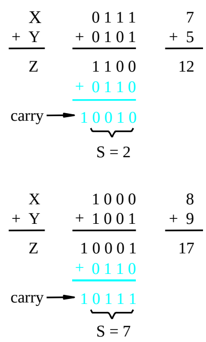
- Fixed point numbers: limited range
- Floating point: numbers are represented by a mantissa and an exponent: Mantissa x $R^{Exponent}$
- Normalized: radix point is the right of fist nonzero digit
- Example: 5.234 x 10^{43}
- For binary R=2
- How mantissa and exponent are represented has been standardized by IEEE
- Single precision (32 bits) and double precision (64 bits)



BCD

- BCD representation was used in some early computers
- Drawback: complexity of circuits that perform arithmetic operations
- BCD addition:
- X and Y two BCD digits (each four bits)
- S=X+Y
- If $X + Y \leq 9$ the addition is the same as the addition of 2 unsigned binary numbers
- If X+Y > 9 the result requires two BDC digits and the four-bit sum may be incorrect.

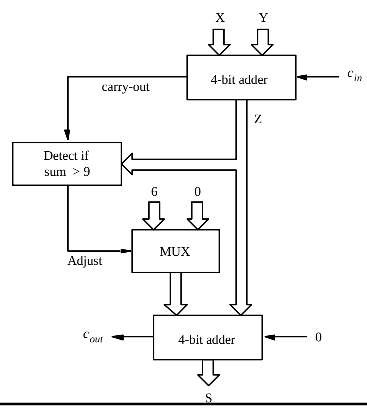
- Single precision
  - Exponent=E-127
  - Value=(+ or -)1.M x2^{E-127}
- Double precision
  - Exponent=E-1023
  - Value=(+ or -)1.M x2^{E-1023}



Binary coded decimal (BCD)

- Each digit in a decimal number is represented by its binary form
- Since there are 10 digits we need 4 bits per digit

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001



ASCII code

- ASCII code: the most popular code for representing information in digital systems used for letters numbers and some control characters.
- Control characters: those needed in computer systems to handle and transfer data, e.g., return character
- ACII representation of numbers is not convenient for arithmetic operations
- It is best to covert ASCII numbers to binary for arithmetic operations

Bit positions 3210	Bit positions 654: 000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgement
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACQ	Acknowledgement	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

ASCII code

- ASCII uses 7-bit, natural size in computer systems is one-byte (8-bits)
- Two common ways on going to 8-bits
  - Set the eight bit to 0
  - Use the eight-bit to indicate the parity of the other bits
- Even parity: the parity bit is given a value such that total number of 1's is even
- Odd parity: the parity bit is given a value such that total number of 1's is odd
- Even parity generator: $p = x_6 \oplus x_5 \oplus \ldots \oplus x_0$
- Parity checker: $c = p \oplus x_6 \oplus x_5 \oplus \ldots \oplus x_0$