CS 303
Logic & Digital System Design

Ömer Ceylan

Sabancı Üniversitesi

Binary Systems

Binary Numbers 1/2

- Internally, information in digital systems is of binary form
  - groups of bits (i.e. binary numbers)
  - all the processing (arithmetic, logical, etc) are performed on binary numbers.
- Example: 4392
  - In decimal, 4392 = $4 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$.
  - Convention: write only the coefficients.
  - $A = a_1 a_0 . a_{-1} a_{-2} a_{-3}$ where $a_j \in \{0, 1, \ldots, 9\}$

Binary Numbers 2/2

- Decimal system
  - coefficients are from {0,1, …, 9}
  - and coefficients are multiplied by powers of 10
  - base-10 or radix-10 number system
- Using the analogy, binary system {0,1}
  - base(radix)-2
- Example: 25.625
  - $25.625 = 2\times10^{1} + 5\times10^{0} + 6\times10^{-1} + 2\times10^{-2} + 5\times10^{-3}$
  - $25.625 = 1\times2^{4} + 1\times2^{3} + 1\times2^{0} + 1\times2^{-1} + 1\times2^{-3}$
  - $25.625 = (11001.101)_2$

Base-r Systems

- base-r (n, m)
  - $A = a_{n-1}\, r^{n-1} + \ldots + a_1\, r^1 + a_0\, r^0 + a_{-1}\, r^{-1} + a_{-2}\, r^{-2} + \ldots + a_{-m}\, r^{-m}$
- Octal
  - base-8 = base-2^3
  - digits {0,1, …, 7}
  - <u>Example</u>: $(31.5)_8$ = octal expansion =
- Hexadecimal
  - base-16
  - digits {0, 1, …, 9, A, B, C, D, E, F}
  - <u>Example</u>:
  - $(19.A)_{16}$ = hexadecimal expansion =

Powers of 2

- 2^{10} = 1,024 (K) -
- 2^{20} = 1,048,576 (M) -
- $2^{30} \rightarrow$ (G) -
- $2^{40} \rightarrow$ (T) -
- $2^{50} \rightarrow$ (P) -
- exa, zetta, yotta, … (exbi, zebi, yobi, ...)
- Examples:
  - A byte is 8-bit, i.e. 1 B
  - 16 GB = ?? B = 17,179,869,184

Arithmetic with Binary Numbers

	10101	**21**	augend
+	**10011**	**19**	addend
1	**01000**	**40**	sum

	10101	**21**	minuend
-	**10011**	**19**	subtrahend
0	**00010**	**2**	difference

			0	**0**	**1**	**0**	multiplicand (2)
		×	**1**	**0**	**1**	**1**	multiplier (11)
			0	**0**	**1**	**0**	
		0	**0**	**1**	**0**		
	0	**0**	**0**	**0**			
+ 0	**0**	**1**	**0**				
0	**0**	**1**	**0**	**1**	**1**	**0**	product (22)

Multiplication with Octal Numbers

				3	**4**	**5**	**229**	multiplicand
			×	**6**	**2**	**1**	**401**	multiplier
				3	**4**	**5**		
			7	**1**	**2**			
+	**2**	**5**	**3**	**6**				
	2	**6**	**3**	**2**	**6**	**5**	**91829**	product

Base Conversions

- From base-*r* to decimal is easy
  - expand the number in power series and add all the terms
- Reverse operation is somewhat more difficult
- Simple idea:
  - divide the decimal number successively by *r*
  - accumulate the remainders
- If there is a fraction, then integer part and fraction part are handled separately.

Base Conversion Examples 1/3

- Example 1:
  - 55
  - (decimal to binary)

– Example 2:
  - 144
  - (decimal to octal)

Base Conversion Examples 2/3

- Example 1: 0.6875 (decimal to binary)
  - When dealing with fractions, instead of dividing by *r* multiply by *r* until we get an integer
  - $0.6875 \times 2 = 1.375 \rightarrow 1$
  - $0.375 \times 2 = 0.750 \rightarrow 0$
  - $0.750 \times 2 = 1.5 \rightarrow 1$
  - $0.5 \times 2 = 1.0 \rightarrow 1$
  - $0.6875 = (0.1011)_2$

