DI03000011: Digital Electronics Fundamentals

GTU Course Presentation

July 25, 2026

Course Contents I

Course Contents II

Course Contents III

Course Contents IV

Course Contents V

Analog vs. Digital Signals I

Physical quantities are inherently continuous, but digital systems operate on quantized, discrete values to achieve high reliability and design simplicity.

Analog vs. Digital Signals II

Key Points

- Analog signals vary continuously in time and range, representing infinite states (e.g., sound waves, temperature, voltage).
- Digital signals are restricted to discrete, quantized levels, most commonly binary states (0 and 1) mapped to specific voltage thresholds.
- The discretization process introduces quantization noise, but yields immense improvements in signal transmission fidelity.
- Modern systems convert analog inputs to digital via ADCs, process them, and restore them using DACs.

Advantages of Digital Systems I

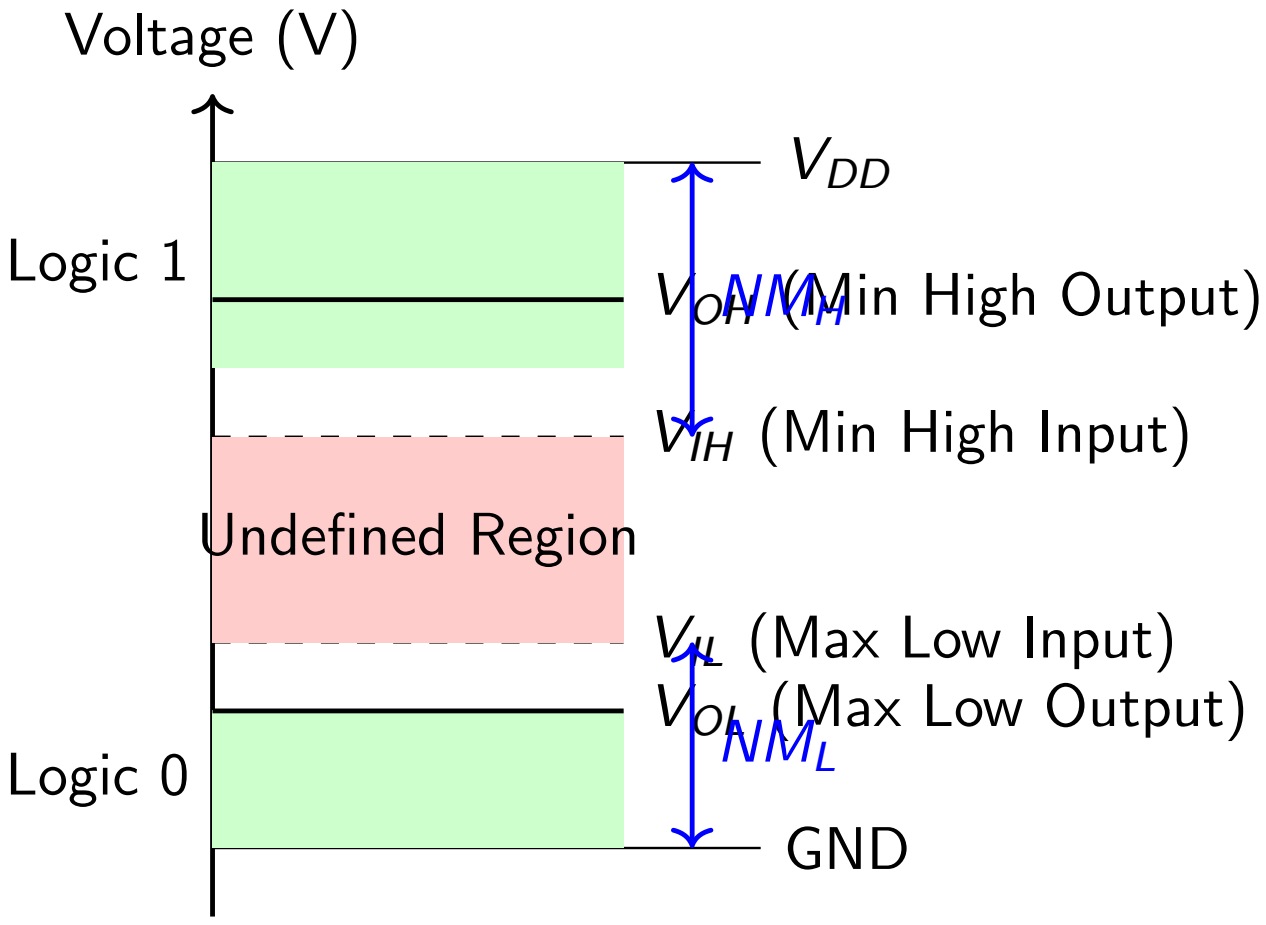
Digital electronic systems have almost completely replaced analog systems in computational and storage applications due to several engineering advantages.

Advantages of Digital Systems II

Key Points

- Superior Noise Immunity: Digital circuits use noise margins, allowing signals to degrade significantly before a state flip occurs.
- Reliable Information Storage: Binary states are easily stored in static/dynamic latching circuits (flip-flops, DRAM) or non-volatile cells (flash).
- Programmability & Reconfigurability: System behaviors can be altered through software instruction sets or hardware descriptions (VHDL/Verilog) rather than physical component changes.
- High-Density Integration: Transistors act as simple binary switches, allowing billions of gates to be fabricated on a single integrated circuit (VLSI).

Logic Levels and Noise Margins I



Logic Levels and Noise Margins II

Key Points

- Logic values are mapped to voltage ranges defined by four critical thresholds: V_OH, V_OL, V_IH, and V_IL.
- Noise Margin Low ($NM_L = V_IL$ - V_OL) represents the system's tolerance to additive ground/signal noise for a low state.
- Noise Margin High ($NM_H = V_OH$ - V_IH) represents the tolerance to subtractive voltage noise for a high state.
- Voltages entering the Undefined Region (V_IL to V_IH) can result in logic meta-stability and unpredictable circuit behavior.

Positional Number Systems I

Digital hardware relies on mathematical representations of value using base-specific positional notation.

Positional Number Systems II

Key Points

- Positional Notation Formula: A number in radix r is defined by the sum of its digits multiplied by corresponding powers of r.
- Decimal (Base 10): Standard human representation utilizing ten distinct digits (0-9).
- Binary (Base 2): Hardware-native representation utilizing two digits (0 and 1) corresponding directly to off/on transistor states.
- Octal (Base 8) & Hexadecimal (Base 16): Shorthand representations for long binary strings. Hexadecimal groups bits into sets of 4 (nibbles) for readability.

Radix Conversions I

Bases can be systematically converted using polynomial expansion (to base-10) or repeated division/multiplication (from base-10).

Radix Conversions II

Key Points

- Binary-to-Decimal: Compute the sum of weighted binary coefficients, e.g., $1101.1_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 13.5_{10}$.
- Decimal-to-Binary (Integers): Successive division by 2; the remainders collected from last to first yield the binary representation from MSB to LSB.
- Decimal-to-Binary (Fractions): Successive multiplication by 2; the integer parts extracted sequentially yield the fractional bits from MSB to LSB.
- Hexadecimal-to-Binary: Directly replace each hex digit with its corresponding 4-bit binary code (e.g., $E7_{16} = 1110\ 0111_2$).

Binary Arithmetic I

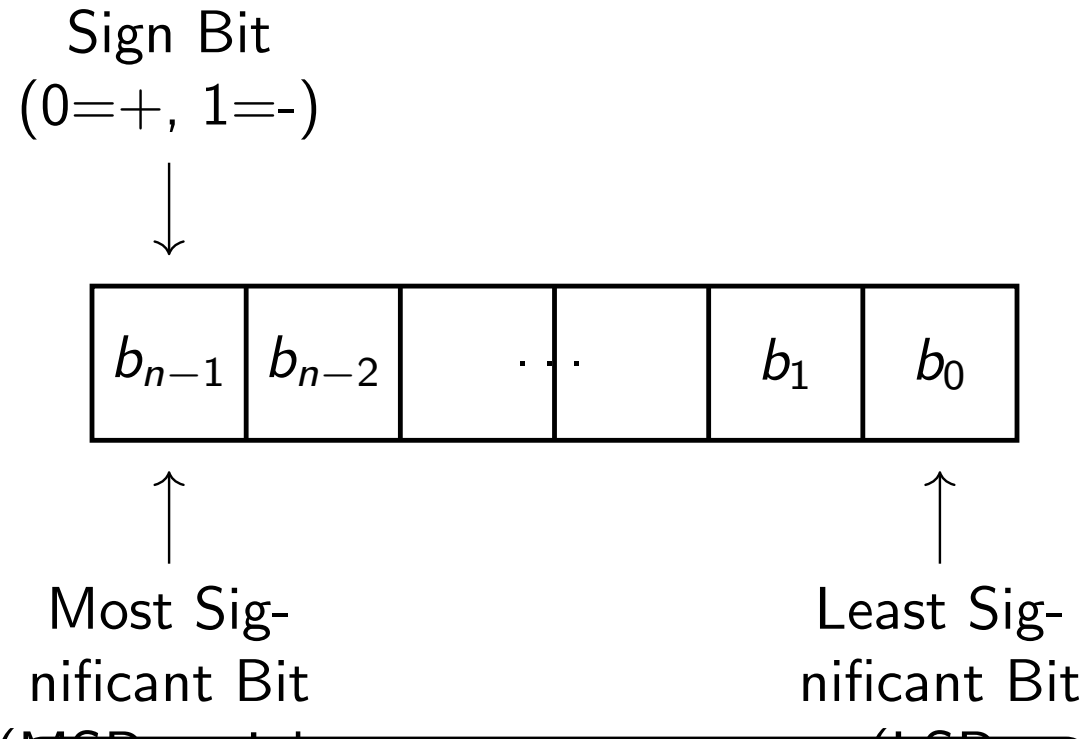
Binary arithmetic operations are performed using identical logic to decimal arithmetic, governed by carry-propagation rules.

Binary Arithmetic II

Key Points

- Binary Addition Rules: $0+0=0$, $0+1=1$, $1+0=1$, and $1+1=0$ with a carry-out of 1 to the next column.
- Binary Subtraction Rules: $0-0=0$, $1-1=0$, $1-0=1$, and $0-1=1$ with a borrow-in of 1 from the next higher-order column.
- Hardware Constraints: Standard binary subtraction logic requires borrow-routing circuits, which add propagation delay and component count.
- Optimized arithmetic processors circumvent subtraction subtraction-circuits entirely by employing signed representations.

Signed Number Representation I



2's Complement Weighting:
$A = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

Signed Number Representation II

Key Points

- Signed Magnitude: Standard sign representation but introduces a dual representation of zero (+0 and -0), complicating CPU logic.
- 1's Complement: Formed by inverting all bits of a positive binary number; still suffers from the dual zero representation problem.
- 2's Complement: Formed by adding 1 to the 1's complement representation. Only contains a single representation of zero (0000...).
- Sign Extension: To represent an n-bit signed integer in m bits (where m ¿ n), the sign bit (MSB) must be replicated into the new higher-order positions.

Binary Codes I

Binary codes assign structured bit patterns to represent decimal digits, characters, or state transitions under specific constraints.

Binary Codes II

Key Points

- Binary Coded Decimal (BCD): Encodes each decimal digit separately into 4 binary bits (e.g., $35_{10} = 0011\ 0101_{BCD}$). Simplifies numeric displays.
- Gray Code: An unweighted cyclic binary code where adjacent numbers differ by exactly one bit. Extremely useful for reducing errors in physical encoders.
- ASCII Code: Standardized 7-bit patterns representing text characters, numbers, punctuation, and system control signals.
- Error Detection Codes: Parity bits (even/odd check) detect single-bit errors in transmission by forcing the total count of 1-bits to match a rule.

Summary and Next Steps I

We have established the connection between physical logic levels and the binary number system which powers modern computer architecture.

Summary and Next Steps II

Key Points

- Physical voltage thresholds map continuous electricity to discrete Boolean states (0 and 1).
- Positional binary notation enables numerical processing, while 2's complement facilitates simplified subtraction hardware.
- Binary coding formats (BCD, Gray, ASCII) bridge hardware states with human-usable numeric and symbolic interfaces.
- Preview of Lecture 2: Boolean Algebra, standard Boolean theorems, logic gate primitives, and sum-of-products minimization.

Positional Weighted Number Systems I

A number system is defined by its base (radix) r. Any real number N can be represented as: $N = \sum_{i=-m}^{n-1} d_i \cdot r^i$, where d_i are the digits in the range $0 \leq d_i < r$.

Positional Weighted Number Systems II

Key Points

- The radix or base (r) determines the total number of unique symbols used in the system.
- The position of each digit corresponds to a weight equal to a power of the base (r^i).
- The radix point (decimal point, binary point, etc.) separates the integer part from the fractional part.
- For base 10 (Decimal): digits are 0-9. For base 2 (Binary): digits are 0-1. For base 8 (Octal): digits are 0-7.

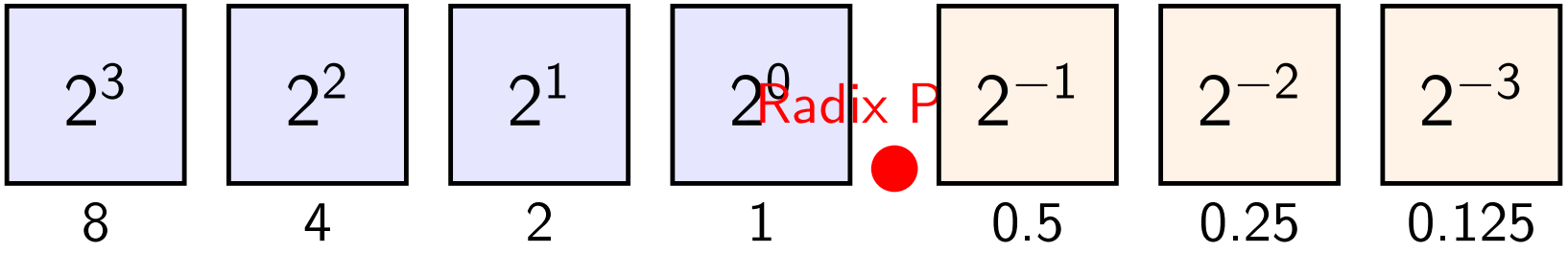
The Binary System (Base 2) I

Binary digits (bits) have two possible states: 0 (Low/Off) and 1 (High/On). In digital hardware, these are mapped to physical voltage levels (e.g., 0V and 5V).

Key Points

- Binary representation of a number: $(1101.101)_2$.
- Expansion:
  $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$.
- Decimal equivalent:
  $8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 = (13.625)_{10}$.
- Bits are the fundamental building blocks of digital logic circuits and computer memory.

Binary Weight Visualization I



Binary Weighted Positions (Base 2)

Key Points

- Positions to the left of the radix point have positive power-of-two weights $(1, 2, 4, 8, \dots)$.
- Positions to the right of the radix point have negative power-of-two weights $(0.5, 0.25, 0.125, \dots)$.
- The radix point acts as the anchor separating integer components from fractional components.

The Octal System (Base 8) I

The Octal system uses radix $r = 8$ with digits $0, 1, 2, 3, 4, 5, 6, 7$. It is widely used in computing as a compact shorthand for binary numbers, as one octal digit represents exactly three binary digits ($8 = 2^3$).

Key Points

- Octal representation: $(357.2)_8$.
- Expansion: $3 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 + 2 \cdot 8^{-1}$.
- Decimal equivalent: $3 \cdot 64 + 5 \cdot 8 + 7 \cdot 1 + 2 \cdot 0.125 = 192 + 40 + 7 + 0.25 = (239.25)_{10}$.
- Highly useful in operating systems (e.g., file permissions in UNIX/Linux).

