CS100 #12

How Does Memory Work

Vadim Surov

Outline

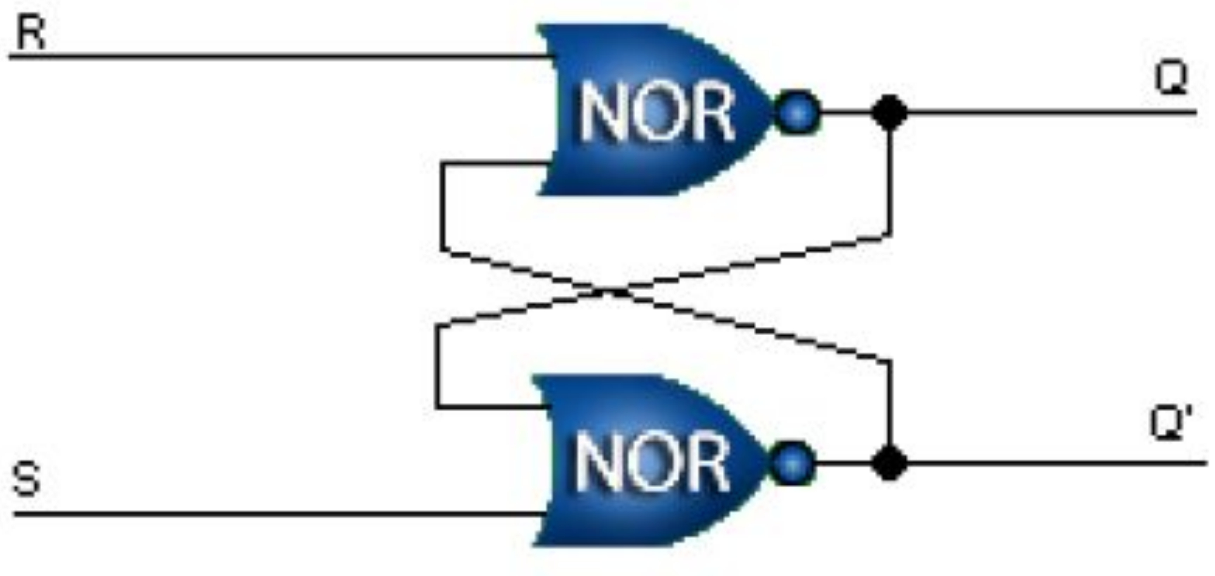
- Combinatorial and Sequential logic
- RS-Flip-Flop
- D-Flip-Flop
- Memory Address And Units
- Layout
- Address decoder
- Storage Elements
- How to store numbers and characters
- Big- and Little-Endian Formats
- Self-test questions

Sequential logic

- **Combinatorial logic** circuits output depends only on the inputs. Ex: half and full adders.
- **Sequential logic** is a type of logic circuit whose output depends not only on the present value of its input signals but on the sequence of past inputs
- Sequential logic has state (memory) while combinational logic does not.

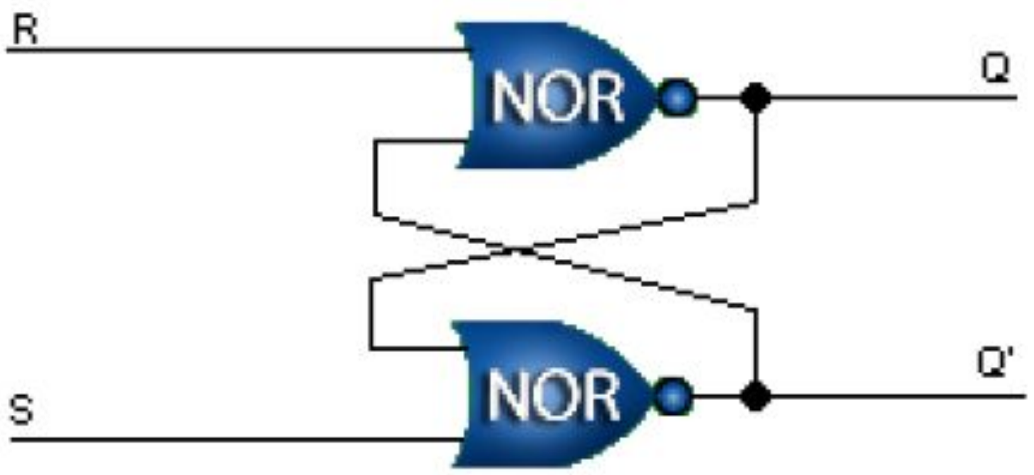
Reset, Set Flip-Flop (RS-Flip-Flop)

- RS-flip-flop is the simplest possible memory element.
- The RS-flip-flop is composed of two NOR gates. (Can also be constructed from NAND gates)



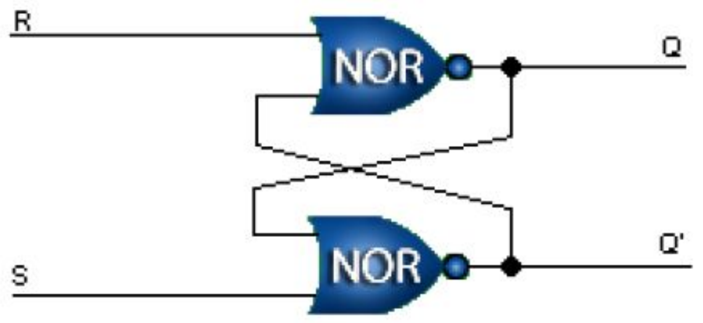
- The output Q is the opposite of Q'.

RS-Flip-Flop Truth Table



R	S	Q	Q’	Description
0	0	Q	Q’	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

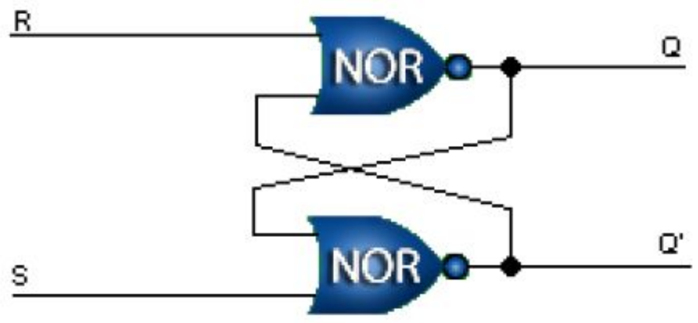
RS-Flip-Flop Truth Table



R	S	Q	Q'	Description
0	0	Q	Q'	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

- When S=1 and R=0.
  - The output at the bottom NOR gate is equal to zero (Q' = 0).
  - Hence, both inputs to the top NOR gate are equal to zero, thus Q =1.
  - Implies that the input combinations S=1 and R=0 leads to the Flip-flop being set to Q=1.

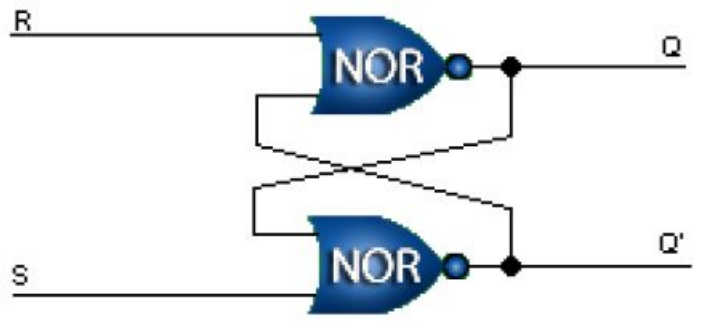
RS-Flip-Flop Truth Table



R	S	Q	Q'	Description
0	0	Q	Q'	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

- When S=0 and R=1.
  - The output becomes Q=0 and Q'=1.
  - We say that the flip-flop is reset.