Base Conversion Examples 2/3

- We are not always this lucky
- Example 2: (144.478) to octal
  - Treat the integer part and fraction part separately
  - $0.478 \times 8 = 3.824 = 3 + 0.824 \rightarrow a_{-1} = 3$
  - $0.824 \times 8 = 6.592 = 6 + 0.592 \rightarrow a_{-2} = 6$
  - $0.592 \times 8 = 4.736 = 4 + 0.736 \rightarrow a_{-3} = 4$
  - $0.736 \times 8 = 5.888 = 5 + 0.888 \rightarrow a_{-4} = 5$
  - $0.888 \times 8 = 7.104 = 7 + 0.104 \rightarrow a_{-5} = 7$
  - $0.104 \times 8 = 0.832 = 0 + 0.832 \rightarrow a_{-6} = 0$
  - $0.832 \times 8 = 6.656 = 6 + 0.656 \rightarrow a_{-7} = 6$
  - $144.478 = (220.3645706\ldots)_8$

Conversions between Binary, Octal and Hexadecimal

- *r* = 2 (binary), *r* = 8 (octal), *r* = 16 (hexadecimal)

```
10110001101001.101100010111

10 110 001 101 001.101 100 010 111

10 1100 0110 1001.1011 0001 0111
```

- Octal and hexadecimal representations are more compact.
- Therefore, we use them in order to communicate with computers directly using their internal representation

Complement

- Complementing is an operation on base-*r* numbers
- <u>Goal</u>: To simplify subtraction operation
  - Rather turn the subtraction operation into an addition operation
- Two types
  1. Radix complement (a.k.a. *r*’s complement)
  2. Diminished complement (a.k.a. (*r*-1)’s complement)
- When $r = 2$
  1. 2’s complement
  2. 1’s complement

How to Complement?

- A number N in base-r (n-digit)
  1. $r^n - N$ r's complement
  2. $(r^n\text{-}1) - N$ $(r$-1)'s complement
  - where n is the number of digits we use
- Example: $r = 2$, $n = 4$, $N = 7$
  - $r^n = 2^4 = 16$, $r^n - 1 = 15$.
  - 2's complement of 7 → 16-7 = 9
  - 1's complement of 7 → 15-7 = 8
- Easier way to compute 1's and 2's complements
  - Use binary expansions
  - 1's complement: negate
  - 2's complement: negate + increment

Subtraction with Complements 1/3

- Conventional subtraction
  - Borrow concept
  - If the minuend digit is smaller than the subtrahend digit, you borrow “1” from *a* digit in higher significant position
- With complements
  - M-N = ?
  - $r^n - N$ r’s complement of N
  - $M + (r^n - N) =$

Subtraction with Complements 2/3

- $M\text{-}N \rightarrow M + (r^n - N)$
  - $M + (r^n - N) = M - N + r^n$

1. if $M \geq N$,
   - the sum will produce a carry, that can be discarded
2. Otherwise,
   - the sum will not produce a carry, and will be equal to $r^n - (N\text{-}M)$, which is the r's complement of $N\text{-}M$
   - Since $M - N + r^n = r^n - (N\text{-}M)$

Subtraction with Complements 3/3

- Example:
  - X = 1010100 (84) and Y = 1000011 (67)
  - X-Y = ? and Y-X = ?

```
                              X      1010100
2's complement of Y                + 0111101
                                   ---------
                                    10010001

                              Y      1000011
2's complement of X                + 0101100
                                   ---------
                                    01101111
```

Signed Binary Numbers

- Pencil-and-paper
  - Use symbols “+” and “-”
- We need to represent these symbols using bits
  - Convention:
    - 0 positive
      1 negative
    - The leftmost bit position is used as a sign bit
  - In <u>signed representation</u>, the leftmost bit is the sign bit
  - In <u>unsigned representation</u>, the leftmost bit is a part of the number (i.e., the most significant bit (MSB))

Signed Number Representation

Signed magnitude		One's complement		Two's complement	
000	+0	000	+0	000	0
001	+1	001	+1	001	+1
010	+2	010	+2	010	+2
011	+3	011	+3	011	+3
100	-0	111	-0	111	-1
101	-1	110	-1	110	-2
110	-2	101	-2	101	-3
111	-3	100	-3	100	-4

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

Which One?

- Signed magnitude:
  - There are two representations for 0.
  - Adders may need an additional step to set the sign
- Try to subtract a large number from a smaller one.

```
2 = 0 0 1 0
5 = 0 1 0 1
  = 1 1 0 1
```

- 2's complement provides a natural way to represent signed numbers (every computer today uses two's complement)
- Think that there is an infinite number of 1's in a signed number

  -3 = 1101 ≡ …11111101

- What is **11111100**?