Decimal to Binary Conversion Methods I

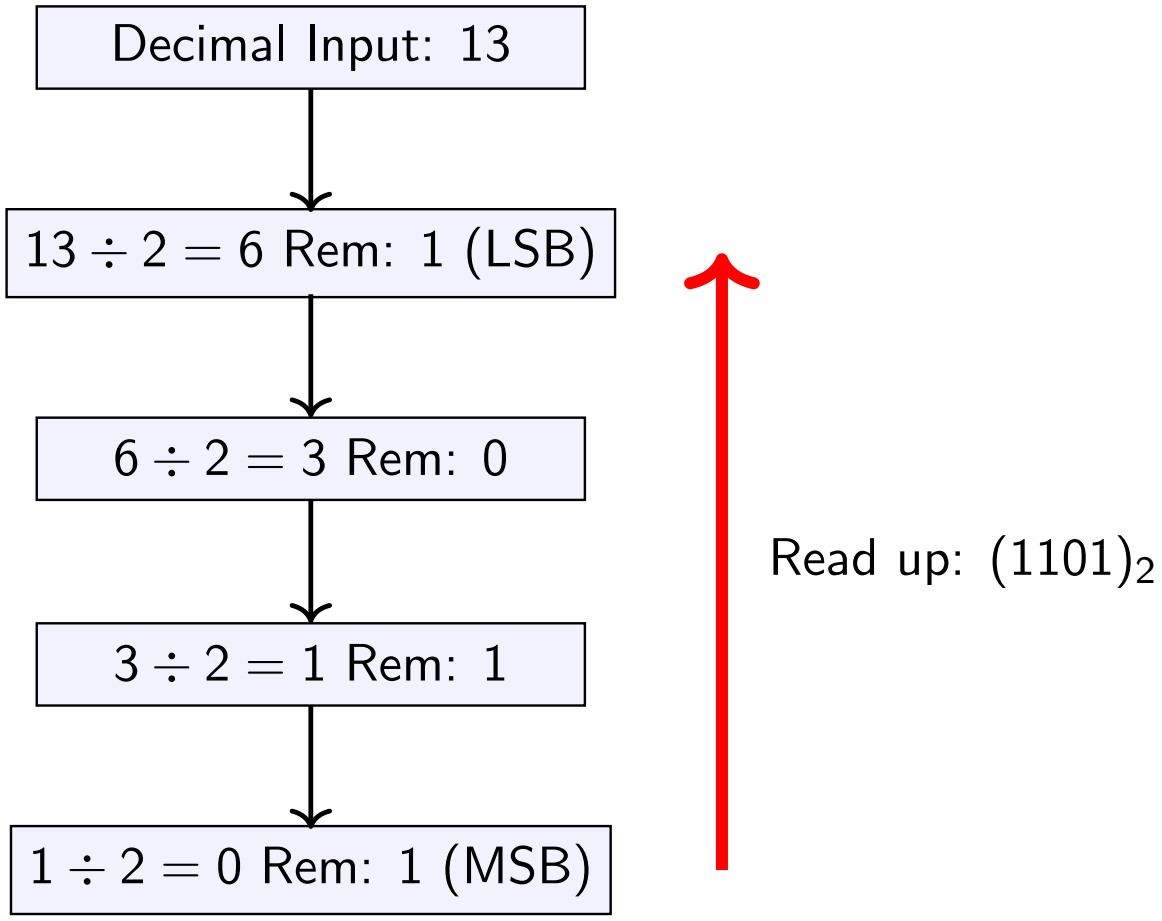
To convert decimal integers to binary, use successive division by 2 and record the remainders (LSB to MSB). For fractional parts, use successive multiplication by 2 and record the integer parts (MSB to LSB).

Decimal to Binary Conversion Methods II

Key Points

- Successive Division: Divide the decimal number by 2, write down the remainder. Continue until the quotient is 0.
- Remainders read from the bottom up give the binary integer representation (MSB at the bottom, LSB at the top).
- Successive Multiplication: Multiply the fractional part by 2. The integer part of the product is the binary digit.
- Repeat the multiplication on the remaining fractional part. This process may terminate or recur infinitely.

Decimal to Binary Division Diagram I



Decimal to Binary Division Diagram II

Key Points

- Illustrates the conversion of decimal 13_{10} to binary 1101_2.
- First remainder is the Least Significant Bit (LSB) because it represents the coefficient of 2^0.
- Last remainder is the Most Significant Bit (MSB) because it corresponds to the highest power of 2.

Binary to Octal & Octal to Binary Shortcut I

Since $8 = 2^3$, grouping binary digits in sets of three allows direct conversion between Binary and Octal. Grouping starts at the radix point and moves outward (left for integer, right for fraction).

Binary to Octal & Octal to Binary Shortcut II

Key Points

- To convert Binary to Octal: Group bits in threes. $(110101.011)_2 \rightarrow (110)(101).(011)_2 = (65.3)_8$.
- If necessary, pad with leading zeros for the integer part or trailing zeros for the fractional part.
- To convert Octal to Binary: Expand each octal digit into its 3-bit binary equivalent.
- Example: $(47.1)_8 \rightarrow (100)(111).(001)_2 = (100111.001)_2$.

Octal and Decimal Conversions I

Octal to Decimal is done using polynomial expansion: $(N)_{10} = \sum d_i \cdot 8^i$. Decimal to Octal uses successive division/multiplication by 8.

Octal and Decimal Conversions II

Key Points

- Octal to Decimal: $(25.4)_8 = 2 \cdot 8^1 + 5 \cdot 8^0 + 4 \cdot 8^{-1} = 16 + 5 + 0.5 = (21.5)_{10}$.
- Decimal to Octal (Integer): Successive division by 8. E.g., $150 \div 8 = 18$ rem 6; $18 \div 8 = 2$ rem 2; $2 \div 8 = 0$ rem 2. Thus, $150_{10} = 226_8$.
- Decimal to Octal (Fraction): Successive multiplication by 8. E.g., $0.3125 \times 8 = 2.5$ (int 2); $0.5 \times 8 = 4.0$ (int 4). Thus, $0.3125_{10} = 0.24_8$.
- These direct algorithms form the baseline for manual conversions before using binary shortcuts.

Lecture Summary and Exercises I

Summary: Positional number systems allow representation of numbers in any base r. Binary ($r = 2$) and Octal ($r = 8$) are key representations in digital engineering due to powers of 2.

Key Points

- Key Takeaway: The base r determines the number of digits and the positional weight.
- Grouping binary digits in sets of 3 facilitates direct conversion to/from Octal.
- Practice Problem 1: Convert $(75.375)_{10}$ to Binary and Octal.
- Practice Problem 2: Convert $(101111.101)_2$ to Octal and Decimal.

The Decimal Number System (Base-10) I

Decimal representation is the standard base-10 mathematical notation used globally, characterized by ten symbols and weighted positions.

The Decimal Number System (Base-10) II

Key Points

- Defined by radix or base r = 10, using the digit set 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Positional weight is defined by 10^i, where i is the position index relative to the radix point.
- Value evaluation: The value of a decimal number is the sum of the digits multiplied by their positional weight (e.g., $25.4 = 2*10^1 + 5*10^0 + 4*10^-1$).
- Though intuitive for humans, decimal representation is poorly suited for hardware since representing 10 levels requires high precision and noise-immune circuitry.

The Hexadecimal Number System (Base-16) I

Hexadecimal is a base-16 positional system widely used in computing as a compact, human-readable shorthand for raw binary values.

The Hexadecimal Number System (Base-16) II

Key Points

- Defined by radix r = 16, using the standard alphanumeric set 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
- The alphabetic characters represent values 10 through 15: A=10, B=11, C=12, D=13, E=14, F=15.
- Positional weights are powers of 16 ($16^0 = 1$, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, etc.).
- Hexadecimal is extremely efficient for representing bytes; a single byte (8 bits) can be written as exactly two hexadecimal digits (e.g., $11111111_2 = FF_16$).

Decimal-to-Hexadecimal Conversion I

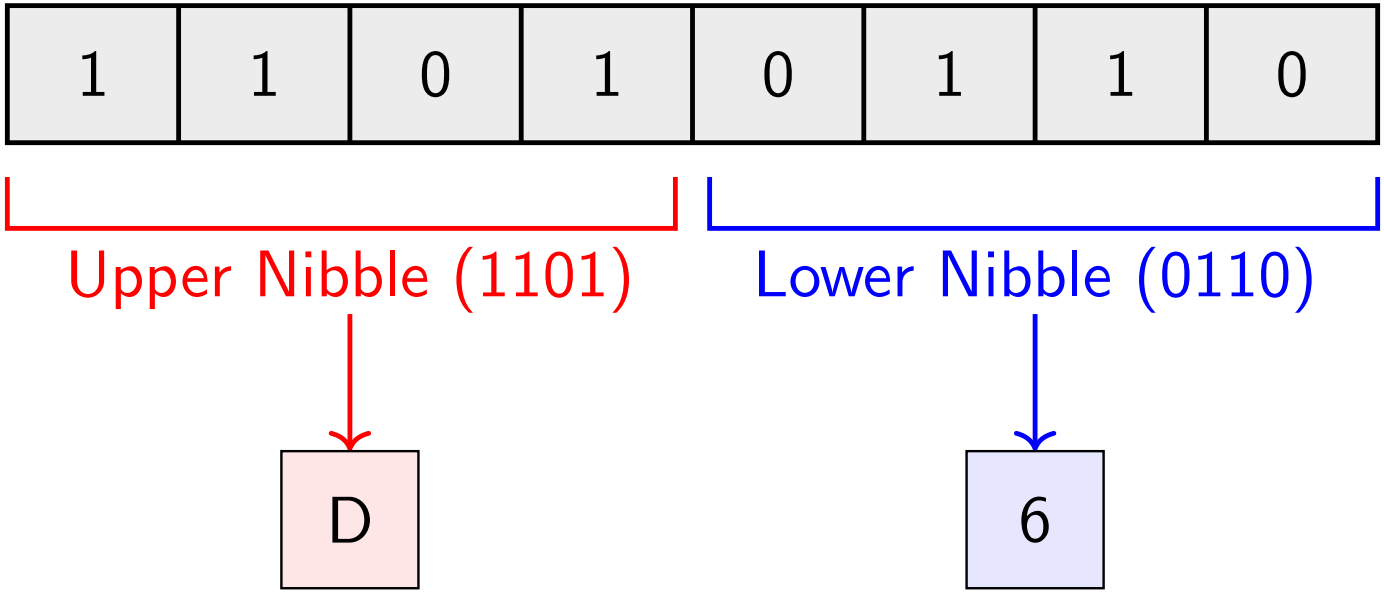
Converting decimal numbers to hexadecimal relies on the successive division-by-16 method for integers and multiplication-by-16 for fractions.

Decimal-to-Hexadecimal Conversion II

Key Points

- Integer conversion: Divide the decimal integer by 16, record the remainder, and repeat with the quotient until it reaches 0. The remainders form the hex number from LSD (first) to MSD (last).
- Fractional conversion: Multiply the fraction by 16, record the integer part as the digit, and repeat with the remaining fraction. This progresses from MSD to LSD.
- Example: Convert decimal 423 to hex. 423 / 16 = 26 remainder 7 (LSD); 26 / 16 = 1 remainder 10 (A); 1 / 16 = 0 remainder 1 (MSD). Result: $1A7_16$.
- Hexadecimal-to-Decimal: Compute the sum of the digits multiplied by their positional weight (e.g., $A3_16 = 10*16^1 + 3*16^0 = 163_10$).

Binary-to-Hexadecimal Nibble Mapping I



Hexadecimal Result: D6

Binary-to-Hexadecimal Nibble Mapping II

Key Points

- Hexadecimal serves as a direct mapping to binary since $16 = 2^4$.
- To convert, binary bits are grouped into 4-bit sets (nibbles) starting from the binary point moving outward.
- Upper Nibble: 1101_2 converts to 13_10, which corresponds to symbol 'D' in Hexadecimal.
- Lower Nibble: 0110_2 converts to 6_10, which corresponds to symbol '6' in Hexadecimal.

Binary Addition: Fundamental Rules I

Binary addition follows rules analogous to decimal addition but is simplified by having only two digits (0 and 1) and a base of 2.

Key Points

- Rule 1: $0 + 0 = 0$ (Carry $= 0$)
- Rule 2: $0 + 1 = 1$ (Carry $= 0$) and $1 + 0 = 1$ (Carry $= 0$)
- Rule 3: $1 + 1 = 0$ (Carry $= 1$) - equivalent to decimal 2 (10_2)
- Rule 4: $1 + 1 + 1 = 1$ (Carry $= 1$) - occurs when adding two 1s with an incoming carry of 1, equivalent to decimal 3 (11_2).

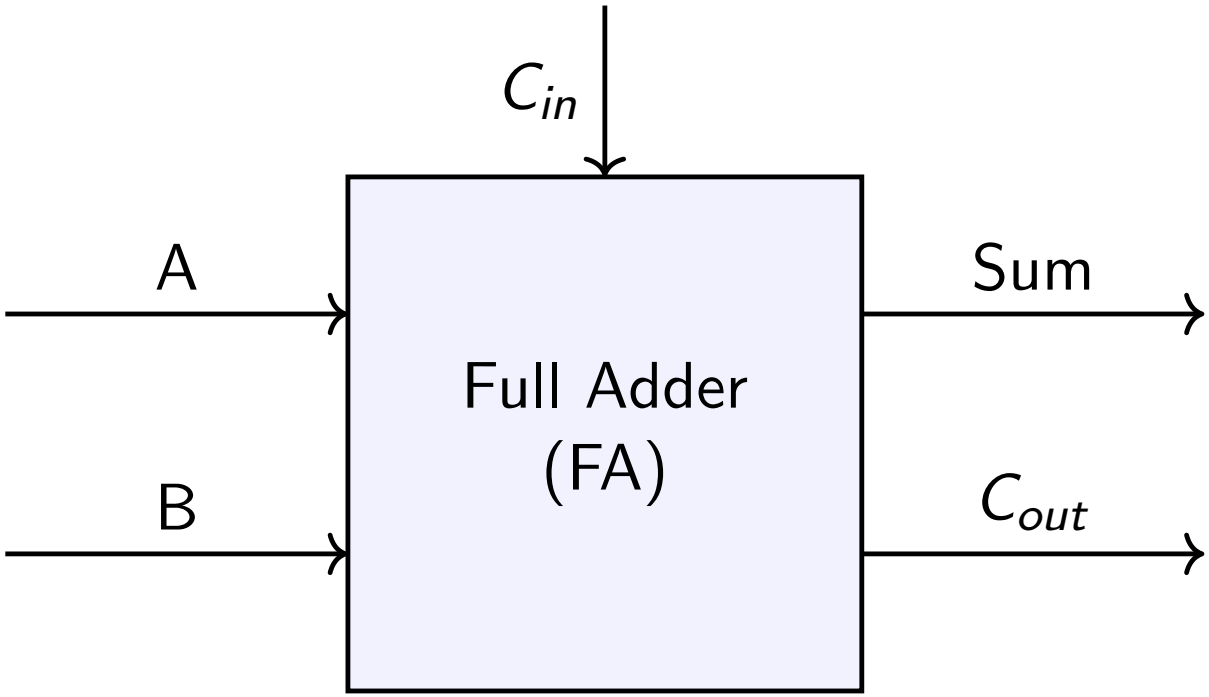
Multi-Bit Binary Addition and Carries I

When adding multi-bit numbers, addition proceeds from the Least Significant Bit (LSB) to the Most Significant Bit (MSB), propagating carry bits to the left.

Key Points

- Align the bits of the two operands vertically by their binary points.
- Add columns from right to left, including any carry-in (C_in) from the previous column.
- Compute the sum (S) and carry-out (C_out) for each bit position.
- The final carry-out from the MSB indicates an overflow condition if the result exceeds the system word size.

Full Adder Gate Logic Block I



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + C_{in}(A \oplus B)$$

Full Adder Gate Logic Block II

Key Points

- A Full Adder is a combinational circuit that performs the arithmetic addition of three input bits.
- It consists of three inputs: A (operand bit), B (operand bit), and C_in (carry bit from the previous less-significant stage).
- It produces two outputs: Sum (S) and Carry-out (C_out).
- The boolean expression for Sum is an XOR operation: S = A ^ B ^ C_in.
- The Carry-out expression is defined as C_out = A*B + $C_in * (A \hat{} B)$.

Half Adder vs. Full Adder Comparison I

Implementing binary addition in hardware requires cascading adder units, where the choice between Half Adders and Full Adders depends on the bit position.

Half Adder vs. Full Adder Comparison II

Key Points

- Half Adder: Takes two single-bit inputs (A, B) and generates Sum (S) and Carry (C). It cannot handle an incoming carry bit.
- Full Adder: Takes three single-bit inputs (A, B, Carry-in) and generates Sum (S) and Carry-out (C_out).
- LSB Addition: A Half Adder can be used for the Least Significant Bit since there is no carry-in from a lower column.
- Cascading: To add n-bit numbers, we cascade one Half Adder (for LSB) and n-1 Full Adders, propagating carries sequentially.

Binary Addition Step-by-Step Examples I

Let's examine physical execution of binary addition including how carry bits are generated and propagated.

Binary Addition Step-by-Step Examples II

Key Points

- Example 1: Add 5 (0101_2) and 6 (0110_2). Column addition: LSB 1+0=1 (C=0). Second 0+1=1 (C=0). Third 1+1=0 (C=1). MSB 0+0+1=1 (C=0). Result: $1011_2 = 11_10$.
- Example 2: Add 7 (0111_2) and 3 (0011_2). LSB 1+1=0 (C=1). Second 1+1+1=1 (C=1). Third 1+0+1=0 (C=1). MSB 0+0+1=1 (C=0). Result: $1010_2 = 10_10$.
- Notice how the propagation of carries can create a propagation delay in physical circuits (Ripple Carry Adder).
- Advanced processors use Carry Lookahead Adders (CLA) to compute carry signals in parallel, overcoming this propagation delay.