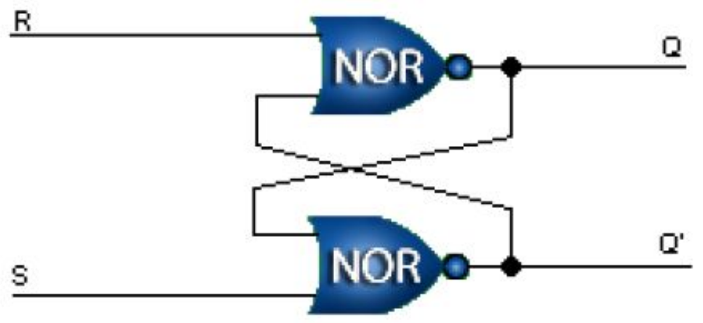
RS-Flip-Flop Truth Table



R	S	Q	Q'	Description
0	0	Q	Q'	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

- When S=0, R=0, Q=0 and Q’=1.
  - The output at the top NOR gate remains at Q=0.
  - The output at the bottom NOR gate stays at Q’=1.

RS-Flip-Flop Truth Table

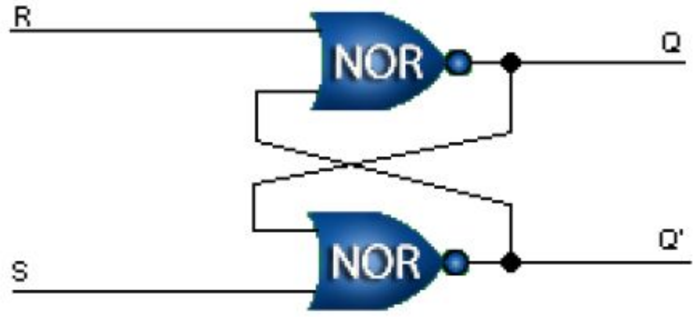


R	S	Q	Q'	Description
0	0	Q	Q'	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

- When S=0, R=0, Q=1 and Q'=0.
  - The output at the top NOR gate remains at Q=1.
  - The output at the bottom NOR gate stays at Q'=0.

- Therefore, when S=0 and R=0, the flip-flop remains in its state.

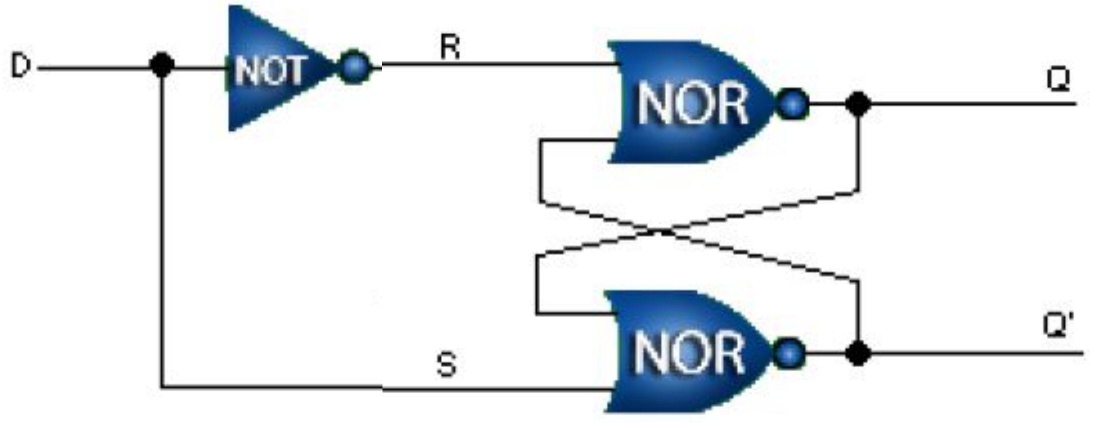
RS-Flip-Flop Truth Table



R	S	Q	Q’	Description
0	0	Q	Q’	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Not Allowed

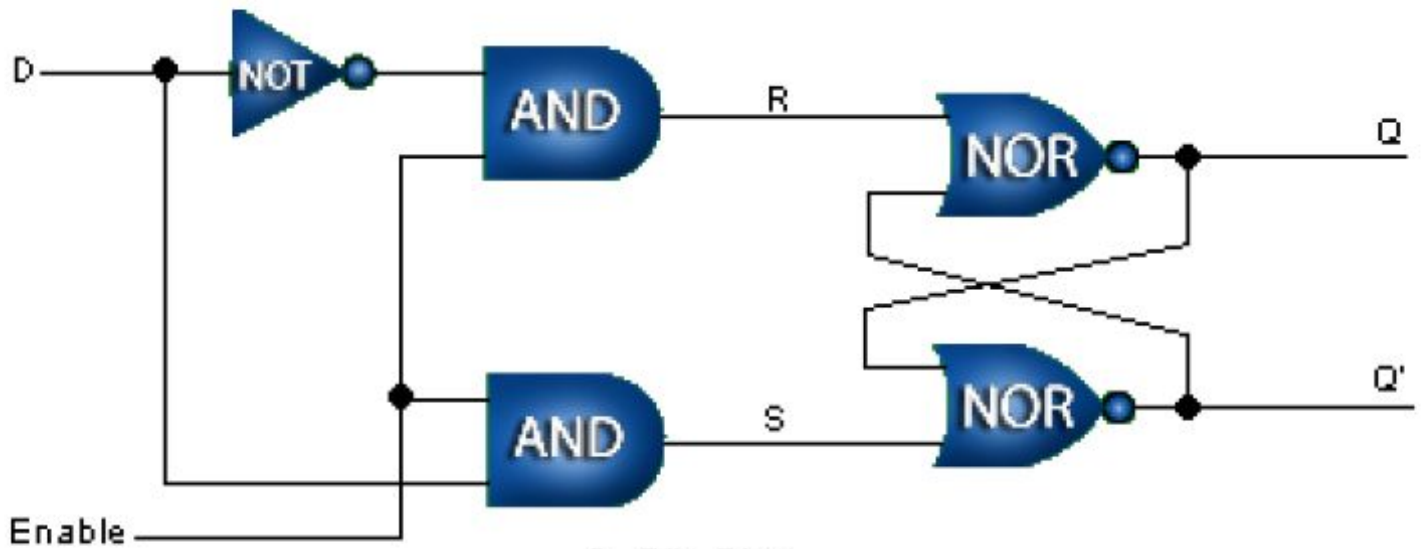
- S=1 and R=1 is not allowed. Therefore, it should be avoided.

Data-Flip-Flop (D-Flip-Flop)



- The D-flip-flop has a single data input.
  - The data input is connected to the S input of an RS-flip-flop.
  - The inverse of the D is connected to the R input.
- The S=1 and R=1 combination will never occur.