Arithmetic Addition

- Examples:

```
+11        00001011        -11        11110101
 +9    +   00001001         +9    +   00001001
       ------------              ------------


+11        00001011        -11        11110101
 -9    +   11110111         -9    +   11110111
       ------------              ------------
```

- No special treatment for sign bits

Arithmetic Overflow 1/2

- In hardware, we have limited resources to accommodate numbers (precision)
  - Computers use 8-bit, 16-bit, 32-bit, and 64-bit registers for the operands in arithmetic operations.
  - Sometimes the result of an arithmetic operation get too large to fit in a register.

Arithmetic Overflow 2/2

- Example:

```
+2      0010
+4  +   0100
    ----------
```

```
-3      1101
-5  +   1011
    ----------
```

```
+7      0111
+6  +   0110
    ----------
```

```
-3      1101
-6  +   1010
    ----------
```

Subtraction with Signed Numbers

- Rule: is the same
- We take the 2's complement of the subtrahend
  - It does not matter if the subtrahend is a negative number.
  - (±*A*) - (-*B*) = ±*A*+*B*

```
 -6       11111010
-13  -    11110011
     ______________
```

```
 -6       11111010
     +    00001101
     ______________
          00000111
```

- Signed-complement numbers are added and subtracted in the same way as unsigned numbers
- With the same circuit, we can do both signed and unsigned arithmetic

Alphanumeric Codes

- Besides numbers, we have to represent other types of information
  - letters of alphabet, mathematical symbols.
- For English, alphanumeric character set includes
  - 10 decimal digits
  - 26 letters of the English alphabet (both lowercase and uppercase)
  - several special characters
- We need an alphanumeric code
  - ASCII
  - American Standard Code for Information Exchange
  - Uses 7 bits to encode 128 characters

ASCII Code

- 7 bits of ASCII Code
  - $(b_6 b_5 b_4 b_3 b_2 b_1 b_0)_2$
- Examples:
  - *A* → **65 = (1000001),...,** *Z* → **90 = (1011010)**
  - *a* → **97 = (1100001),...,** *z* → **122 = (1111010)**
  - **0→48 = (0110000),..., 9 → 57 = (0111001)**
- 128 different characters
  - 26 + 26 + 10 = 62 (letters and decimal digits)
  - 32 special printable characters %, *, $
  - 34 special control characters (non-printable): BS, CR, etc.

Binary Logic

- Binary logic is equivalent to what it is called “two-valued Boolean algebra”
  - Or we can say that it is an implementation of two-valued Boolean algebra
- Deals with variables that take on “two discrete values” and operations that assume logical meaning
- Two discrete values:
  - {true, false}
  - {yes, no}
  - {1, 0}

Binary Variables and Operations

- We use A, B, C, x, y, z, etc. to denote binary variables
  - each can take on {0, 1}
- Logical operations
  1. AND → $x \cdot y = z$ or $xy = z$
  2. OR → $x + y = z$
  3. NOT → $x = z$ or $x' = z$
  - For each combination of the values of x and y, there is a value of specified by the definition of the logical operation.
  - This definition may be listed in a compact form called <u>truth table</u>.

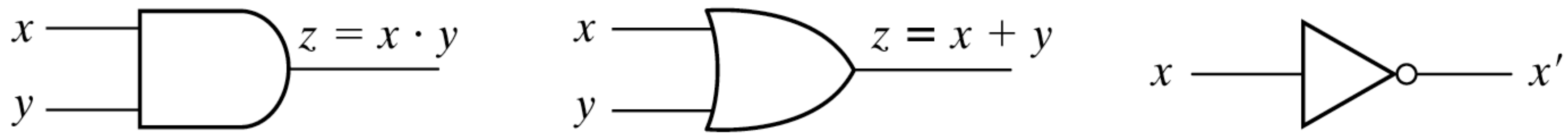
Truth Table

x	y	AND $x \cdot y$	OR $x + y$	NOT x'
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Logic Gates

- Binary values are represented as electrical signals
  - Voltage, current
- They take on either of two recognizable values
  - For instance, voltage-operated circuits
  - **0V** $\rightarrow$ **0**
  - **4V** $\rightarrow$ **1**
- Electronic circuits that operate on one or more input signals to produce output signals
  - **AND** gate, **OR** gate, **NOT** gate