Introduction to Complement Arithmetic I

In digital systems, hardware subtraction is inefficient. By using complements, subtraction is converted into addition. This allows the arithmetic logic unit (ALU) to use a binary adder for both addition and subtraction, simplifying the circuit design and minimizing gate propagation delay.

Introduction to Complement Arithmetic II

Key Points

- Complements are used in digital computers to simplify the subtraction operation and for logical manipulation.
- There are two types of complements for each base r system: the radix complement (r's complement) and the diminished radix complement ((r-1)'s complement).
- For binary numbers (base 2), these correspond to the 2's complement and the 1's complement.
- Using complements reduces hardware cost because separate subtractor circuits are not required.

1's Complement Representation I

The 1's complement of a binary number is the diminished radix complement for base 2. It is mathematically defined and easily implemented in hardware using NOT gates.

1's Complement Representation II

Key Points

- For an n-digit binary number N, its 1's complement is defined as $(2^n - 1) - N$.
- Operationally, the 1's complement is obtained by inverting all bits of the binary number (0s become 1s, and 1s become 0s).
- For example, the 1's complement of 1011001 is 0100110.
- The representation of signed numbers in 1's complement has the disadvantage of having two representations for zero: +0 (00000000) and -0 (11111111).

1's Complement Subtraction: Case M ¿= N I

When subtracting a smaller or equal binary number N from a larger binary number M (M - N), we add M to the 1's complement of N. An end carry is generated, which is added to the least significant bit (LSB). This is called the end-around carry.

1's Complement Subtraction: Case M ¿= N II

Key Points

- Step 1: Obtain the 1's complement of the subtrahend N.
- Step 2: Add this complement to the minuend M.
- Step 3: If an end carry of 1 is generated from the MSB, remove it and add it to the LSB of the result.
- Example: 1110 (14) - 1001 (9) in 4-bit binary. 1's complement of 1001 is 0110. Add: $1110 + 0110 = 10100$. Carry 1 is added to LSB: $0100 + 1 = 0101$ (+5).

1's Complement Subtraction: Case M ¡ N I

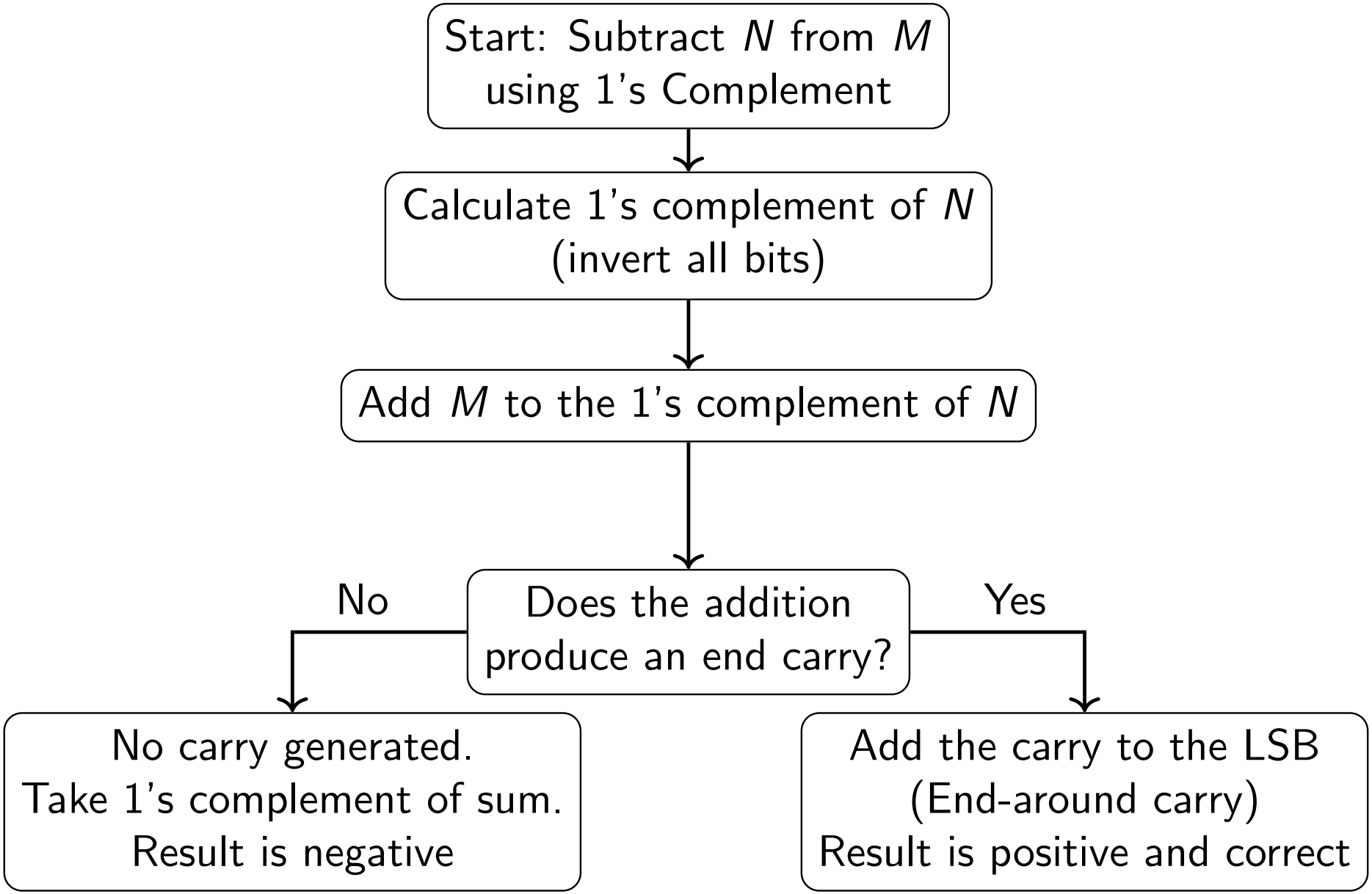
When subtracting a larger binary number N from a smaller binary number M (M - N), we add M to the 1's complement of N. No end carry is generated, indicating that the result is negative and is in 1's complement form.

1's Complement Subtraction: Case M ¡ N II

Key Points

- Step 1: Obtain the 1's complement of the subtrahend N.
- Step 2: Add this complement to the minuend M.
- Step 3: If no end carry is generated, the result is negative.
- Step 4: Take the 1's complement of the sum to find the magnitude, and append a negative sign.
- Example: 1001 (9) - 1110 (14). 1's complement of 1110 is 0001. Add: $1001 + 0001 = 1010$. No carry. 1's complement of 1010 is 0101, so the result is -0101 (-5).

Algorithm Flow for 1's Complement Subtraction I



Algorithm Flow for 1's Complement Subtraction II

Key Points

- The flow begins with bitwise inversion of the subtrahend N.
- The decision block evaluates the carry-out (C_out) of the binary adder.
- The yes path implements the end-around carry loop, which requires an additional incrementer cycle in hardware.
- The no path shows that the final answer is negative and must be recomplemented to be read as a magnitude.

2's Complement Representation I

The 2's complement of a binary number is the radix complement for base 2. It is the dominant representation of signed numbers in modern processors because it simplifies hardware arithmetic circuits.

2's Complement Representation II

Key Points

- For an n-digit binary number N, its 2's complement is defined as 2^n - N.
- Operationally, the 2's complement is obtained by adding 1 to the 1's complement of the number.
- Alternatively, copy all bits from right to left up to and including the first 1, then invert all remaining bits.
- Example: The 2's complement of 1011000 is 0101000.
- It provides a unique representation for zero (00000000) and allows the subtraction operation to be performed directly without end-around carry.

2's Complement Subtraction: Case M ¿= N I

To subtract N from M where M ¿= N using 2's complement, add M to the 2's complement of N. The subtraction results in an end carry of 1, which is discarded. The remaining bits form the correct positive difference.

2's Complement Subtraction: Case M ¿= N II

Key Points

- Step 1: Obtain the 2's complement of the subtrahend N.
- Step 2: Add the 2's complement of N to the minuend M.
- Step 3: Discard the end carry ($C_out = 1$). The remaining sum is the positive result.
- Example: 1110 (14) - 1001 (9). 2's complement of 1001 is 0110 + 1 = 0111. Add: 1110 + 0111 = 10101. Discard the end carry 1: result is 0101 (+5).

2's Complement Subtraction: Case M ¡ N I

To subtract N from M where M ¡ N using 2's complement, add M to the 2's complement of N. No carry is generated, indicating that the result is negative and is in its 2's complement form.

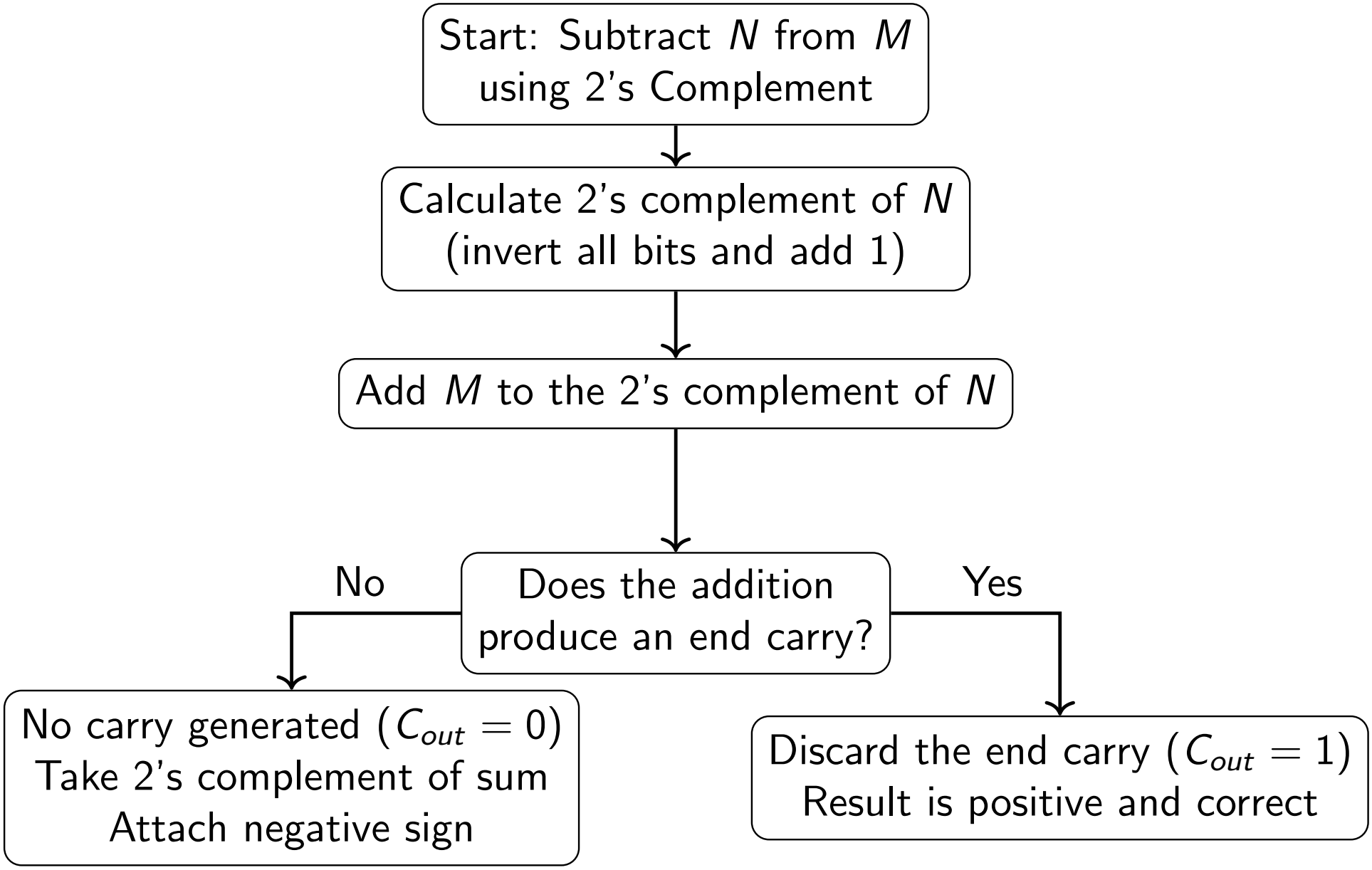
2's Complement Subtraction: Case M ¡ N II

Key Points

- Step 1: Obtain the 2's complement of the subtrahend N.
- Step 2: Add this complement to the minuend M.
- Step 3: If no end carry is generated ($C_out = 0$), the result is negative.
- Step 4: Take the 2's complement of the sum to find the magnitude, and append a negative sign.
- Example: 1001 (9) - 1110 (14). 2's complement of 1110 is 0010. Add: 1001 + 0010 = 1011. No carry. 2's complement of 1011 is 0101, so the result is -0101 (-5).

Algorithm Flow for 2's Complement Subtraction I

Algorithm Flow for 2's Complement Subtraction II



Algorithm Flow for 2's Complement Subtraction III

Key Points

- The flowchart shows the operational flow for subtracting numbers using 2's complement arithmetic.
- Discarding the carry bit simplifies hardware implementation because no end-around carry loop is required.
- If carry is 0, the sum is in 2's complement form and must be converted to magnitude form for human-readable output.

Positional Number Systems & Radix Representation I

A positional number system represents numbers using a base (radix) r. The value of a digit is determined by its position relative to the radix point.

Positional Number Systems & Radix Representation II

Key Points

- General representation of a number: N = $(a_{n-1}\ a_{n-2}\ \ldots\ a_0\ .\ a_{-1}\ a_{-2}\ \ldots\ a_{-m})\_r$
- Positional weights: Each position i carries a weight of $\hat{r}i$.
- Polynomial expansion: N = $sum\_i = -m^{n-1}\ a_i\ *\ r^i$
- For binary (r=2), digits are 0, 1. For decimal (r=10), digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Decimal to Binary: Successive Division-by-2 I

To convert an integer from decimal (r=10) to binary (r=2), we repeatedly divide the decimal number by 2 and record the remainders.

Decimal to Binary: Successive Division-by-2 II

Key Points

- Step 1: Divide the decimal number by 2. Record the quotient and the remainder (0 or 1).
- Step 2: Divide the quotient by 2. Record the new quotient and remainder.
- Step 3: Repeat until the quotient becomes 0.
- The first remainder generated is the Least Significant Bit (LSB).
- The final remainder generated is the Most Significant Bit (MSB). The binary number is read from MSB to LSB.

Diagram: Decimal to Binary Conversion of 13 I

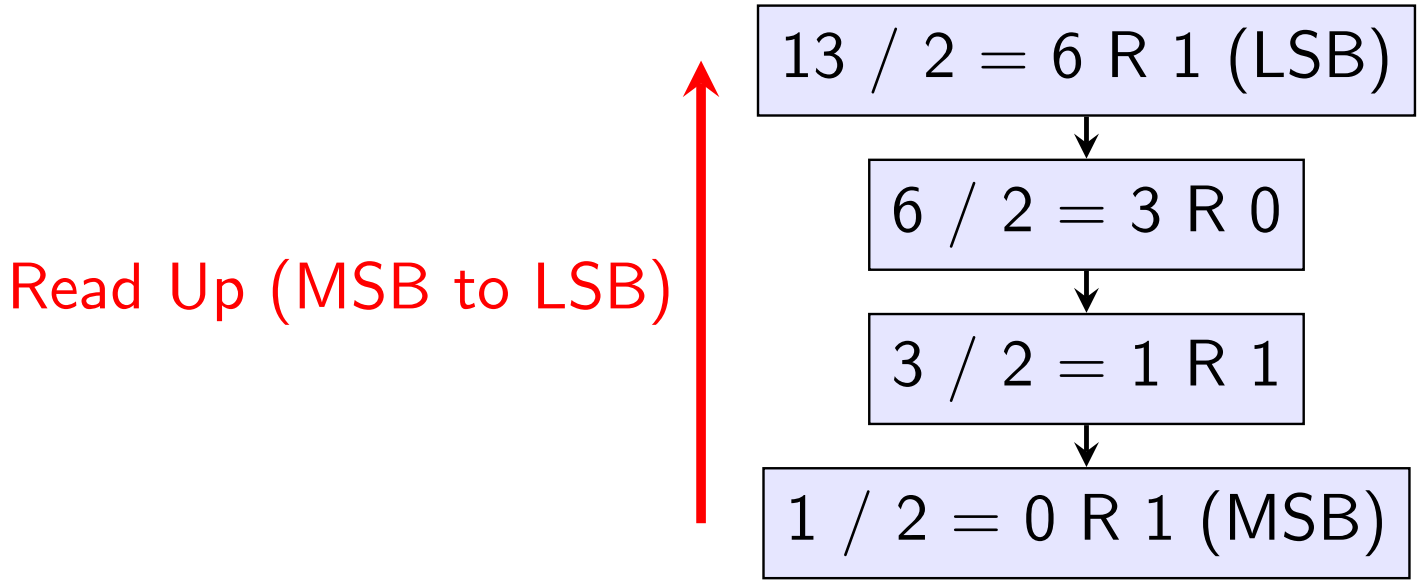


Diagram: Decimal to Binary Conversion of 13 II

Key Points

- This flowchart shows the successive division-by-2 method for converting the decimal integer 13 to binary.
- Result: $13_{10} = 1101_2$.
- The remainder path forms the binary representation in reverse order of generation.