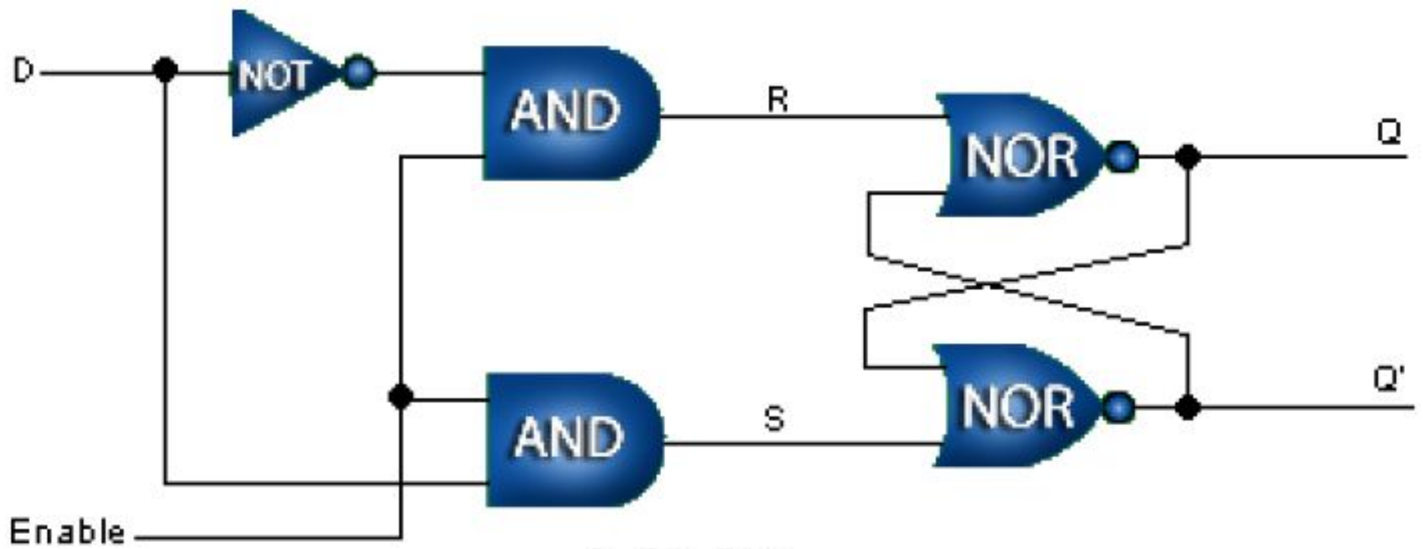
Data-Flip-Flop (D-Flip-Flop)



D - Flip-Flop

- To allow a flip-flop to be in a holding state, an additional Enable input is required.
- The Enable input is AND-ed with the D input.
- When Enable=0, then R=0 and S=0. Therefore, the flip-flop state is held.

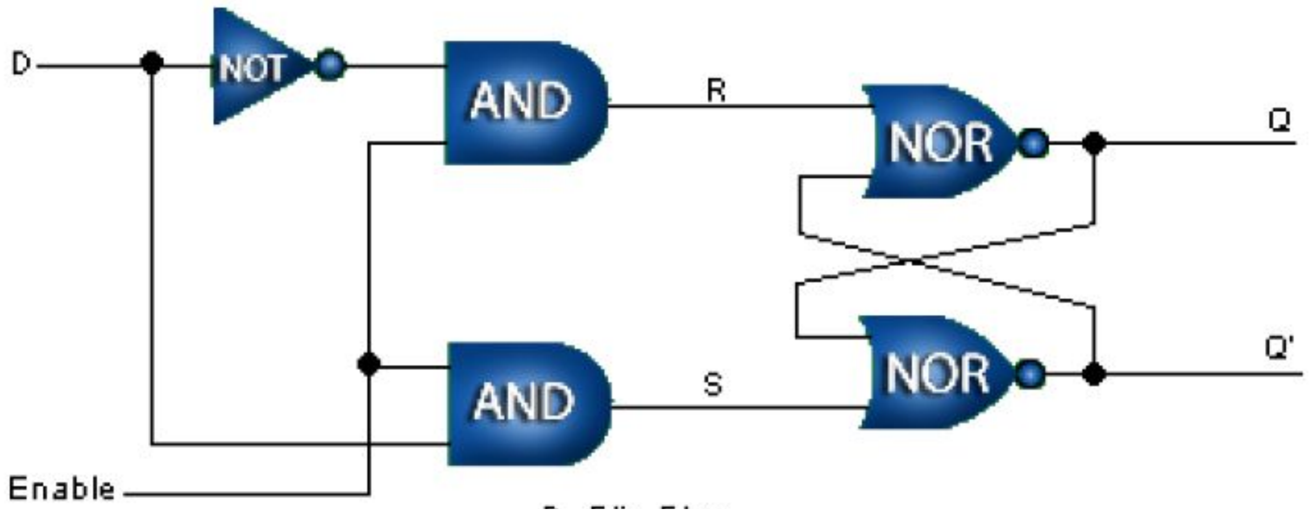
Data-Flip-Flop (D-Flip-Flop)



D - Flip-Flop

- When Enable=1, S=D and R is the inverse of D, then the value of D determines the value of the output Q when Enable=1.
- When Enable returns to 0, the most recent input D is remembered.

D-Flip-Flop Truth Table



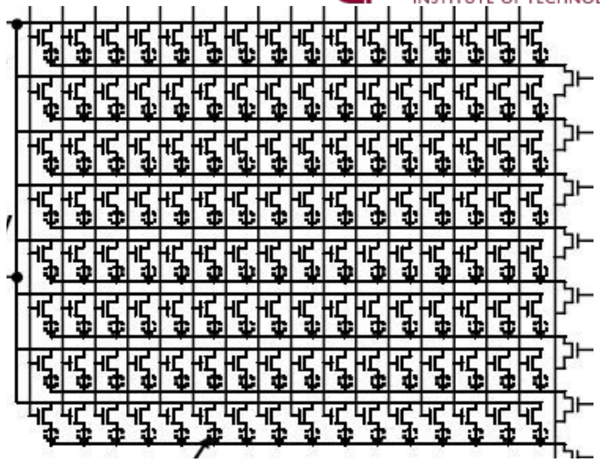
E	D	Q	Q’	Description
0	0	Q	Q’	Hold state
0	1	Q	Q’	Hold state
1	0	0	1	Reset
1	1	1	0	Set

Memory Address And Units

- RAM is made up of **storage elements**, 8-bits each.
- Storage elements could be similar to the D-flip-flop.
- Every storage element has a unique address as a number to read from or write to a particular storage element.

- **Byte** = 8 bits the smallest addressable unit for a CPU.
- **Word** - the natural size with which a processor is handling data, for example, using registers. The most common word sizes encountered today are 32 and 64 bits.
- 1024 bytes = 1 **Kilobyte** (KB). 1024 KB = 1 **Megabyte** (MB). 1024 MB = 1 **Gigabyte** (GB).