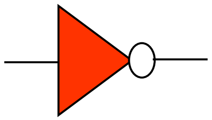
Logic Gate Symbols



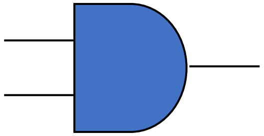
(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

Fig. 1-4 Symbols for digital logic circuits

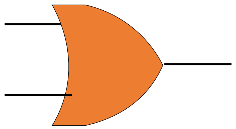
Logic Gate Symbols



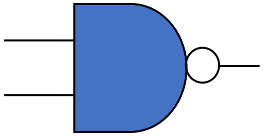
NOT



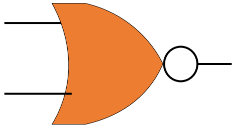
AND



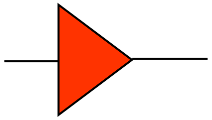
OR



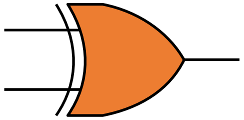
NAND



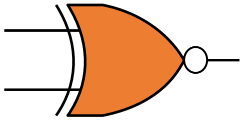
NOR



TRANSFER



XOR



XNOR

Range of Electrical Signals

- What really matters is the range of the signal value

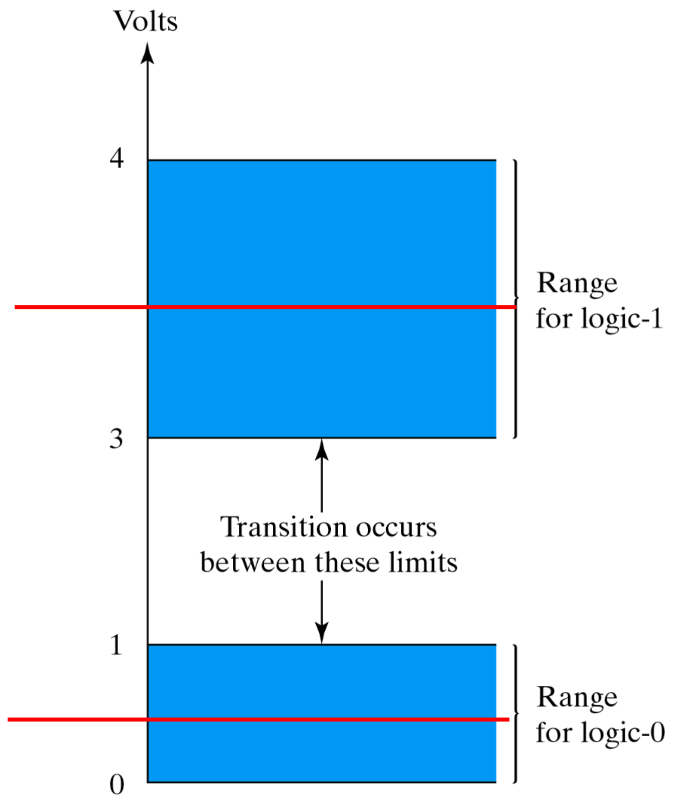
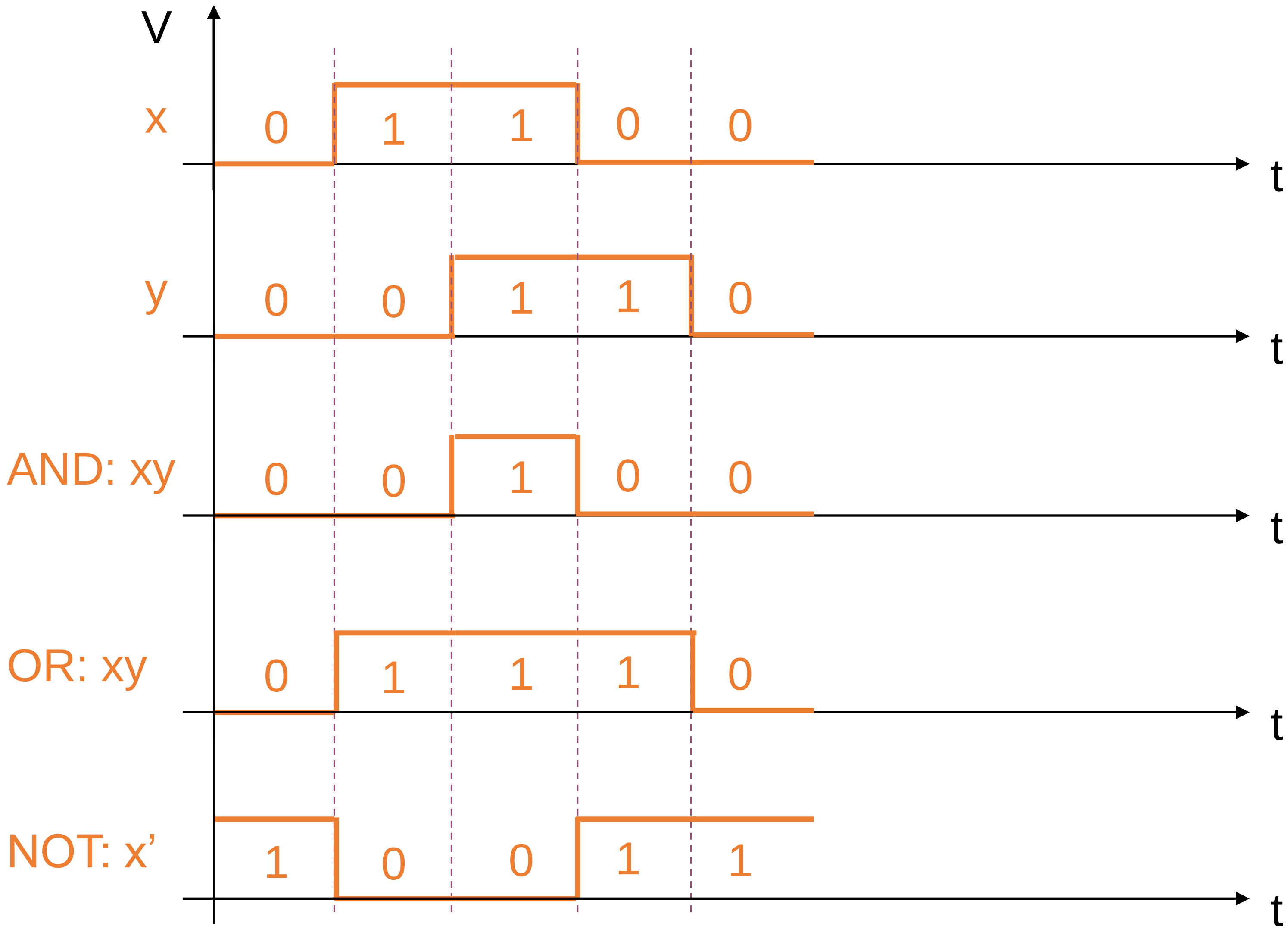