Decimal Fractions to Binary: Successive Multiplication I

To convert a decimal fraction to binary, multiply the fractional part by 2. The integer part of the product is the binary digit, and the new fractional part is multiplied again.

Decimal Fractions to Binary: Successive Multiplication II

Key Points

- Step 1: Multiply the fractional part by 2.
- Step 2: Record the integer part (0 or 1) as the binary digit.
- Step 3: Take the remaining fractional part and repeat the multiplication.
- Step 4: Continue until the fraction becomes 0, or the desired level of precision is achieved.
- The first integer digit generated is the MSB (placed directly after the binary point).

Binary to Decimal Conversion I

Binary numbers are converted to decimal by summing the products of each binary digit and its corresponding power-of-2 weight.

Key Points

- General formula: $N_{10} = sum_i\ (d_i * 2^i)$, where d_i is 0 or 1.
- Example: Convert 1101.11_2 to decimal.
- Integer part: $(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 8 + 4 + 0 + 1 = 13$.
- Fractional part: $(1 * 2^{-1}) + (1 * 2^{-2}) = 0.5 + 0.25 = 0.75$.
- Combined Result: $1101.11_2 = 13.75_{10}$.

Diagram: Binary Weights and Values I

1	1	0	1	1	1
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

Radix Point

Key Points

- Graphical representation of position values for the binary number 1101.11_2.
- The binary point separates the positive powers of 2 from negative powers of 2.
- Multiplying digits by their respective weights and summing yields the decimal equivalent.

Introduction to Number Complements I

Complements are used in digital systems to simplify subtraction operations and represent negative numbers. There are two types: Radix (r's) Complement and Diminished Radix ((r-1)'s) Complement.

Introduction to Number Complements II

Key Points

- For a given base r and an n-digit integer N:
- Diminished Radix Complement ((r-1)'s complement): $(r^n$ - 1) - N.
- Radix Complement (r's complement): r^n - N (for N != 0).
- In decimal (r=10), these are the 9's and 10's complements.
- In binary (r=2), these are the 1's and 2's complements.

Binary Complements: 1's and 2's Complements I

The 1's and 2's complements are fundamental in binary arithmetic for representing signed numbers and performing subtraction using adder circuits.

Binary Complements: 1's and 2's Complements II

Key Points

- 1's Complement: Obtained by inversion (bitwise NOT) of all bits. E.g., 1010_2 -¿ 0101_2.
- Formula for 1's complement of N: $(2^n - 1)$ - N. Since 2^n - 1 is a string of all 1s, subtraction simply flips the bits.
- 2's Complement: Obtained by adding 1 to the 1's complement. E.g., 1010_2 -¿ $0101_2 + 1 = 0110_2$.
- Formula for 2's complement of N: 2^n - N.
- Alternative 2's complement shortcut: Copy bits from right to left up to and including the first '1', then flip all remaining bits.

Lecture Summary & Engineering Applications I

Understanding binary-decimal conversion and complements is key to computer hardware design and CPU arithmetic logic unit (ALU) implementation.

Lecture Summary & Engineering Applications II

Key Points

- Base conversion ensures data compatibility between human-readable interfaces (decimal) and processor hardware (binary).
- The division-by-2 and multiplication-by-2 methods cover both integer and fractional components.
- 2's complement is the standard representation for signed integers in modern computing systems because it allows subtraction to be performed using addition hardware.
- There is no representation for '-0' in 2's complement, unlike 1's complement, which simplifies ALU control logic.

Introduction to Octal Numbering I

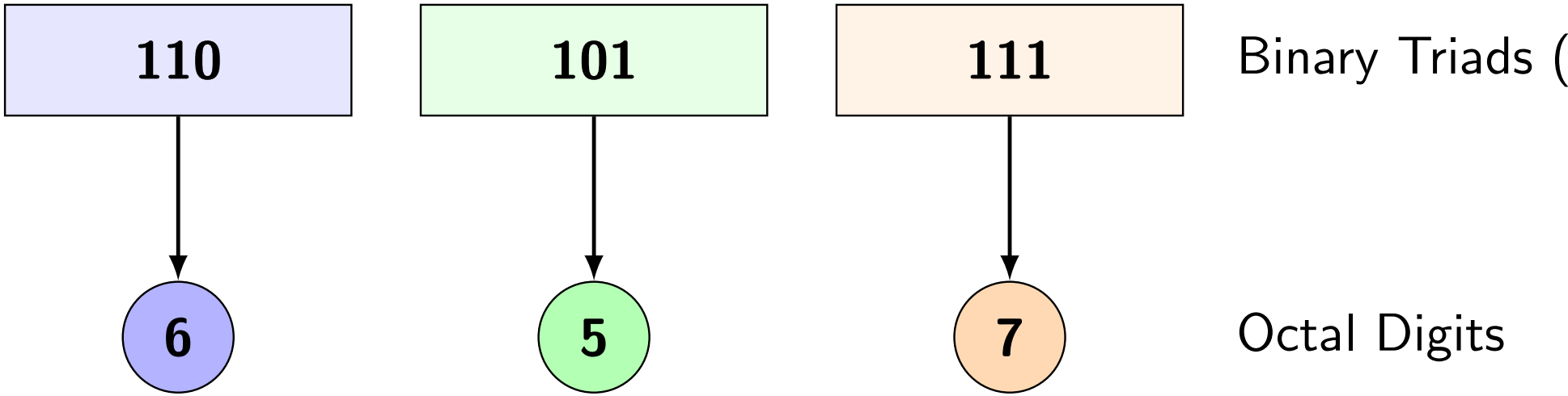
Understanding Radix-8 positional numeral representation.

Introduction to Octal Numbering II

Key Points

- Octal representation is based on radix r = 8, utilizing digits 0, 1, 2, 3, 4, 5, 6, and 7.
- A positional number is evaluated as $\sum_{i=-m}$^n-1d_i·8^i, *where*d_i*isthedigitatpositioni*.
- Because 8 is a power of 2 (2^3), there is a direct logarithmic mapping between binary and octal digits.
- Used historically in architectures with word lengths that are multiples of 3, such as the PDP-8 (12-bit) and DEC PDP-10 (36-bit).

Binary-to-Octal Conversion Diagram I



Binary-to-Octal Conversion Diagram II

Key Points

- Grouping Rule: Partition the binary bits into groups of three (triads) starting from the binary point, moving left for integers and right for fractions.
- Padding Rule: Append leading zeros to the leftmost integer group, and trailing zeros to the rightmost fractional group if they contain fewer than three bits.
- Direct Substitution: Each triad is mapped directly to its octal equivalent (e.g., 010_2 -¿ 2_8, 111_2 -¿ 7_8).

Decimal-to-Octal Conversion Methods I

Algorithmic translation of decimal integers and fractions to base-8.

Decimal-to-Octal Conversion Methods II

Key Points

- Integer Conversion (Double-Dabble for Radix-8): Successive division by 8. Divide the decimal integer by 8, record the remainder, and repeat with the quotient. The first remainder is the LSD, the last is the MSD.
- Fractional Conversion: Successive multiplication by 8. Multiply the fractional component by 8, extract the integer portion as the digit, and repeat on the remaining fraction. Read the digits top-down.
- Octal-to-Decimal: Convert by expanding the number in polynomial form using weights of 8^i. For example, 143_8 = 1·8^2+4·8^1+3·8^0= 64 + 32 + 3 =99_10.

The Hexadecimal (Base-16) System I

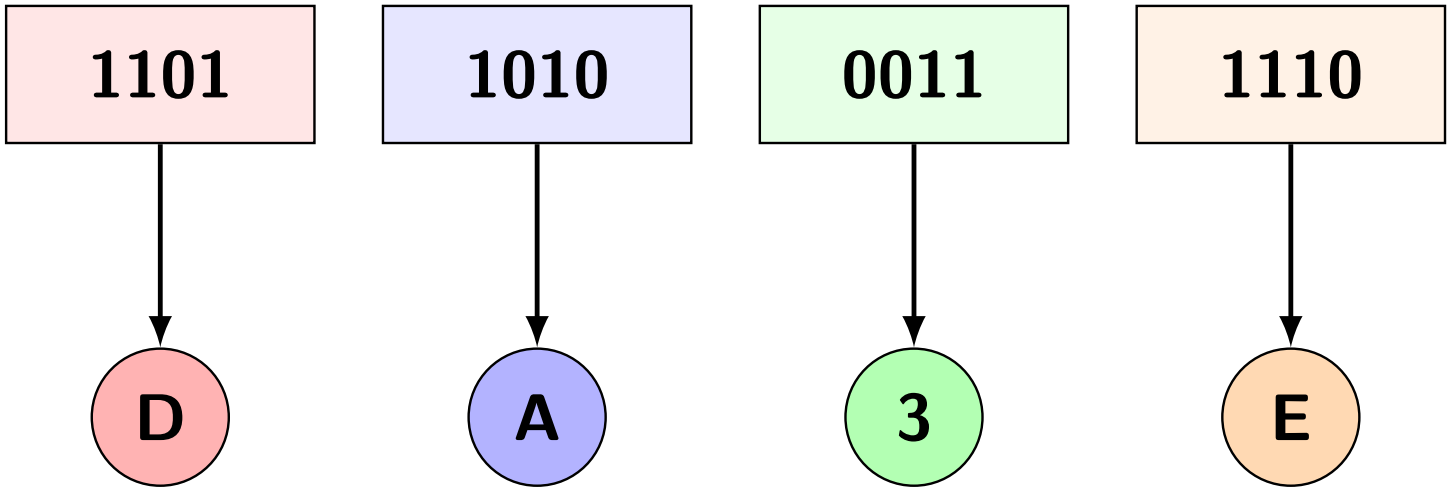
Why modern hardware engineering heavily favors radix-16 representation.

The Hexadecimal (Base-16) System II

Key Points

- Radix-16 representation utilizes 16 distinct symbols: 0-9 for values 0-9, and A-F for values 10-15.
- Since $16 = 2^4$, a single hexadecimal digit represents exactly 4 binary bits (a nibble).
- Two hexadecimal digits represent exactly one 8-bit byte (0x00 to 0xFF), which aligns perfectly with modern byte-addressable memory architectures.
- Hexadecimal is widely preferred over binary for software debugging and system status registers due to its density and direct binary mapping.

Binary-to-Hexadecimal Conversion Diagram I



Binary-to-Hexadecimal Conversion Diagram II

Key Points

- Nibble Grouping: Group binary bits into sets of four (nibbles) starting from the binary point.
- Zero-Padding: Insert leading zeros for the integer part, and trailing zeros for the fractional part to complete any incomplete 4-bit groups.
- Hex Mapping: Each nibble is translated into a single hexadecimal character (e.g., 1010_2 -¿ A_16, 1111_2 -¿ F_16).

Octal-to-Hexadecimal Conversion via Binary I

Bridge conversion methodology using radix-2 as an intermediate form.

Octal-to-Hexadecimal Conversion via Binary II

Key Points

- Indirect Conversion Path: To convert from octal to hexadecimal, it is computationally faster to convert to binary first, then to hexadecimal, rather than using decimal division.
- Step 1 (Octal to Binary): Expand each octal digit into its corresponding 3-bit binary equivalent.
- Step 2 (Binary to Hex): Regroup the continuous binary sequence into 4-bit nibbles starting from the radix point.
- Example: Convert 356_8 to Hex. $356\_8 = (011)(101)(110)\_2 = 011101110\_2$. Grouping: $(0000)(1110)(1110)\_2 = 0EE\_16 = EE\_16$.

System Applications of Octal and Hexadecimal I

Real-world engineering applications of non-decimal radix systems.

System Applications of Octal and Hexadecimal II

Key Points

- Unix Permission Masks: File access control lists use octal digits (e.g., chmod 755), where each digit represents Owner, Group, and Public permissions as a 3-bit mask of Read, Write, and Execute.
- Memory Address Space representation: Microprocessor addresses (e.g., x86 or ARM pointer structures) are written in hex (e.g., 0x7FFE0010) to reflect register boundaries.
- Instruction Encoding: Machine language opcodes are often expressed in hex to quickly identify byte-aligned operation fields.
- Network Protocols: IPv6 addresses are written as eight groups of four hexadecimal digits (e.g., 2001:db8:3c4d:15::1a2f) to represent 128-bit addresses.

Base-8 and Base-16 Arithmetic Principles I

Mathematical operations under non-decimal positional constraint.

Base-8 and Base-16 Arithmetic Principles II

Key Points

- Radix Addition: Perform standard column addition. If the sum of a column is equal to or greater than the radix r (8 or 16), subtract r from the sum, record the difference, and carry 1 to the next column.
- Hex Addition Example: $7F\_16 + A4\_16$. LSB: F + 4 = 15 + 4 = 19. 19 - 16 = 3 (carry 1). MSB: 7 + A + 1 = 7 + 10 + 1 = 18. 18 - 16 = 2 (carry 1). Result = 123_16.
- Radix Subtraction: When a column requires borrowing from the left, add the base value (8 or 16) to the current column digit before performing subtraction.
- Octal Subtraction Example: 52_8 - 37_8. LSB: 2 ¡ 7, borrow 1 from 5. New value = 2 + 8 = 10. 10 - 7 = 3. MSB: (5 - 1) - 3 = 1. Result = 13_8.

Lecture 6 Summary and Checkpoint Exercises I

Synthesizing base conversion, representations, and applications.

Lecture 6 Summary and Checkpoint Exercises II

Key Points

- Key Concept: Octal (radix-8, 3-bit groups) and Hexadecimal (radix-16, 4-bit groups) are representations used to express binary numbers in compact, human-readable form.
- Engineering Checkpoint 1: Convert the decimal number 254.375_10 to both octal and hexadecimal representation.
- Engineering Checkpoint 2: Translate the hexadecimal value 0x3F8C directly to octal using the binary bridge method.
- Engineering Checkpoint 3: Solve the following arithmetic problem: 742_8 - 657_8, showing all borrow steps in radix-8.

Binary Logic and Boolean Variables I

Binary logic deals with variables that take on two discrete values: 1 (True) and 0 (False). It provides the mathematical framework for analyzing and designing digital electronic systems.

Binary Logic and Boolean Variables II

Key Points

- Boolean variables are represented by letters (e.g., A, B, X, Y) and are physically mapped to voltage ranges in hardware.
- In positive logic systems, a High voltage level (e.g., 5V or 3.3V) represents logic 1, and a Low voltage level (e.g., 0V) represents logic 0.
- In negative logic systems, these definitions are reversed (Low represents logic 1, High represents logic 0).
- Binary signals represent discrete states rather than continuous values, ensuring noise immunity in electronic circuits.

The Three Fundamental Logic Operations I

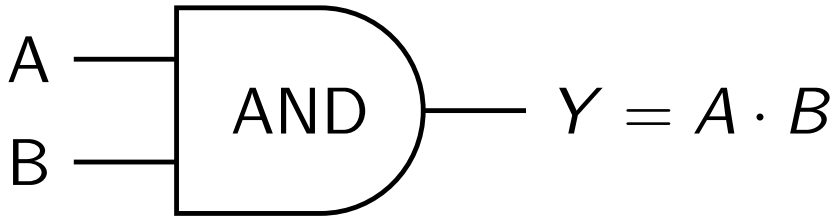
Every complex digital circuit is composed of combinations of three basic logic operations: AND (logical multiplication), OR (logical addition), and NOT (logical complementation).