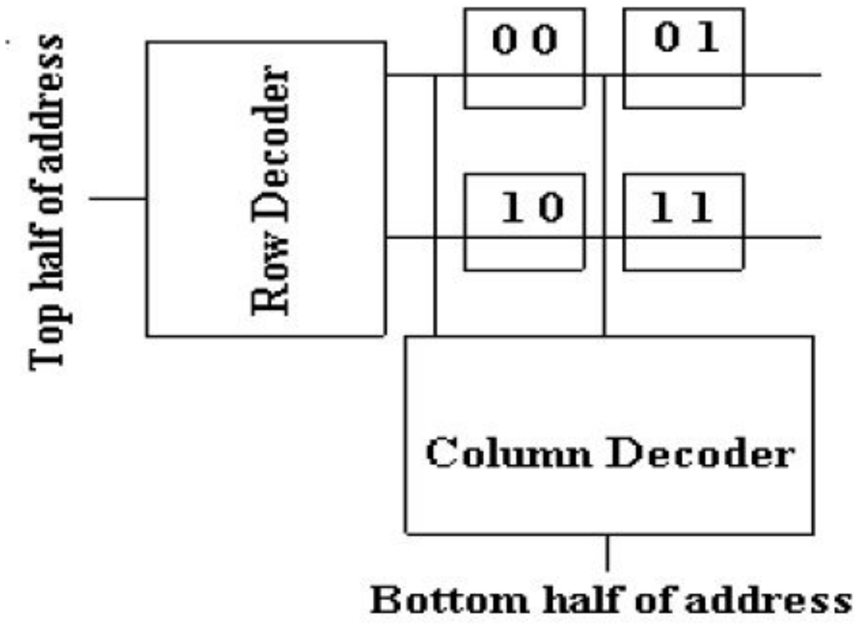
Layout



- Storage elements are arranged on a square grid of columns and rows.
- Ex: 1 Megabit = 128 Kilobyte of RAM consists of 1024 rows and 1024 columns of storage elements.
- Each individual storage element can be identified through its coordinates. In other words, **individual storage elements can be identified by their row and column**.
- To address a particular storage element, the address information must be translated into row and column specification.

Layout

- The address information is divided into two parts.
  - The top half is used to select the row.
  - The bottom half is used to select the column.
- A 4-bit RAM figure will look like this:

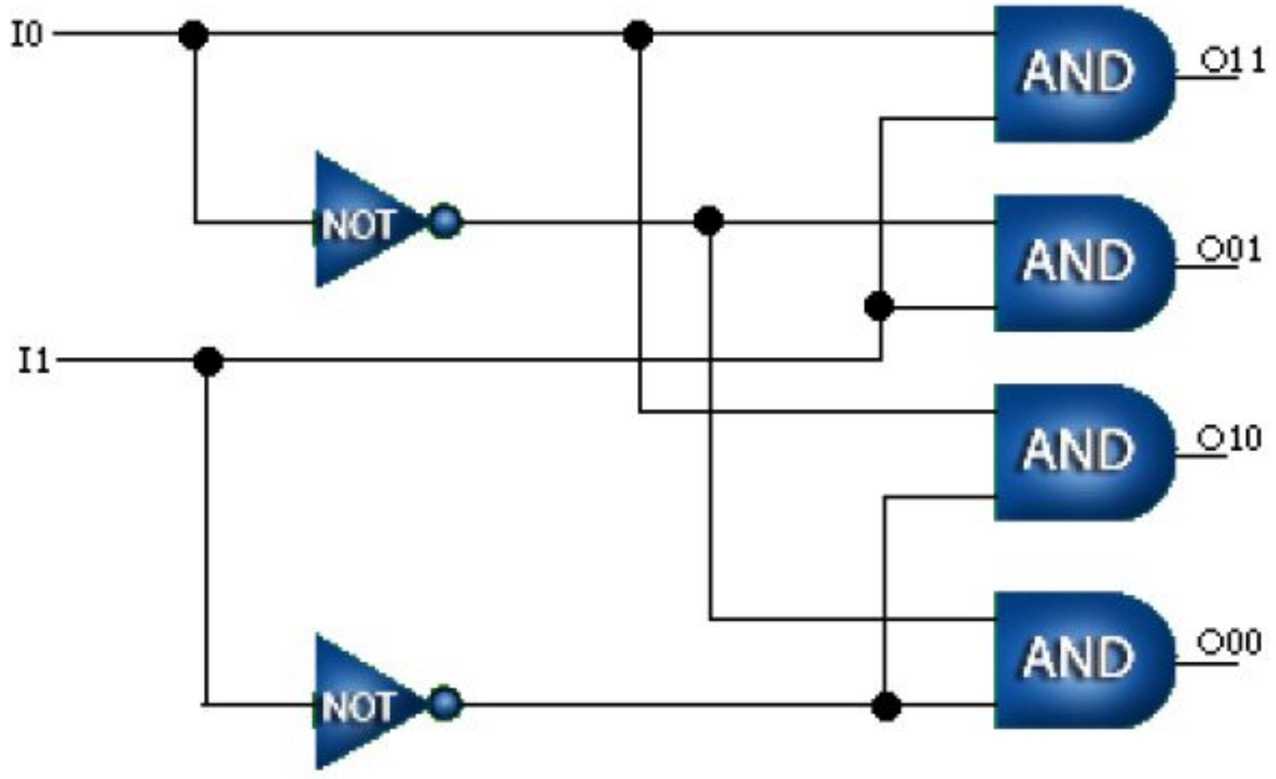


Address decoder

- The address decoder has a binary number with N bits as its input. It has 2^N outputs.
- At any time, only one output line is ‘1’ and all the others are ‘0s’.
- The line that is ‘1’ specifies the desired column or row.

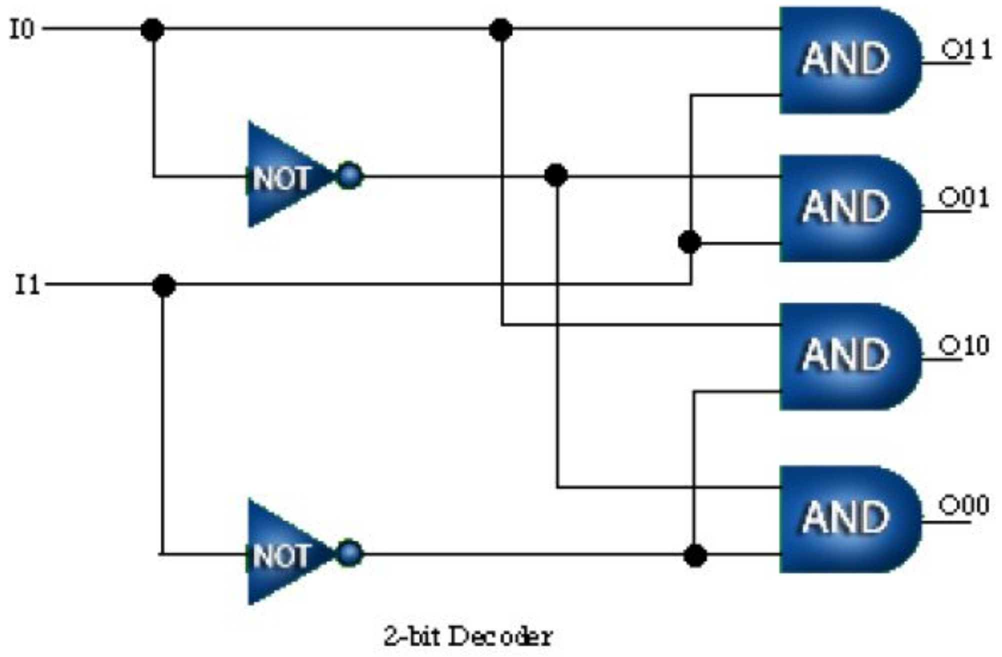
Address decoder

- Example of a decoder with 2 inputs and 4 (=2^2) outputs:



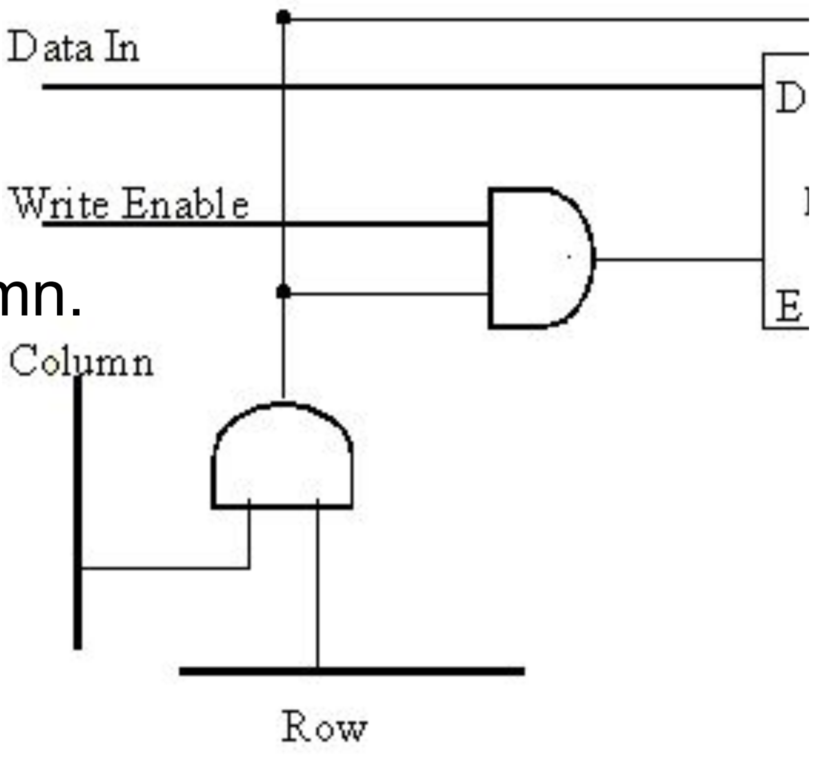
2-bit Decoder

Address decoder. Truth Table



Input		Output			
I0	I1	O11	O10	O01	O00
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

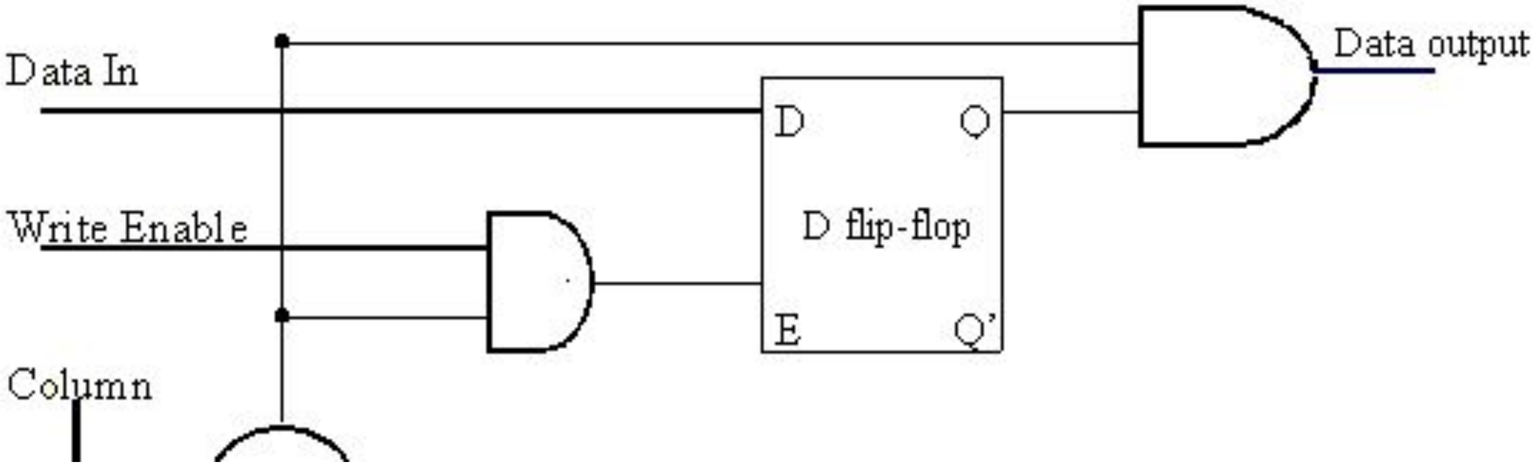
Storage Elements



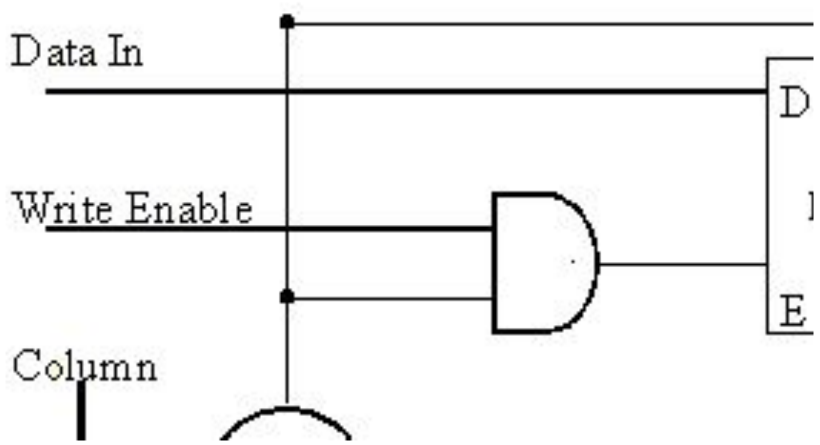
- Each storage element is connected to one row and column.
- Since for every address, only one row and one column selector line will be '1', exactly one storage element can be selected.
- The storage element is selected by AND-ing its row and its column selector line.
- If a D-flip-flop is used as a storage element and connect its Enable input to the output of the AND gate mentioned above.

Storage Elements

- The D-input is connected to the data input line that is common to all storage elements.
- The output of the flip-flop is AND-ed with the output of the first AND gate and then connected to a data output line that is common to all storage elements.
- Only the selected storage element contributes to the common output line.