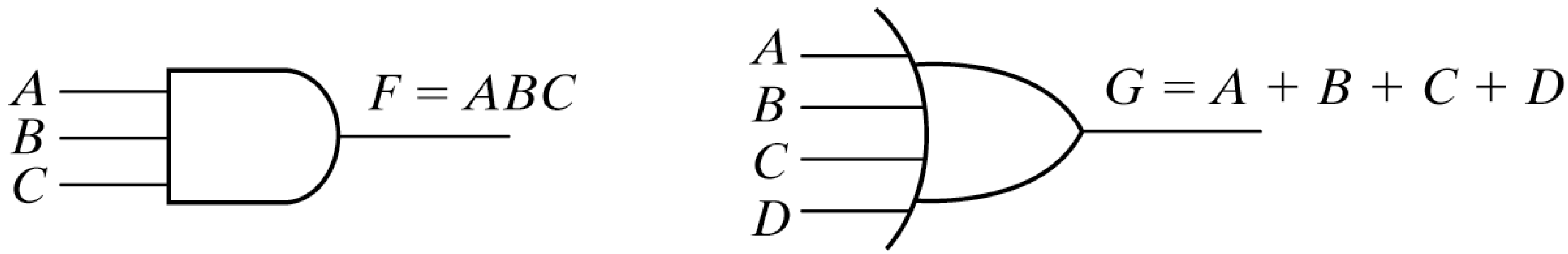
Fig. 1-3 Example of binary signals

Gates Operating on Signals



Input-Output Signals for gates

Gates with More Than Two Inputs



(a) Three-input AND gate

(b) Four-input OR gate

Fig. 1-6 Gates with multiple inputs