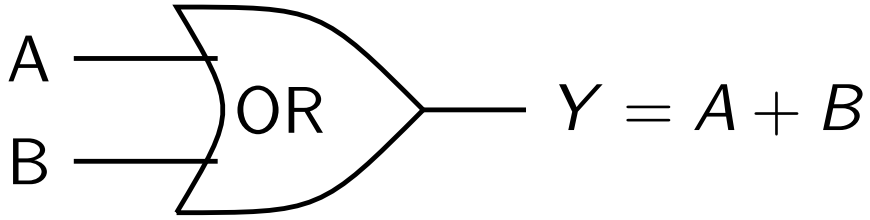
The Three Fundamental Logic Operations II

Key Points

- AND operation: Output is 1 if and only if all inputs are 1. Represented algebraically as $Z = A \cdot B$ or simply $Z = AB$.
- OR operation: Output is 1 if at least one input is 1. Represented algebraically as $Z = A + B$.
- NOT operation: Output is the inversion of the input. Represented algebraically as $Z = A'$ or $Z = \bar{A}$.
- These operations form a complete set, meaning any Boolean function can be expressed using only these three operations.

AND and OR Gates Circuit Symbols I





AND and OR Gates Circuit Symbols II

Key Points

- The AND gate symbol has a flat input side and a rounded output side.
- The OR gate symbol has a curved input side and a pointed output side.
- Logic gate schematic symbols are standardized globally (IEEE/ANSI Std 91-1984).

Truth Tables and Logic Functions I

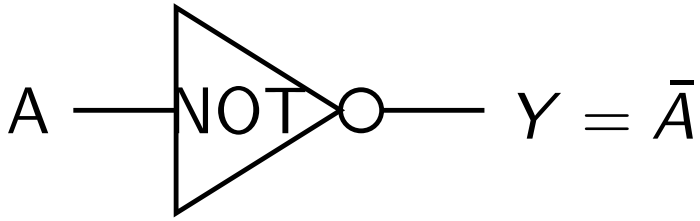
A truth table lists all possible combinations of input variables and the corresponding output state of a logic system.

Truth Tables and Logic Functions II

Key Points

- For 'n' binary input variables, a truth table contains exactly 2^n rows.
- Truth tables represent the mathematical behavior of a circuit, independent of its transistor-level implementation.
- They serve as the starting point for logical simplification tools like Karnaugh Maps (K-maps) and the Quine-McCluskey algorithm.
- Logic functions are equivalent if and only if their truth tables match row-for-row.

The NOT Gate and Inversion I



Key Points

- The NOT gate (inverter) consists of a buffer triangle and a small inversion circle (bubble).
- The bubble represents the logical negation operation in schematic diagrams.
- Bubbles can be placed on inputs (active-low inputs) or outputs (active-low outputs).

Derived Logic Gates: NAND and NOR I

NAND and NOR gates are constructed by adding an inverter to the output of AND and OR gates, respectively. They are highly significant because they are universal gates.

Derived Logic Gates: NAND and NOR II

Key Points

- NAND gate: Output is 0 only when all inputs are 1. Expressed as Y = (AB)'.
- NOR gate: Output is 1 only when all inputs are 0. Expressed as Y = (A+B)'.
- Universality: Any Boolean function (AND, OR, NOT) can be constructed using only NAND gates or only NOR gates.
- In physical CMOS technology, NAND and NOR gates are simpler, faster, and consume less area than standard AND and OR gates.

Exclusive-OR (XOR) and Exclusive-NOR (XNOR) I

XOR and XNOR gates perform comparison and parity operations, which are fundamental to arithmetic and error-detection circuits.

Exclusive-OR (XOR) and Exclusive-NOR (XNOR) II

Key Points

- XOR (Exclusive-OR): Output is 1 if and only if the inputs are different. Expressed as $Y = A \oplus B = A'B + AB'$.
- XNOR (Exclusive-NOR): Output is 1 if and only if the inputs are identical. Expressed as $Y = A \odot B = AB + A'B'$.
- Arithmetic operations: The XOR gate computes the sum bit in half-adders and full-adders.
- Error handling: XOR gates are used to build parity generators, parity checkers, and cyclic redundancy check (CRC) circuits.

Fundamental Laws of Boolean Algebra I

Boolean algebra consists of a set of algebraic rules used to simplify Boolean expressions, minimizing the gate count in physical hardware.

Fundamental Laws of Boolean Algebra II

Key Points

- Identity Laws: $A + 0 = A$, and $A \cdot 1 = A$.
- Null Laws: $A + 1 = 1$, and $A \cdot 0 = 0$.
- Idempotent Laws: $A + A = A$, and $A \cdot A = A$.
- Complementarity Laws: $A + A' = 1$, and $A \cdot A' = 0$.
- Distributive Law: $A(B + C) = AB + AC$, and $A + BC = (A + B)(A + C)$.

De Morgan's Theorems and Gate Duality I

De Morgan's theorems allow logical operators to be interchanged through negation, which is critical for transforming logic diagrams to use universal gates.

De Morgan's Theorems and Gate Duality II

Key Points

- First Theorem: The complement of a product is equal to the sum of the complements: (AB)' = A' + B'.
- Second Theorem: The complement of a sum is equal to the product of the complements: (A+B)' = A'B'.
- Duality Principle: Any Boolean identity remains valid if AND/OR operations and 0/1 values are systematically swapped.
- Hardware application: A NAND gate is logically equivalent to an OR gate with active-low inputs (Bubble-OR).

Physical Voltage vs. Logical State Mapping I

Physical digital systems process voltages, whereas digital architectures require logic states. Assigning logical values to voltage bands is done using specific mapping conventions.

Physical Voltage vs. Logical State Mapping II

Key Points

- Physical voltage ranges are divided into two distinct logical states: High State (V_H) and Low State (V_L).
- V_H is bounded by $V_IH(min)$ (minimum input high voltage) and V_DD or V_CC (the supply voltage).
- V_L is bounded by $V_IL(max)$ (maximum input low voltage) and GND (0V).
- The gap between $V_IL(max)$ and $V_IH(min)$ is the indeterminate state, ensuring noise margins are maintained.

Positive Logic System Convention I

Positive logic is the standard convention where the higher voltage state represents a boolean 1 (True) and the lower voltage state represents a boolean 0 (False).

Key Points

- Definition: V_H is mapped to Logic 1, and V_L is mapped to Logic 0.
- Active-High signals: Signals that execute their function when they transition to V_H.
- Matches intuitive physical mapping, representing the presence of signal energy as truth.
- Modern IC datasheets assume positive logic by default unless symbols like active-low bubbles are present.

Negative Logic System Convention I

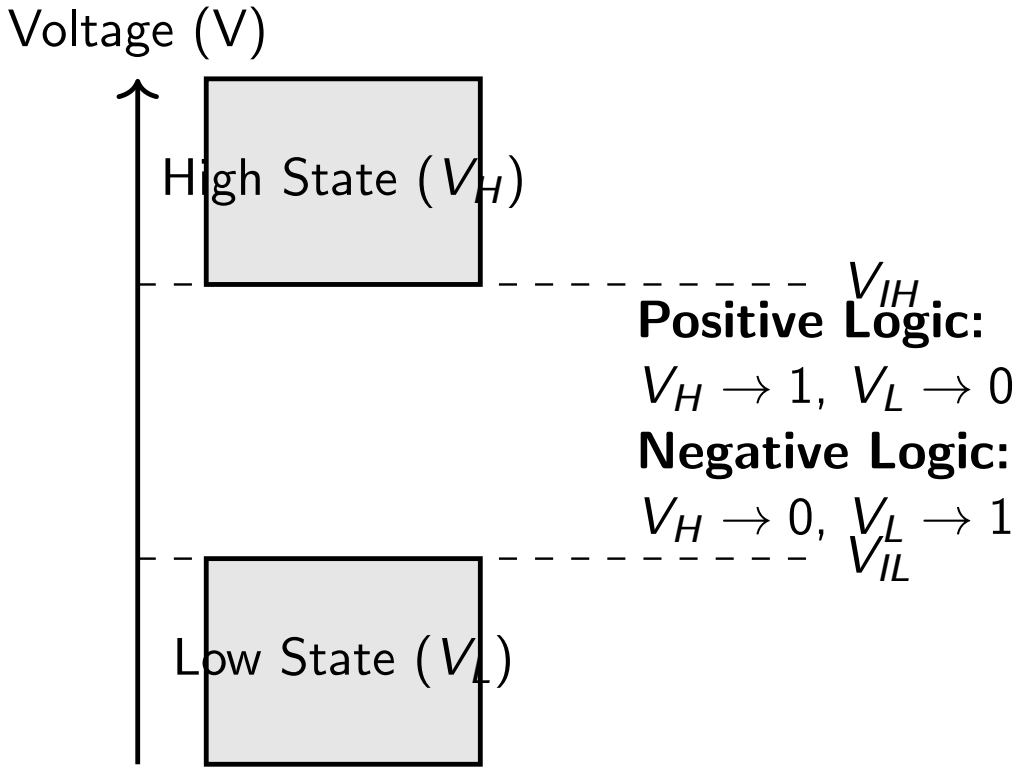
Negative logic is a convention where the lower voltage state represents a boolean 1 (True) and the higher voltage state represents a boolean 0 (False).

Negative Logic System Convention II

Key Points

- Definition: V_L is mapped to Logic 1, and V_H is mapped to Logic 0.
- Active-Low signals: Control signals that trigger an action at V_L (indicated with a line above the label or a slash).
- Commonly used in interface standards like RS-232 and for open-drain bus structures.
- Simplifies wired-logic implementations (e.g., Wired-AND / Wired-OR lines).

Mapping Duality Visualization I



Mapping Duality Visualization II

Key Points

- This diagram outlines the voltage ranges corresponding to low and high physical states.
- The physical states themselves remain unchanged under both conventions.
- Using positive logic, the high state translates to a logic high value.
- In negative logic systems, the high state behaves as a logic low value.

Logic Gate Duality Principle I

A physical logic gate maintains a constant physical response to input voltages, but its boolean logic representation depends entirely on the logic convention.

Logic Gate Duality Principle II

Key Points

- A physical gate that outputs a high voltage when all inputs are high acts as an AND gate in positive logic.
- The same physical gate acts as a logical OR gate when negative logic is applied to the inputs and output.
- Interchanging conventions changes logical AND operations into OR operations, and logical OR operations into AND operations.
- This logic inversion facilitates designing equivalent gate networks using De Morgan's laws.

The AND Gate under Positive Logic I

The AND gate performs a logical conjunction, outputting a logic 1 only if all of its inputs are set to logic 1.

Key Points

- Boolean notation: Y = A * B (or Y = AB).
- Positive logic truth table: Output is 1 when both inputs are 1; otherwise, the output remains 0.
- Electrical switch analogy: A series circuit configuration where all switches must be closed to complete the path.
- Typically used for masking operations and implementing enable lines.

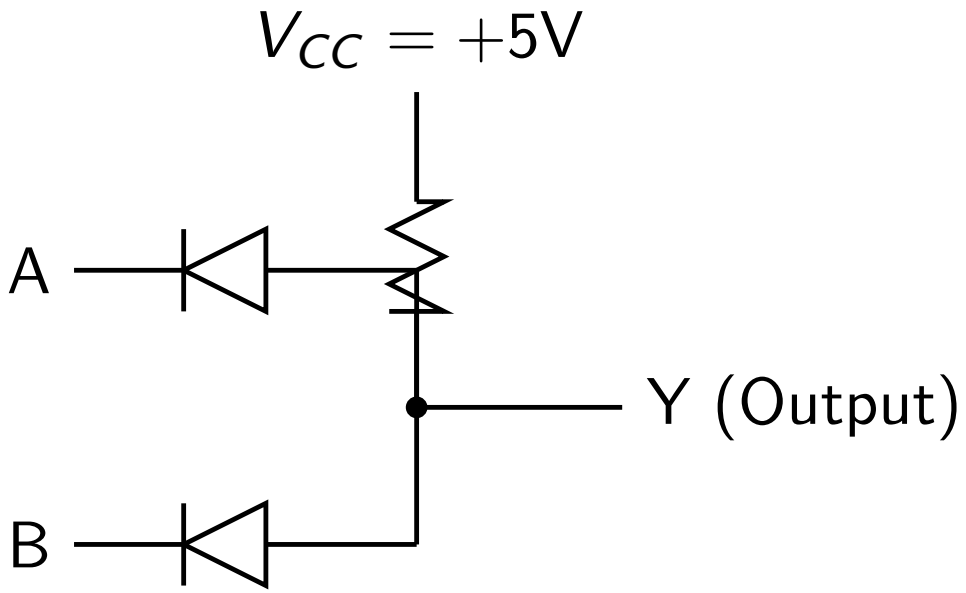
The OR Gate under Positive Logic I

The OR gate performs a logical disjunction, outputting a logic 1 if any of its inputs are set to logic 1.

Key Points

- Boolean notation: $Y = A + B$.
- Positive logic truth table: Output is 0 when all inputs are 0; otherwise, the output is 1.
- Electrical switch analogy: A parallel circuit configuration where any closed switch completes the path.
- Typically used to combine independent trigger signals or interrupt lines.

Diode Logic AND Gate Circuit I



Diode Logic AND Gate Circuit II

Key Points

- A physical schematic of a basic 2-input AND logic gate constructed with signal diodes.
- Operation: If either A or B is low (0V), that diode conducts, bringing the output node Y to a diode drop (0.7V) above ground.
- Operation: If both inputs A and B are high (+5V), both diodes are reverse-biased, allowing output Y to pull up to $V_C C$.
- Limitation: Diode logic circuits suffer from signal degradation and cannot be cascaded indefinitely.

Logic Equivalence Summary I

A reference matrix comparing functional logic transformations across positive and negative logic systems.

Key Points

- A hardware gate that acts as an AND gate in positive logic acts as an OR gate in negative logic.
- A hardware gate that acts as an OR gate in positive logic acts as an AND gate in negative logic.
- Understanding this equivalence ensures engineers can switch conventions to match physical constraints.
- Using positive and negative logic mapping correctly avoids signal phase inversion errors in logic paths.

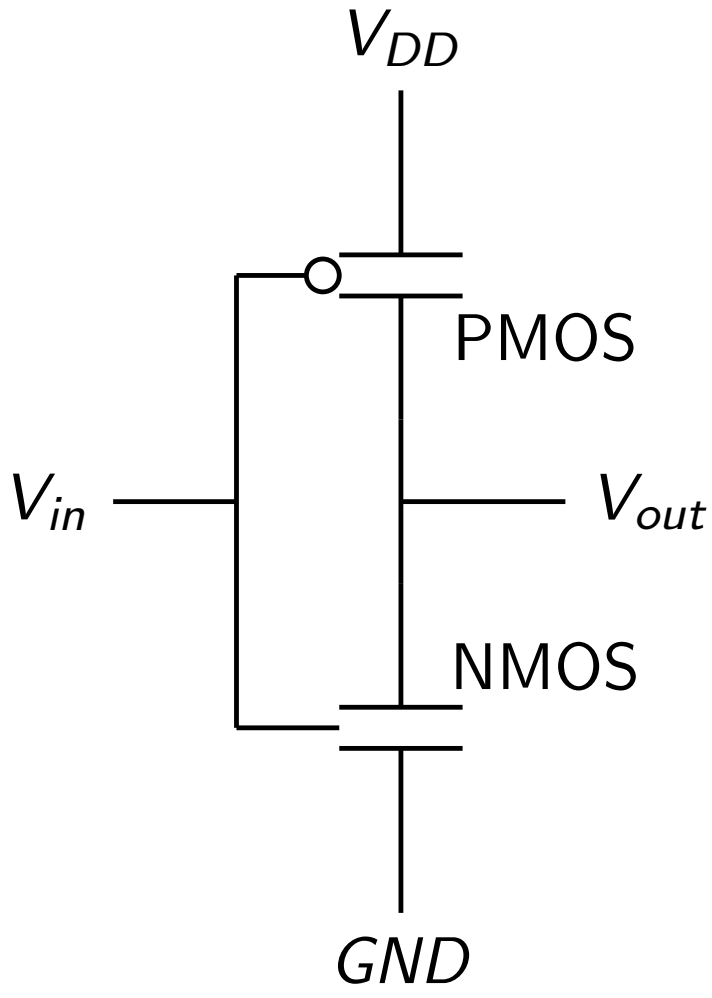
The NOT Gate: Fundamental Inversion I

The NOT gate, or logic inverter, is a single-input, single-output gate that performs Boolean negation (inversion). It maps a logic high (1) to a logic low (0), and vice versa.

Key Points

- Mathematical representation: Y = A' or Y = NOT A
- Truth table: Input 0 yields Output 1; Input 1 yields Output 0
- Serves as the foundation for state storage elements (latches, flip-flops) and clock generators
- Essential for active-low signal assertion and signal buffering applications

CMOS Inverter: Transistor-Level Implementation I



CMOS Inverter: Transistor-Level Implementation II

Key Points

- Uses a complementary pair of transistors: one PMOS (pull-up) and one NMOS (pull-down)
- When Vin is Low (0V), PMOS conducts and NMOS is cut off, pulling Vout to VDD (High)
- When Vin is High (VDD), NMOS conducts and PMOS is cut off, pulling Vout to GND (Low)
- No static power consumption occurs because one transistor is always turned off in steady state

Inverter Voltage Transfer Characteristic (VTC) I

The VTC of an inverter describes the relationship between the input voltage (Vin) and output voltage (Vout). It is critical for determining noise margins and switching thresholds.