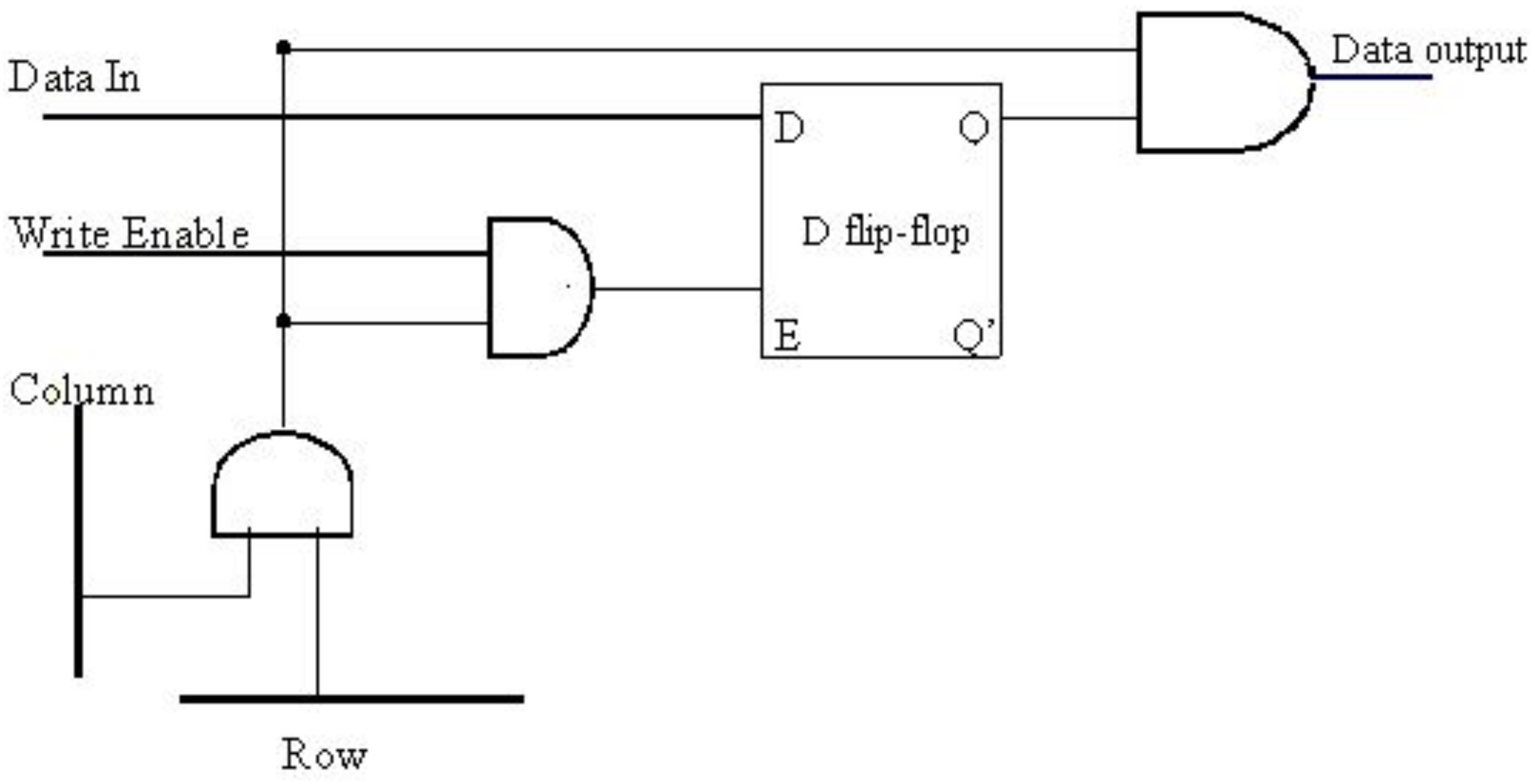


Storage Elements



- The read process should Leave the value of the gate unchanged.
- To prevent data destruction during a read operation a ‘Write Enable’ signal is introduced.
- The signal is ‘1’ only when new data is to be stored.
- It is AND-ed with the output of the first AND gate and then applied to the flip-flop’s enable input.
- During read operations, the ‘Write Enable’ is ‘0’, the flip-flop is disabled and does not change its state.

Storage Elements

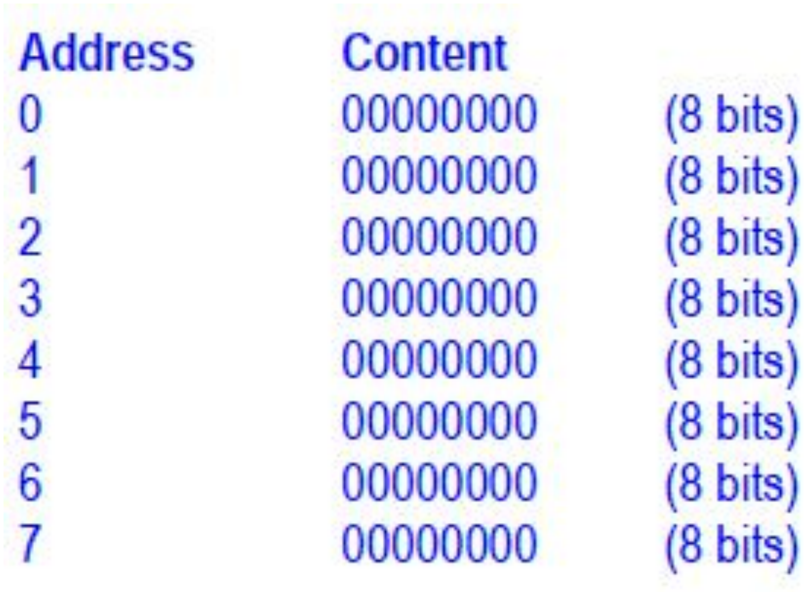


How to store numbers in memory

- A computer machine’s memory is an array of consecutively numbered or addressed memory cells holding a bit value.
- Every byte in a machine memory has a unique number or address.

How to store numbers in memory. Example 1

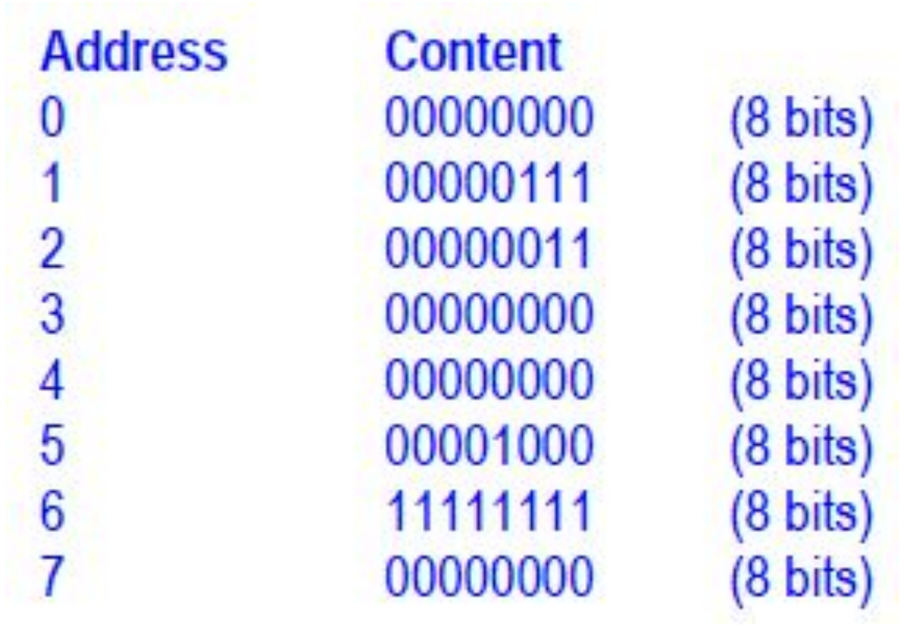
- If we have a memory size of 8 bytes, memory address 0 will point to the first byte, memory address 1 will point to the second byte and memory address 7 will point to the last byte, byte number 8. This is what the empty memory will look like:



Address	Content	
0	00000000	(8 bits)
1	00000000	(8 bits)
2	00000000	(8 bits)
3	00000000	(8 bits)
4	00000000	(8 bits)
5	00000000	(8 bits)
6	00000000	(8 bits)
7	00000000	(8 bits)

How to store numbers in memory. Example 2

- This is what the memory will look like after we store: value 7 in address 1, value 3 in address 2, value 8 in address 5, and value 255 in address 6.