Inverter Voltage Transfer Characteristic (VTC) II

Key Points

- VM (Switching Threshold): The point where Vin = Vout, typically designed at VDD/2
- Noise Margin High (NMH) = VOH - VIH; protects against noise when output is logic high
- Noise Margin Low (NML) = VIL - VOL; protects against noise when output is logic low
- Slope in the transition region represents the gain; a steeper slope indicates higher noise immunity

NOT Gate Propagation Delay and Power Dynamics I

Physical gates exhibit propagation delays (tpd) due to parasitic capacitance charging/discharging through transistor channels. Power dissipation consists of both static and dynamic components.

Key Points

- Propagation delay high-to-low (tPHL) and low-to-high (tPLH) define the switching speed
- Dynamic power is dominated by capacitive charging: P = C * VDD^2 * f
- Short-circuit current flows briefly during transition when both PMOS and NMOS conduct
- Sizing of transistors (W/L ratio) must be balanced to match rise and fall times

The Exclusive-OR (EX-OR) Gate I

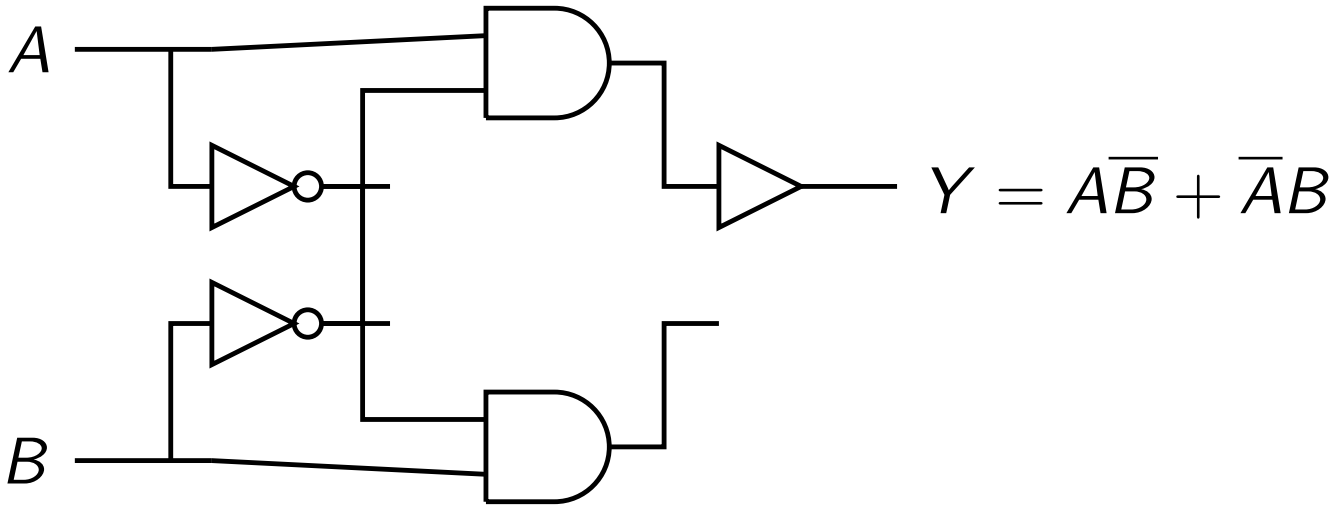
The EX-OR gate is a two-input logic gate that outputs a logic high (1) if and only if the inputs are different. It performs modulo-2 addition and behaves as a programmable inverter.

The Exclusive-OR (EX-OR) Gate II

Key Points

- Boolean equation: $Y = A \oplus B = AB' + A'B$
- Truth Table: Output is 1 for inputs (0,1) and (1,0); output is 0 for (0,0) and (1,1)
- Acts as a parity generator/checker: outputs 1 for odd parity of inputs
- If one input is held High, the EX-OR gate acts as a NOT gate for the other input

EX-OR Sum of Products Realization I



EX-OR Sum of Products Realization II

Key Points

- Schematic displays how two NOT gates, two AND gates, and one OR gate realize the XOR function
- Upper AND gate processes input A and negated input B (producing A B')
- Lower AND gate processes negated input A and input B (producing A' B)
- Output OR gate combines the partial products to yield the final XOR output

Algebraic Properties of EX-OR I

The EX-OR operator satisfies several algebraic properties that simplify complex logic optimization. Understanding these identities is vital for digital design synthesis.

Algebraic Properties of EX-OR II

Key Points

- Commutative: $A \oplus B = B \oplus A$
- Associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- Identity elements: $A \oplus 0 = A, and A \oplus 1 = A'(Inversion property)$
- Self-inverse (Nilpotence): $A \oplus A = 0$

Applications: Half Adder and Binary Addition I

The EX-OR gate is the fundamental arithmetic building block in modern processing units. It is the core logic element used to generate binary sums.

Key Points

- A Half Adder computes the sum of two single-bit inputs using an EX-OR gate
- The Carry output of a Half Adder is generated using an AND gate
- Multi-bit adders (Ripple Carry, Carry Lookahead) cascade EX-OR gates to compute multi-bit sums
- Used in ALU designs to implement subtraction via 2's complement addition

Applications: Parity Generators & Comparators I

EX-OR gates excel at error-detection schemes and identity checking because they determine bit inequality with minimal transistor counts.

Applications: Parity Generators & Comparators II

Key Points

- Parity Generator: Cascaded EX-OR gates count odd/even number of ones in a data word
- Digital Comparator: Cascading EX-OR (or EX-NOR) gates checks if two multi-bit registers are identical
- Pseudo-Random Binary Sequence (PRBS) generators use EX-OR in Linear Feedback Shift Registers (LFSRs)
- Fundamental component in cryptographic hardware engines for symmetric key mixing (XOR cipher)

The NOR Gate: Definition and Truth Table I

The NOR (Not-OR) gate is a fundamental logic gate that produces a High output (1) only when all of its inputs are Low (0). If any input is High (1), the output is Low (0).

The NOR Gate: Definition and Truth Table II

Key Points

- Boolean algebra expression: $Y = \overline{A + B}$.
- Truth table inputs (A, B) -¿ Output (Y): (0,0)-¿1, (0,1)-¿0, (1,0)-¿0, (1,1)-¿0.
- It acts as the logical negation of the disjunction operator (OR).
- In terms of set theory, it represents the complement of the union of two sets.

The NAND Gate: Definition and Truth Table I

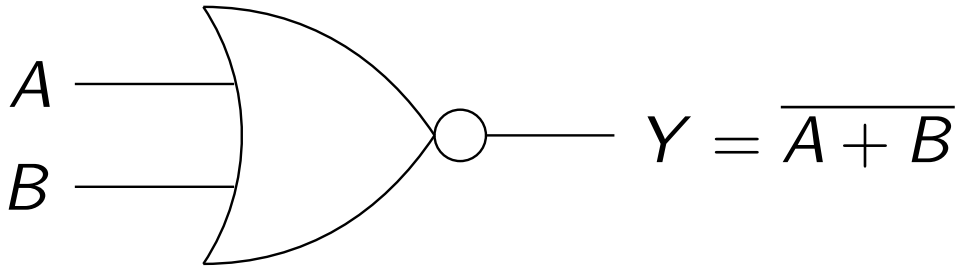
The NAND (Not-AND) gate is a fundamental logic gate that produces a Low output (0) only when all of its inputs are High (1). If any input is Low (0), the output is High (1).

The NAND Gate: Definition and Truth Table II

Key Points

- Boolean algebra expression: $Y = \overline{A \cdot B}$.
- Truth table inputs (A, B) -¿ Output (Y): (0,0)-¿1, (0,1)-¿1, (1,0)-¿1, (1,1)-¿0.
- It acts as the logical negation of the conjunction operator (AND).
- NAND is the primary storage technology design base for non-volatile Flash memory cells.

NOR Gate Schematic Symbol I

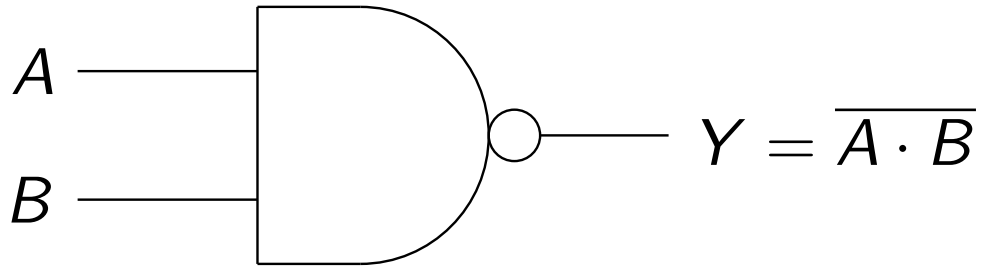


NOR Gate Schematic Symbol II

Key Points

- The NOR gate symbol features a curved input face, pointing output shape, and an inversion bubble.
- The inversion bubble at the output signifies the active-low nature of the logical negation.
- IEEE Std 91-1984 also defines rectangular-shape symbols, but the distinctive shape shown here is widely preferred.
- Inputs A and B enter the curved back, showing the OR relationship before inversion.

NAND Gate Schematic Symbol I



NAND Gate Schematic Symbol II

Key Points

- The NAND gate symbol consists of a straight input face, a semi-circular output face, and an inversion bubble.
- Like the NOR gate, the bubble indicates that the logical conjunction is inverted.
- It represents the combination of an AND gate and a NOT gate in a single compact schematic representation.
- The inputs A and B enter the flat back of the gate structure.

De Morgan's Theorems and Gate Equivalence I

De Morgan's laws state that the complement of a union is the intersection of the complements, and vice versa.

De Morgan's Theorems and Gate Equivalence II

Key Points

- Theorem 1: $\overline{A \cdot B} = \overline{A} + \overline{B}$. *ANANDgateisequivalenttoaBubbledORgate.*
- Theorem 2: $\overline{A + B} = \overline{A} \cdot \overline{B}$. *ANORgateisequivalenttoaBubbledANDgate.*
- These dualities allow engineers to swap gate types to match available library cells during logic optimization.
- De Morgan equivalence is key to reducing the critical path delay in complex logic networks.

Universality of the NAND Gate I

A gate is universal if it can implement all possible Boolean functions. The NAND gate can implement NOT, AND, and OR operations.

Universality of the NAND Gate II

Key Points

- NOT operation: Tie both inputs of a NAND gate together (A NAND A $= \overline{A}$).
- AND operation: Place a NAND NOT configuration after a standard NAND gate ($\overline{\overline{A \cdot B}} = A \cdot B$).
- OR operation: Invert both inputs using NAND NOTs, then pass them to a NAND gate ($\overline{\overline{A} \cdot \overline{B}} = A + B$).
- Since NOT, AND, and OR form a complete set, any logic circuit can be built exclusively with NAND gates.

Universality of the NOR Gate I

Like NAND, the NOR gate is a universal gate and can recreate any logical function.

Universality of the NOR Gate II

Key Points

- NOT operation: Tie both inputs of a NOR gate together (A NOR A $= \overline{A}$).
- OR operation: Place a NOR NOT configuration after a standard NOR gate ($\overline{\overline{A + B}} = A + B$).
- AND operation: Invert both inputs using NOR NOTs, then pass them to a NOR gate ($\overline{\overline{A} + \overline{B}} = A \cdot B$).
- Universality allows manufacturing facilities to standardize on single-gate wafers, lowering production costs.

CMOS Transistor Stacking and Performance Trade-offs I

In CMOS technology, NAND and NOR gates are implemented using complementary networks of PMOS and NMOS transistors.

CMOS Transistor Stacking and Performance Trade-offs II

Key Points

- NAND CMOS structure: Parallel PMOS pulling up to VDD, series NMOS pulling down to GND.
- NOR CMOS structure: Series PMOS pulling up to VDD, parallel NMOS pulling down to GND.
- PMOS mobility is lower than NMOS mobility, meaning series PMOS transistors (in NOR) must be made wider.
- Because of the wider PMOS transistors in series, NOR gates have higher input capacitance and lower switching speed than NAND gates.

Summary of Universal Gate Design I

Modern digital design relies heavily on universal gates for optimization, physical layout, and fabrication efficiency.

Key Points

- NAND gates are generally preferred over NOR gates in CMOS due to higher speed and smaller layout area.
- Universality enables complete logical design using uniform silicon structures.
- De Morgan's laws act as the mathematical bridge to convert between NAND and NOR representations.
- Understanding these physical and logical trade-offs is essential for high-performance VLSI design.

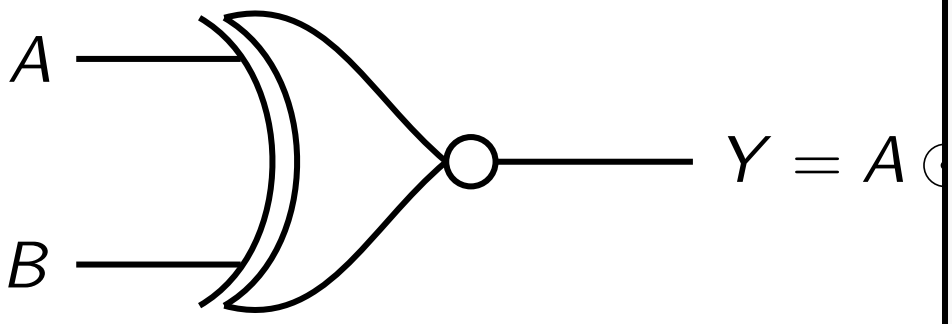
The Exclusive-NOR (EX-NOR) Gate I

The EX-NOR gate is a logical gate that outputs 1 only when both inputs are equal, serving as a logic equivalence detector.

Key Points

- Also known as the Equivalence gate or Coincidence gate.
- Algebraic expression: $\mathrm{Y} = \mathrm{A} \odot B = AB + A'B'$.
- Outputs high (1) when inputs are identical (00 or 11) and low (0) when they differ (01 or 10).
- Crucial for parity checkers, digital comparators, and error-correcting circuits.

EX-NOR Schematic and Truth Table I



EX-NOR Truth Table

A	B	$Y = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

EX-NOR Schematic and Truth Table II

Key Points

- The schematic symbol adds an inversion bubble to the output of an XOR gate.
- Truth table confirms the equivalence detection behavior.
- Standard IEEE symbol and ANSI symbol representations.
- Can be implemented using a combination of NAND gates (requires exactly 4 or 5 depending on configuration).

Huntington's Postulates of Boolean Algebra I

Formulated by Edward V. Huntington in 1904, these formalize the mathematical structure of Boolean algebra over a set K with operators $+$ and $\cdot$.

Huntington's Postulates of Boolean Algebra II

Key Points

- Closure: The set K is closed under both binary operations $+$ (OR) and $\cdot(AND)$.
- Identity elements: 0 is the identity for $+$, and 1 is the identity for $\cdot$.
- Commutativity: both $+$ and $\cdot arecommutative : A + B = B + AandA \cdot B = B \cdot A.$
- Distributivity: $\cdot distributesover + and + distributesover \cdot :$ $A(B + C) = AB + ACandA + (BC) = (A + B)(A + C).$

Basic Boolean Laws & Theorems I

Derived from Huntington's postulates, these rules are used to manipulate and simplify logical expressions.

Basic Boolean Laws & Theorems II

Key Points

- Idempotent Law: $A + A = A$, and $A \cdot A = A$.
- Null (Boundedness) Law: $A + 1 = 1$, and $A \cdot 0 = 0$.
- Involution Law: $(A')' = A$ (double negation recovers the original variable).
- Identity Law: $A + 0 = A$, and $A \cdot 1 = A$.

Advanced Algebraic Theorems I

These theorems enable grouping, ordering, and absorption of variables in complex Boolean expressions.

Key Points

- Associative Law: $(A + B) + C = A + (B + C)$, and $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.
- Absorption Law: $A + AB = A$, and $A(A + B) = A$.
- Consensus Theorem: $AB + A'C + BC = AB + A'C$, and $(A+B)(A'+C)(B+C) = (A+B)(A'+C)$.
- Redundancy Law: $A + A'B = A + B$, and $A(A' + B) = AB$.

Duality and De Morgan's Laws I

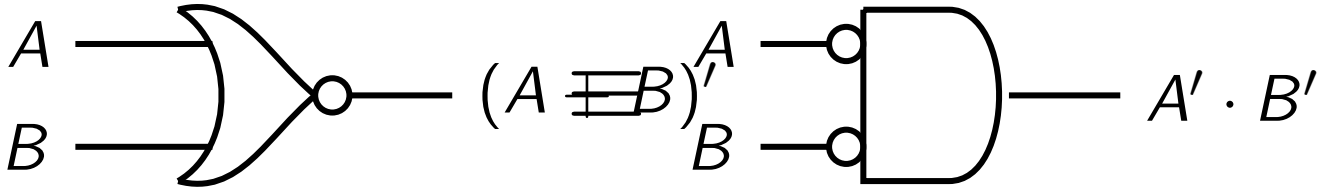
Essential tools for transforming logic structures and implementing gates using alternative logic families.

Duality and De Morgan's Laws II

Key Points

- Duality Principle: Any Boolean identity remains valid if $+$ and $\cdot are interchanged, and 0 and 1 are interchanged$.
- De Morgan's Theorem 1: (A + B)' = A' $\cdot B'$ (*NOR is equivalent to bubbled AND*).
- De Morgan's Theorem 2: (AB)' = A' + B' (NAND is equivalent to bubbled OR).
- Allows converting sum-of-products (SOP) to product-of-sums (POS) and vice-versa.

De Morgan's Logical Equivalence I



NOR Gate Bubbled AND

De Morgan's Logical Equivalence II

Key Points

- Visually demonstrates that a NOR gate is equivalent to an AND gate with inverted inputs.
- This equivalence is key to CMOS logic gate designs where PMOS and NMOS networks are duals.
- Bubble notation represents logic inversion (NOT operation) at inputs or outputs.
- Simplifies schematic analysis when mapping logic to physical NAND/NOR gates.

Algebraic Simplification of Boolean Functions I

Minimizing the number of terms and literals in a Boolean function to reduce physical gate count and propagation delay.

Algebraic Simplification of Boolean Functions II

Key Points

- Step 1: Apply distributive law to expand or factor expressions (e.g., F = AB + A(B+C) = AB + AB + AC = AB + AC).
- Step 2: Use complementarity and identity to eliminate variables (e.g., F = A'B + AB = (A'+A)B = 1 $\cdot B = B$).
- Step 3: Leverage absorption and consensus theorems for advanced reduction.
- Resulting expressions are either in minimal Sum-of-Products (SOP) or Product-of-Sums (POS) form.

Step-by-Step Algebraic Simplification I

Example problem: Simplify the Boolean function F = A'B'C + A'BC + AB'.

Key Points

- Factor out A'C from the first two terms: F = A'C(B' + B) + AB'.
- Apply Complementarity (B' + B = 1): F = A'C(1) + AB'.
- Apply Identity: F = A'C + AB'.
- The simplified expression F = A'C + AB' uses only 2 gates (AND, OR) instead of the original 3-term complex structure, significantly reducing hardware cost.

Foundations of Boolean Logic I

Introduced by George Boole in 1854, Boolean algebra is a mathematical system containing a set of elements B = 0, 1, two binary operators OR (+) and AND (.), and one unary operator NOT (').

Key Points

- Unlike standard algebra, Boolean algebra deals with logical relationships rather than numeric values.
- Variables are restricted to two states representing physical voltage levels (e.g., 0V for Low/0 and 5V for High/1).
- Provides the formal framework for analyzing gates, truth tables, and hardware description languages (HDLs).

Huntington's Postulates I

In 1904, E. V. Huntington proposed a set of formal axioms that define any Boolean algebra. These five postulates form the axiomatic basis for all subsequent theorems.

Huntington's Postulates II

Key Points

- Closure: For any a, b in B, a + b is in B, and a . b is in B.
- Identity Elements: There exists a 0 such that a + 0 = a, and a 1 such that a . 1 = a.
- Commutative Law: a + b = b + a, and a . b = b . a.
- Distributive Law: a + (b . c) = (a + b) . (a + c), and a . (b + c) = (a . b) + (a . c).
- Complement: For every element a, there exists a' such that a + a' = 1, and a . a' = 0.

Basic Single-Variable Theorems I

These basic theorems describe operations involving a single variable and constants. They are the most common rules applied during preliminary circuit minimization.

Key Points

- Idempotency: $A + A = A$ and $A \cdot A = A$.
- Null / Dominance: $A + 1 = 1$ and $A \cdot 0 = 0$.
- Identity: $A + 0 = A$ and $A \cdot 1 = A$.
- Involution: $(A')' = A$ (double negation cancels out).

Multi-Variable Commutative & Associative Laws I

These laws permit the rearrangement of inputs in multi-input logic gates without changing the circuit's output function.

Key Points

- Commutative Law: Identifies that input ordering does not affect AND/OR outputs (e.g., $A + B = B + A$).
- Associative Law: Allows grouping of operators, enabling the construction of N-input gates from cascading 2-input gates.
- Associative OR: $(A + B) + C = A + (B + C) = A + B + C$.
- Associative AND: $(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$.

Distributive Laws in Logic I

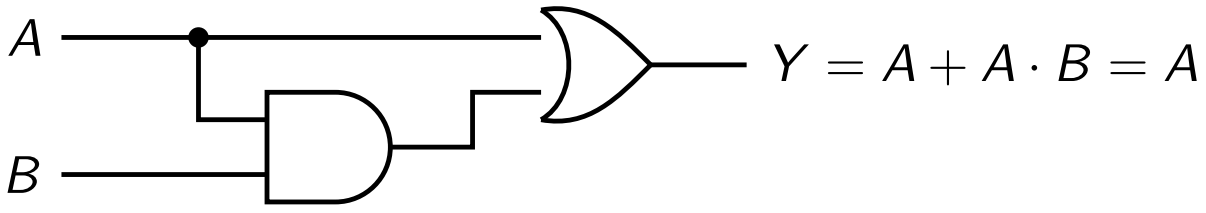
Distributive laws govern how terms are expanded or factored. Boolean algebra has two distributive laws, one of which is highly non-intuitive compared to standard algebra.

Distributive Laws in Logic II

Key Points

- First Distributive Law: AND distributes over OR: A . (B + C) = (A . B) + (A . C).
- Second Distributive Law: OR distributes over AND: A + (B . C) = (A + B) . (A + C).
- The second law is unique to Boolean algebra and does not hold in real-number arithmetic (where A + BC != (A+B)(A+C)).
- Proof of second law: (A+B)(A+C) = AA + AC + AB + BC = A + AC + AB + BC = A(1+C+B) + BC = A(1) + BC = A + BC.

Absorption Law and Gate Representation I

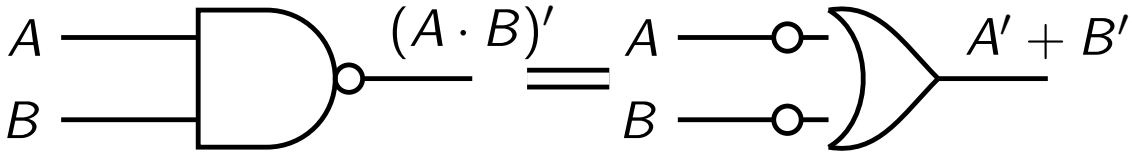


Absorption Law and Gate Representation II

Key Points

- Absorption Theorem 1: A + A . B = A. The term B is completely absorbed by A.
- Absorption Theorem 2: A . (A + B) = A (the dual of the first absorption theorem).
- Simplification Theorem: A + A' . B = A + B. This can be proved using the second distributive law: (A + A')(A + B) = 1 . (A + B) = A + B.
- Practical use: These laws reduce gate counts and input propagation delays in physical circuit implementations.

De Morgan's Theorem I



De Morgan's Theorem II

Key Points

- First Theorem: The complement of a product is equal to the sum of the complements, i.e., (A . B)' = A' + B'.
- Second Theorem: The complement of a sum is equal to the product of the complements, i.e., (A + B)' = A' . B'.
- De Morgan's laws facilitate the conversion between SOP (Sum of Products) and POS (Product of Sums) forms.
- Enables implementing all logic operations using only NAND or only NOR gates (universal gates).

Duality and Consensus Theorems I

The Consensus Theorem is a powerful simplification rule: $XY + X'Z + YZ = XY + X'Z$. Its dual is also valid: $(X + Y)(X' + Z)(Y + Z) = (X + Y)(X' + Z)$.

Duality and Consensus Theorems II

Key Points

- Consensus Theorem: The term YZ is redundant because if Y and Z are both 1, either X or X' must be 1, rendering one of the other terms true.
- Proof: $XY + X'Z + YZ(X + X') = XY + X'Z + XYZ + X'YZ = XY(1 + Z) + X'Z(1 + Y) = XY + X'Z$.
- Duality Principle: Every algebraic identity in Boolean algebra has a corresponding dual that is also valid.
- Duality is formed by swapping AND with OR, and 0 with 1, while preserving all variables and their individual complement states.

Logic Simplification Example I

Simplify the Boolean function F = A $\cdot B + A \cdot (B + C) + B \cdot (B + C)$ *usingthelawsofBooleanalgebra*.

Key Points

- Expand terms using Distributive Law: F = AB + AB + AC + BB + BC.
- Apply Idempotency (AB + AB = AB, BB = B): F = AB + AC + B + BC.
- Group terms containing B: F = AC + B(A + 1 + C).
- Apply Boundedness (A + 1 + C = 1) and Identity (B . 1 = B): F = B + AC.

Introduction to De-Morgan's Theorems I

De Morgan's Theorems establish mathematical equivalencies for complementing logical AND and OR combinations of variables. These theorems form the cornerstone of Boolean algebraic simplification and hardware optimization.

Introduction to De-Morgan's Theorems II

Key Points

- Formulated by Augustus De Morgan, these theorems bridge the operations of conjunction and disjunction through negation.
- First Theorem: The complement of an AND gate output is equivalent to the OR of the complemented inputs.
- Second Theorem: The complement of an OR gate output is equivalent to the AND of the complemented inputs.
- They are critical for converting logic designs into forms that use only universal gates like NAND and NOR.

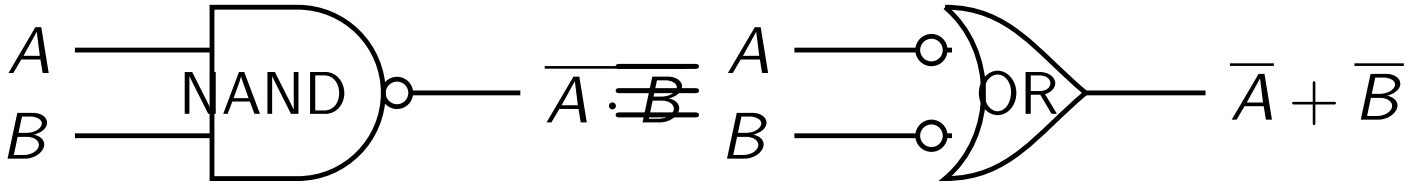
De-Morgan's First Theorem I

Theorem 1 states: $\overline{A \cdot B} = \overline{A} + \overline{B}$. *Itestablishesthatthecomplementofaproductisequaltothesumofthecom*

Key Points

- Left-Hand Side (LHS) represents a NAND operation on variables A and B.
- Right-Hand Side (RHS) represents a Bubbled OR operation, where inputs A and B are inverted before entering the OR gate.
- Allows the distribution of negation over multiplication, switching the operator from multiplication to addition.
- Generalization to N variables: $\overline{X_1} \cdot X_2 \cdots X_n = \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n}$.

Gate Equivalency: De-Morgan's First Theorem I



Key Points

- Visual representation of the equivalence between a NAND gate and a bubbled OR gate.
- Negation distributes over the conjunction, changing the operator to a disjunction.
- Crucial for physical CMOS layout optimization, where NAND gates are preferred due to speed and size.
- Allows logic gates to be transformed without changing the logic functionality.

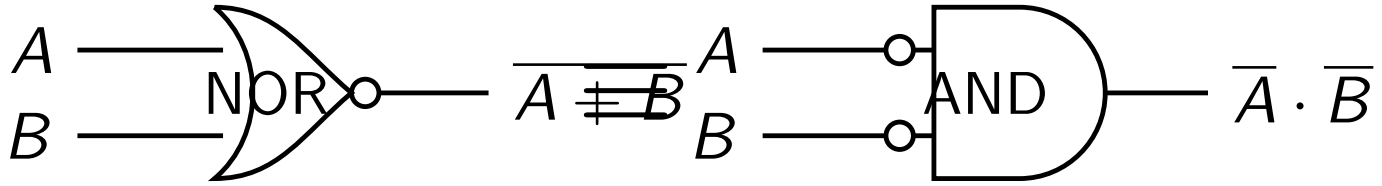
De-Morgan's Second Theorem I

Theorem 2 states: $\overline{A + B} = \overline{A} \cdot \overline{B}$. *Itestablishesthatthecomplementofasumisequaltotheproductofthecom*

Key Points

- Left-Hand Side (LHS) represents a NOR operation on variables A and B.
- Right-Hand Side (RHS) represents a Bubbled AND operation, where inputs A and B are inverted before entering the AND gate.
- Allows the distribution of negation over addition, switching the operator from addition to multiplication.
- Generalization to N variables: $\overline{X_1} + X_2 + \cdots + X_n = \overline{X_1} \cdot \overline{X_2} \cdots \overline{X_n}$.

Gate Equivalency: De-Morgan's Second Theorem I



Key Points

- Visual representation of the equivalence between a NOR gate and a bubbled AND gate.
- Negation distributes over the disjunction, changing the operator to a conjunction.
- Essential for logic restructuring to reduce transistor counts in logic paths.
- Enables bubble-pushing techniques to convert standard AND-OR logic networks to NOR-only configurations.

Truth Table Verification I

Verification of both theorems is achieved by comparing output columns for all possible combinations of inputs A and B (00, 01, 10, 11).

Truth Table Verification II

Key Points

- For NAND vs Bubbled OR: $\overline{A \cdot B}$*outputs*$[1, 1, 1, 0]$. *Thesumofcomplements* $\overline{A} + \overline{B}$*alsooutputs*$[1, 1, 1, 0]$.
- For NOR vs Bubbled AND: $\overline{A + B}$*outputs*$[1, 0, 0, 0]$. *Theproductofcomplements* $\overline{A} \cdot \overline{B}$*alsooutputs*$[1, 0, 0, 0]$.
- Matches column-by-column, proving mathematical equivalence across the entire Boolean domain.
- Provides empirical, finite validation of De Morgan's theorems for two-variable logic.

Complementing Complex Boolean Functions I

To complement any arbitrary Boolean function F, we apply De Morgan's theorems recursively. The process involves swapping AND/OR operators and complementing every individual term.

Complementing Complex Boolean Functions II

Key Points

- Step 1: Replace all OR operators with AND, and all AND operators with OR.
- Step 2: Complement each individual variable, constant, or parenthesized sub-expression.
- Example: Given $F = A(B + C\overline{D})$, *itscomplementis* $\overline{F} = \overline{A} + (\overline{B} \cdot (\overline{C} + D))$.
- Preserving parentheses is critical to maintain the correct operator precedence of the transformed expression.

The Duality Principle I

The Duality Principle states that every Boolean algebraic identity has a dual identity that is also valid, obtained by interchanging AND/OR and 0/1.

The Duality Principle II

Key Points

- Unlike complementation, variables are NOT complemented in duality.
- If the equation $A + 0 = A$ is true, its dual equation $A \cdot 1 = Aisalsotrue$.
- De Morgan's theorems connect complements and duals: the complement of a function is equal to its dual with complemented variables.
- Crucial for translating design architectures between complementary CMOS pull-up and pull-down logic networks.

Practical Engineering Applications I

De Morgan's Theorems are fundamental to physical circuit layout and standard logic cell implementation.

Practical Engineering Applications II

Key Points

- Universal Gates: Allows all circuits to be constructed using only NAND gates or only NOR gates, simplifying manufacturing.
- Bubble Pushing: A visual layout technique where inverters are pushed through logic gates to minimize gate counts and gate delays.
- FPGA Optimization: Synthesis tools apply De Morgan's transformations to map logic expressions into lookup tables (LUTs) efficiently.
- Noise Margin and Fan-out: Restructuring logic paths can optimize critical paths for timing and electrical performance characteristics.