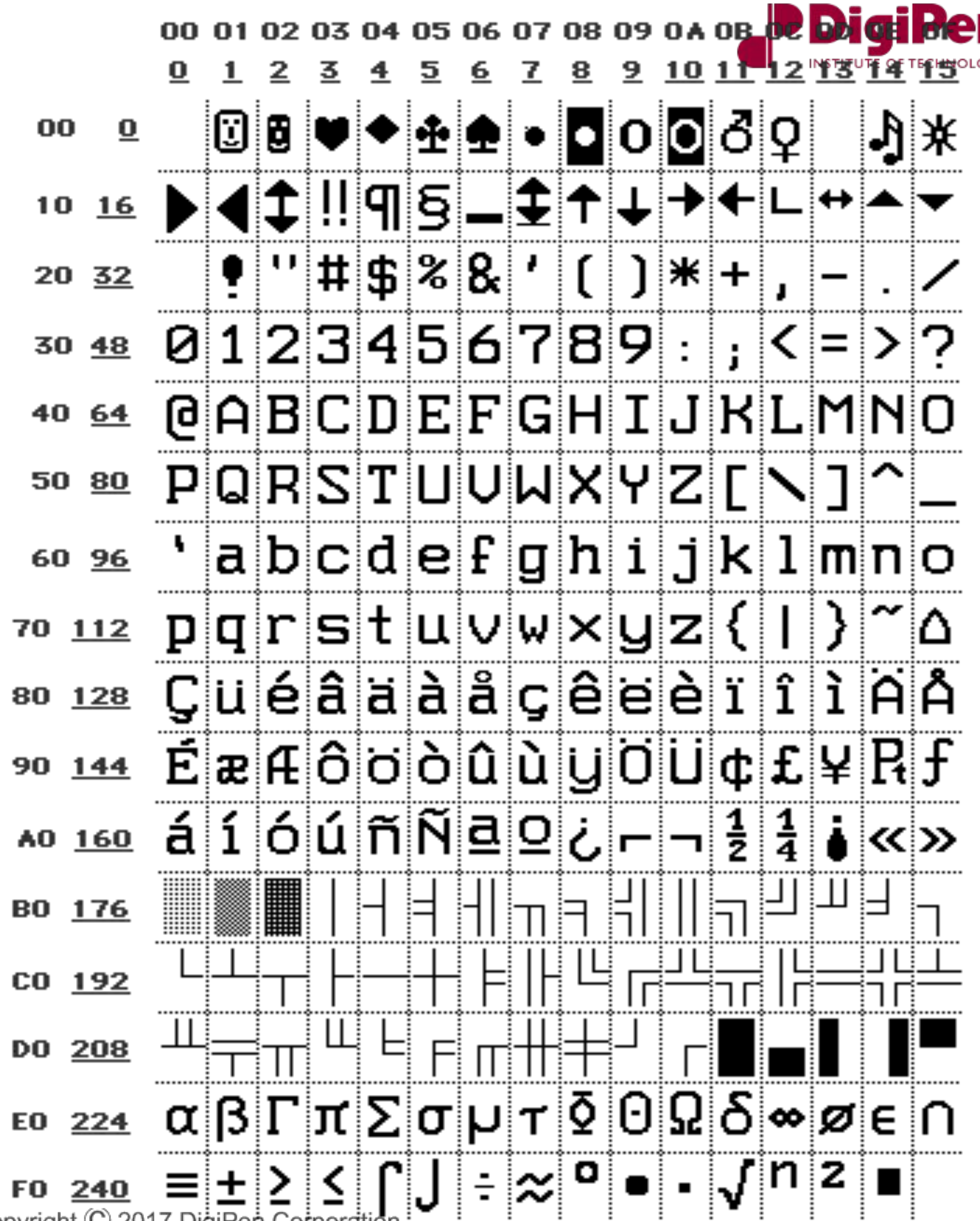


Address	Content	
0	00000000	(8 bits)
1	00000111	(8 bits)
2	00000011	(8 bits)
3	00000000	(8 bits)
4	00000000	(8 bits)
5	00001000	(8 bits)
6	11111111	(8 bits)
7	00000000	(8 bits)

How to store characters in memory

- Since the computer machine can only understand binary numbers, the character set is represented by number.
- Example: Upper case character A is represented by the number 65 in base 10 which is 01000001 in binary.
- The ASCII character code tables contain the decimal values of the extended ASCII (American Standards Committee for Information Interchange) character set.
- The extended character set includes the ASCII character set and 128 other characters for graphics and line drawing, often called the “IBM ® character set.”

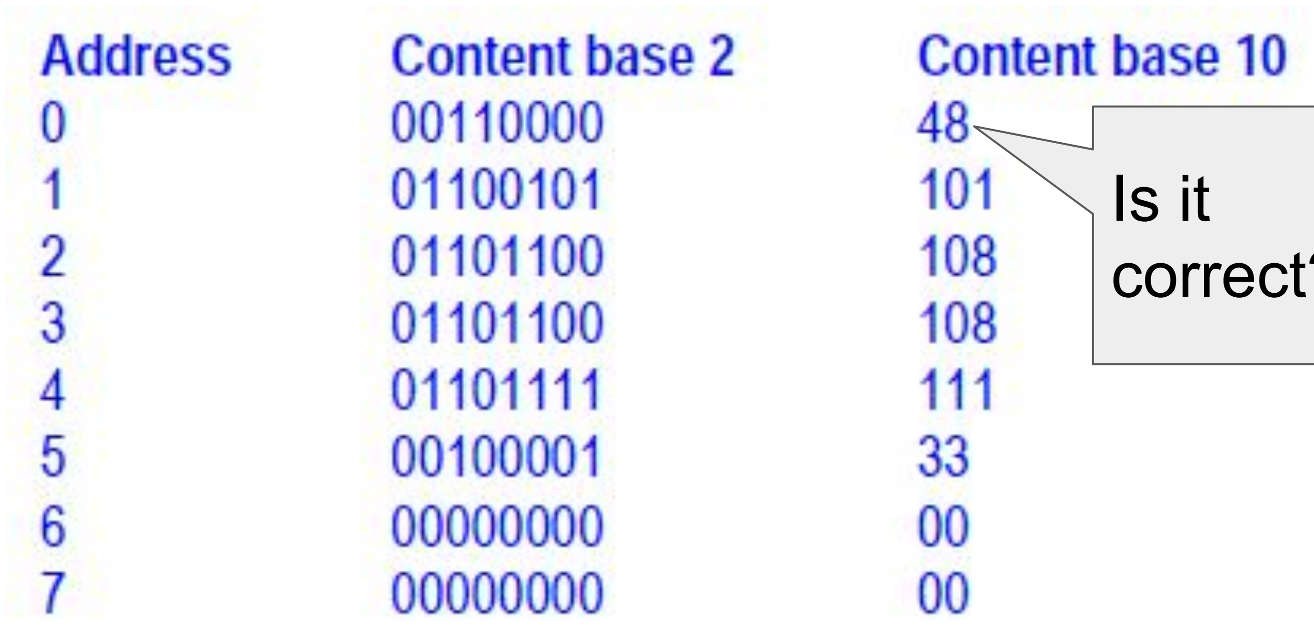
IBM All Character 437 Set (ANSI)



	00 0	01 1	02 2	03 3	04 4	05 5	06 6	07 7	08 8	09 9	0A 10	0B 11	0C 12	0D 13	0E 14	0F 15
00 0		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀		♫	☼
10 16	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20 32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30 48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40 64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50 80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60 96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70 112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
80 128	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
90 144	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A0 160	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B0 176	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C0 192	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D0 208	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E0 224	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F0 240	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

How to store characters in memory. Example

- If we want to write the word “Hello!” in a memory of size 8 bytes.



Address	Content base 2	Content base 10
0	00110000	48
1	01100101	101
2	01101100	108
3	01101100	108
4	01101111	111
5	00100001	33
6	00000000	00
7	00000000	00

Is it correct?

Big- and Little-Endian Formats

- Computers are designed around two different architectures based on the order in which bytes are stored in memory:
    - **Big-Endian** (from ‘Big End In’) and
    - **Little-Endian**.
- On an Intel based CPU computer, the little-end (least significant byte) is stored first.
- On a Motorola based CPU computer, the big-end (most significant byte) is stored first.

Big- and Little-Endian Formats

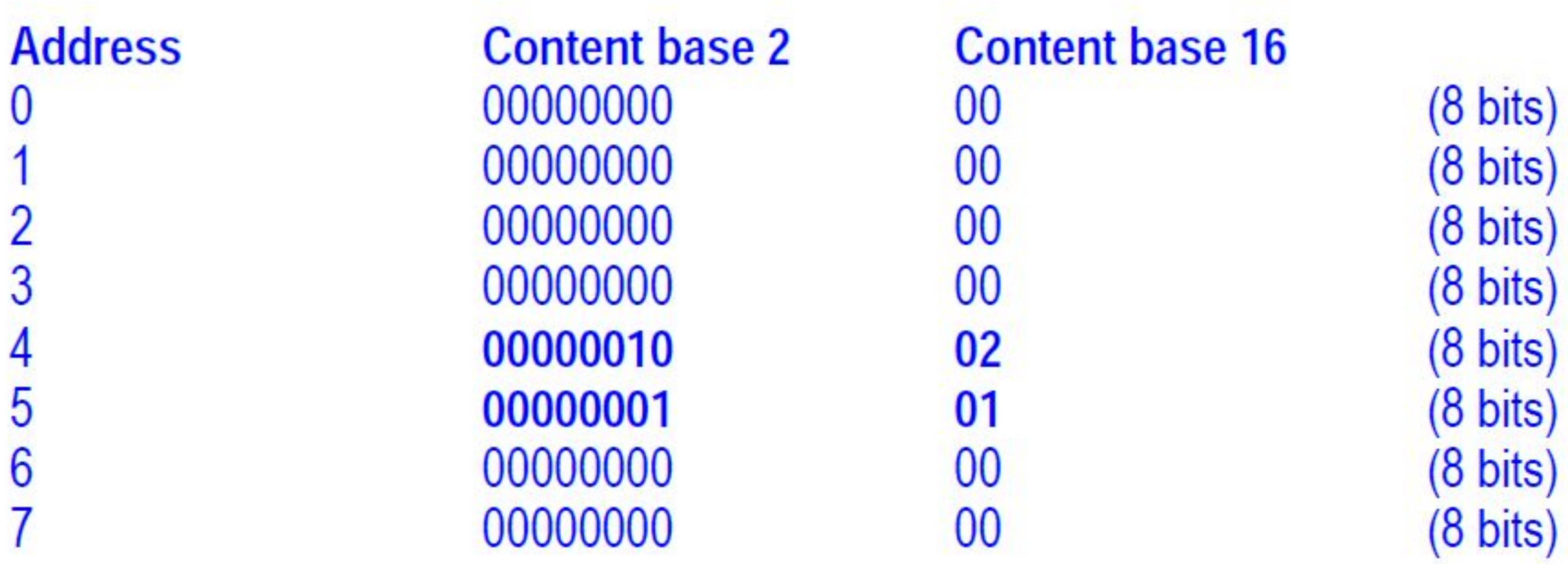
- IEEE 754 floating-point standard does not specify endianness
- We will assume that the endianness is the same for floating-point numbers as for integers

Big- and Little-Endian Formats

- Little-Endian Example:
  - A 4-byte value like 0x87654321 would be stored as 0x21 0x43 0x65 0x87.
- Big-Endian Example:
  - A 4-byte value like 0x87654321 would be stored as 0x87 0x65 0x43 0x21.

Big- and Little-Endian Formats

- Let's see what the Little-Endian based memory will look like after we store value $258_{10} = 00000001\ 00000010_2 = 0102_{16}$ in address 4.



Address	Content base 2	Content base 16	
0	00000000	00	(8 bits)
1	00000000	00	(8 bits)
2	00000000	00	(8 bits)
3	00000000	00	(8 bits)
4	**00000010**	**02**	(8 bits)
5	**00000001**	**01**	(8 bits)
6	00000000	00	(8 bits)
7	00000000	00	(8 bits)

Big- and Little-Endian Conversion

```
/* C function to change endianness for byte
swap in an unsigned 32-bit integer */

uint32_t ChangeEndianness(uint32_t value)
{
    uint32_t result = 0;
    result |= (value & 0x000000FF) << 24;
    result |= (value & 0x0000FF00) << 8;
    result |= (value & 0x00FF0000) >> 8;
    result |= (value & 0xFF000000) >> 24;
    return result;
}
```

Big- and Little-Endian Determination

```
int is_big_endian(void)
{
    union {
        uint32_t i;
        char c[4];
    } e = { 0x01000000 };

    return e.c[0];
}
```

Self-test question

The unsigned integer 3,505,468,161 can be written in 32-bit binary as 11010000 11110001 00110011 00000001. Putting it into four bytes of memory beginning at address 98370 in little endian fashion would give which picture?

○	98370	98371	98372	98373
	11010000	111100001	00110011	00000001

○	98370	98371	98372	98373
	00000001	111100001	00110011	11010000

○	98370	98371	98372	98373
	00000001	00110011	111100001	11010000

○	98370	98371	98372	98373
	00110011	00000001	11010000	111100001

Self-test question

Why is it necessary to know whether a computer is big-endian or little-endian?

A) because some programs write integers to memory in a certain order
B) because arithmetic errors can result if the computer gets the numbers backward
C) because sharing files and data between different computers can result in misinterpretation

References

- https://en.wikipedia.org/wiki/Address_decoder
- https://en.wikipedia.org/wiki/ASCII
- https://en.wikipedia.org/wiki/Endianness