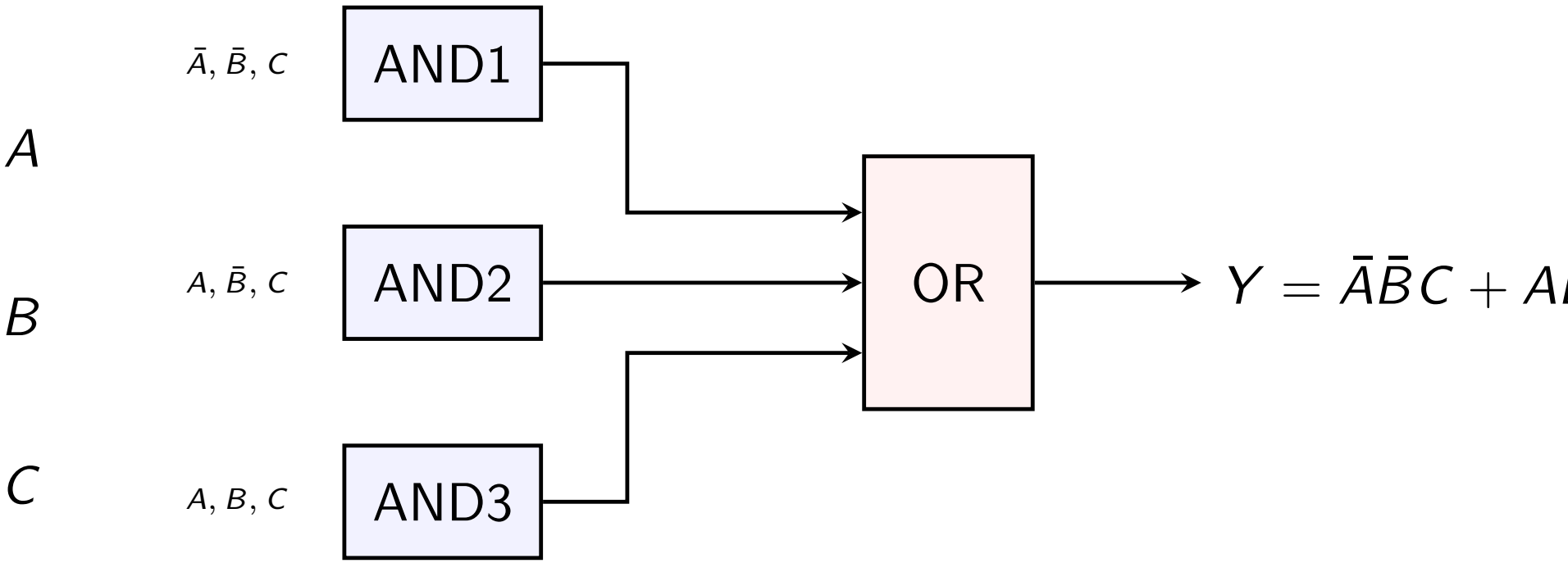
Introduction to Logic Simplification I

Logic simplification is the process of reducing the complexity of a Boolean expression to implement it with the minimum number of logic gates and interconnections, without altering its logical behavior.

Key Points

- A Boolean function can be represented in multiple algebraically equivalent forms.
- Canonical forms (SOP/POS) are often highly redundant and inefficient for direct hardware realization.
- Simplification directly impacts physical metrics: chip area, cost, speed, and power dissipation.

Unsimplified Logic Realization I



Unsimplified Logic Realization II

Key Points

- Example Function: $Y = C + AC + ABC$.
- Realization requires three 3-input AND gates and one 3-input OR gate (excluding inverters).
- High transistor count and complex routing increase manufacturing defects and cost.
- Significant propagation delay due to multi-level gating and load capacitance.

Impact on Silicon Area and Packaging I

Integrated circuits (ICs) are priced based on die area. Logic simplification minimizes gate count and reduces the physical footprint of the layout.

Key Points

- Gate reduction directly translates to fewer transistors (e.g., CMOS requires 2N transistors for an N-input NAND/NOR).
- Smaller silicon area allows more dies per wafer, lowering the per-unit manufacturing cost.
- Reduced interconnect routing simplifies layout, decreases parasitic capacitance, and prevents routing congestion.

Propagation Delay and Speed I

Every logic gate introduces a propagation delay (t_pd) due to transistor switching and charging/discharging of parasitic capacitances. Minimizing levels of logic increases speed.

Key Points

- Propagation delay is cumulative across cascading levels of logic (critical path delay).
- Simplification reduces the logic depth (number of gates in series from input to output).
- Fewer gates and shorter interconnects lead to faster signal propagation, supporting higher clock frequencies.

Power Dissipation in CMOS I

Dynamic power dissipation in CMOS circuits is given by P = C * $V_d d\hat{}2 * f *$ $A, where Cisloadcapacitance and Aisactivityfactor. Simplificationtargetsbo$

Key Points

- Each gate removed eliminates its internal capacitance and input gate capacitance from the circuit.
- Reducing gate count lowers the overall switching capacitance (C) of the network.
- Fewer transitions mean lower dynamic power dissipation, extending battery life and reducing thermal management needs.

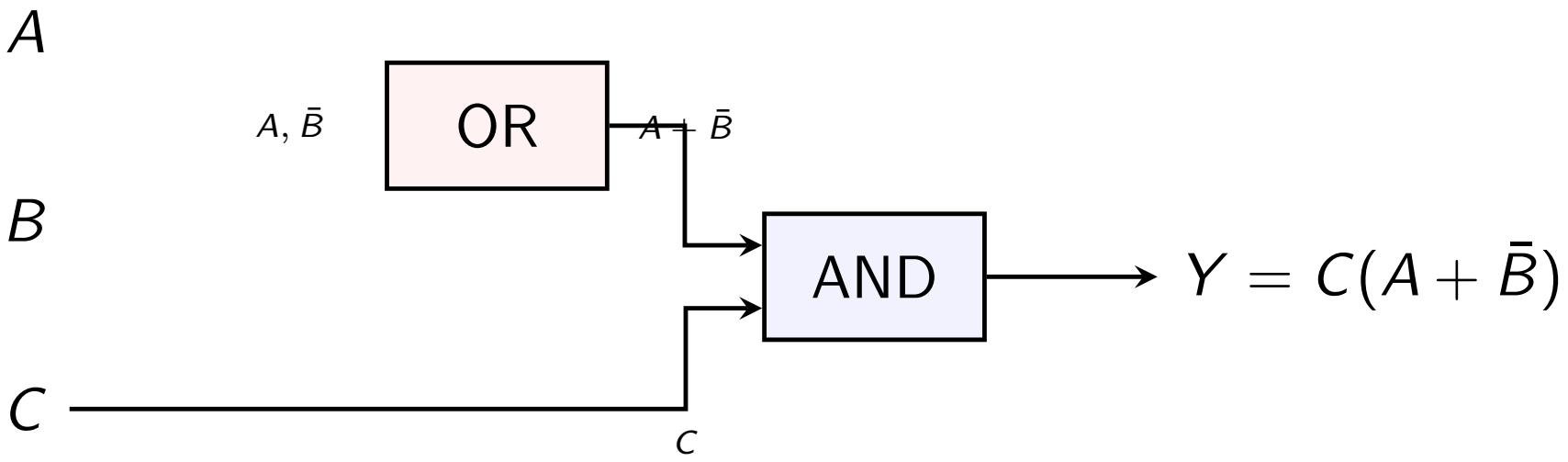
Fan-in and Fan-out Constraints I

Physical gates have limitations on the number of inputs they can accept (fan-in) and the number of loads they can drive (fan-out) without signal degradation.

Key Points

- High fan-in gates require larger, slower transistors in CMOS due to stacks of PMOS/NMOS transistors.
- Simplifying expressions eliminates terms, reducing the required fan-in of individual gates.
- Lower fan-in requirements permit faster transition times and more robust noise margins.

Simplified Logic Realization I



Simplified Logic Realization II

Key Points

- Simplified Function: $Y = C(A +)$.
- Realization requires only one 2-input OR gate and one 2-input AND gate.
- Saves 2 gates, simplifies routing, and reduces total transistor count from 24 (unsimplified SOP) to 10 (simplified).
- Reduces logic levels and dynamic power consumption substantially.

Methods of Simplification I

Several formal methodologies exist to simplify Boolean functions, ranging from manual algebraic manipulation to algorithmic computer-aided design (CAD) tools.

Key Points

- Boolean Algebra Theorems: Direct application of identity, distributive, consensus, and De Morgan's laws (often non-systematic).
- Karnaugh Maps (K-Maps): A visual, systematic method suitable for manual simplification of up to 4-6 variables.
- Quine-McCluskey Algorithm: A tabular method ideal for algorithmic execution in CAD tools for functions with a large number of variables.

Summary and Looking Ahead I

Simplification is not just a mathematical exercise; it is a fundamental hardware engineering necessity that defines the performance, cost, and efficiency of digital systems.

Key Points

- Reducing gate count, logic depth, and routing complexity improves speed, area, and power metrics.
- In the upcoming lectures, we will master Karnaugh Maps (K-Maps) as our primary tool for systematic manual simplification.
- Understanding the 'why' of simplification helps engineers design reliable, high-performance, and cost-effective digital chips.

The Principle of Logical Adjacency I

K-maps organize truth table rows such that adjacent cells differ by exactly one binary variable. This is known as logical adjacency, enabling the algebraic simplification: $AB + AB' = A(B + B') = A$.

Key Points

- A Karnaugh map is an array of cells where each cell represents a single minterm or maxterm.
- Adjacency rule: Cells sharing a physical boundary must differ by only one literal (Gray code order).
- This mapping translates algebraic minimization into visual pattern recognition of adjacent 1s or 0s.
- Eliminates the trial-and-error nature of symbolic Boolean algebraic simplification.

Gray Code Encoding in K-maps I

Unlike standard binary count (00, 01, 10, 11), K-map coordinates use Gray code sequence: 00, 01, 11, 10. This ensures that moving between any two adjacent cells (horizontally or vertically) results in only one variable changing state.

Gray Code Encoding in K-maps II

Key Points

- Binary count transitions from 01 to 10 change two bits, violating physical adjacency.
- Gray code (reflected binary code) guarantees a Hamming distance of 1 between adjacent columns or rows.
- A 1-bit change allows the factoring out of the changing variable: e.g., ABCD' + ABCD = ABC(D' + D) = ABC.
- Gray code sequences wrap around: the first and last columns/rows are also adjacent (Hamming distance = 1).

3-Variable K-map Layout I

begintikzpicture[scale=1.1, every node/.style=transform shape]
draw (0,0) grid (4,2);
draw (0,2) – (-0.8, 2.8) node[pos=0.7, above right] BC
node[pos=0.3, below left] A;
node at (-0.4, 1.5) 0;
node at (-0.4, 0.5) 1;
node at (0.5, 2.3) 00;
node at (1.5, 2.3) 01;
node at (2.5, 2.3) 11;
node at (3.5, 2.3) 10;
node at (0.5, 1.5) m_0;
node at (1.5, 1.5) m_1;
node at (2.5, 1.5) m_3;
node at (3.5, 1.5) m_2;
node at (0.5, 0.5) m_4;

3-Variable K-map Layout II

node at (1.5, 0.5) m_5;
node at (2.5, 0.5) m_7;
node at (3.5, 0.5) m_6;
endtikzpicture

3-Variable K-map Layout III

Key Points

- A 3-variable K-map contains $2^3 = 8$ cells, mapped to minterms m0 through m7.
- Rows represent variable A (0 or 1); columns represent variables BC in Gray code (00, 01, 11, 10).
- Note the non-sequential minterm ordering: columns 3 and 4 are swapped (m3 before m2, m7 before m6).
- Simplification groups can be of size 1, 2, 4, or 8 cells.

4-Variable K-map Minimization Example I

begintikzpicture[scale=0.9]
draw (0,0) grid (4,4);
draw (0,4) – (-0.8, 4.8) node[pos=0.7, above right] CD
node[pos=0.3, below left] AB;
node at (-0.4, 3.5) 00;
node at (-0.4, 2.5) 01;
node at (-0.4, 1.5) 11;
node at (-0.4, 0.5) 10;
node at (0.5, 4.3) 00;
node at (1.5, 4.3) 01;
node at (2.5, 4.3) 11;
node at (3.5, 4.3) 10;
node at (0.5, 3.5) 0;
node at (1.5, 3.5) 1;
node at (2.5, 3.5) 1;

4-Variable K-map Minimization Example II

node at (3.5, 3.5) 0;
node at (0.5, 2.5) 0;
node at (1.5, 2.5) 1;
node at (2.5, 2.5) 1;
node at (3.5, 2.5) 0;
node at (0.5, 1.5) 0;
node at (1.5, 1.5) 0;
node at (2.5, 1.5) 0;
node at (3.5, 1.5) 0;
node at (0.5, 0.5) 1;
node at (1.5, 0.5) 0;
node at (2.5, 0.5) 0;
node at (3.5, 0.5) 1;
draw[rounded corners=3pt, red, thick] (1.1, 2.1) rectangle (2.9, 3.9);

4-Variable K-map Minimization Example III

node[red, anchor=west] at (4.2, 3.0) Group 1 (Quad): $A'D$;
draw[blue, thick] (0, 0.2) – (0.8, 0.2) – (0.8, 0.8) – (0, 0.8);
draw[blue, thick] (4, 0.2) – (3.2, 0.2) – (3.2, 0.8) – (4, 0.8);
node[blue, anchor=west] at (4.2, 0.5) Group 2 (Pair): $AB'D'$;
endtikzpicture

4-Variable K-map Minimization Example IV

Key Points

- A 4-variable K-map contains $2^4 = 16$ cells, representing minterms m0 through m15.
- Red group forms a quad (4 cells: m1, m3, m5, m7), simplifying to A'D.
- Blue group forms a wrapping pair (2 cells: m8, m10), simplifying to AB'D'.
- Resulting simplified Sum-of-Products (SOP) expression: $Y = A'D + AB'D'$.
- This represents the absolute minimum gate realization for the given 1s.

Map Rolling and Group Sizes I

K-maps are not flat tables; they have a toroidal (donut-like) topology. The top edge wraps around to meet the bottom edge, and the left edge wraps around to meet the right edge. Group sizes must always be powers of 2 (1, 2, 4, 8, 16, ...).

Map Rolling and Group Sizes II

Key Points

- Wrapping property allows grouping cells across opposite boundaries (e.g., corners or opposite edges).
- A group of size 2 (pair) eliminates 1 variable from the minterm term.
- A group of size 4 (quad) eliminates 2 variables from the minterm term.
- A group of size 8 (octet) eliminates 3 variables, and so on.
- Grouping non-power-of-2 sizes (like 3 or 6 cells) is invalid and mathematically incorrect.

Implicants, Prime Implicants, and Essential Prime Implicants I

Understanding K-map terms: An Implicant is any single cell (minterm) or group of cells with a value of 1. A Prime Implicant (PI) is a group of adjacent 1s that cannot be merged into a larger group. An Essential Prime Implicant (EPI) is a PI that contains at least one '1' not covered by any other PI.

Implicants, Prime Implicants, and Essential Prime Implicants II

Key Points

- Every 1 in the K-map must be covered by at least one Prime Implicant in the final SOP expression.
- EPIs are critical: they must be included in the simplified Boolean expression to ensure completeness.
- Non-essential PIs can be selected or omitted depending on whether their 1s are already covered by EPIs.
- Systematic selection: (1) Find all PIs, (2) Identify all EPIs, (3) Cover remaining 1s with minimal set of PIs.

Don't Care Conditions (d or X) I

In some digital systems, certain input combinations cannot occur or their outputs are irrelevant. These are called 'Don't Care' conditions, represented by 'X' or 'd' in the truth table and K-map.

Key Points

- Don't Cares can be treated as either 0 or 1, depending on which choice yields a larger group.
- Larger groups eliminate more variables, leading to simpler expressions and less hardware gates.
- You do not have to group all 'X's; only include them if they help enlarge a group of 1s.
- Treating an 'X' as 1 is optional, whereas grouping all actual 1s is mandatory.

Product-of-Sums (POS) Simplification I

K-maps can also simplify functions into Product-of-Sums (POS) form. Instead of grouping 1s, we group the 0s (maxterms) in the K-map. The resulting expression is the complement of the function: F' = sum(grouped 0s). Applying De Morgan's theorem yields the POS form of F.

Product-of-Sums (POS) Simplification II

Key Points

- Grouping 0s is mathematically equivalent to finding the simplified SOP expression for the inverse function F'.
- To convert to POS, write the SOP expression for F' and apply De Morgan's laws: Y = (Y')'.
- Dual mapping: a '0' represents a maxterm where variables are summed (e.g., A + B + C' for binary 001).
- POS simplification is highly useful when there are fewer 0s than 1s in the K-map.

Limitations of K-maps and Alternative Methods I

While K-maps are highly visual and intuitive, they become difficult to construct and solve beyond 4 variables. 5-variable K-maps require two overlayed 4-variable maps (3D visualization), and 6-variable K-maps require four. Beyond 6 variables, algorithmic methods are preferred.

Limitations of K-maps and Alternative Methods II

Key Points

- Human visual pattern recognition breaks down when dealing with 5 or more variables.
- For N ¿ 4, Karnaugh maps require complex spatial visualization of adjacent planes.
- Tabular methods like the Quine-McCluskey algorithm are used for automated computer-aided design (CAD) tool implementation.
- Heuristic algorithms, such as the Espresso heuristic logic minimizer, are standard in modern VLSI synthesis software.

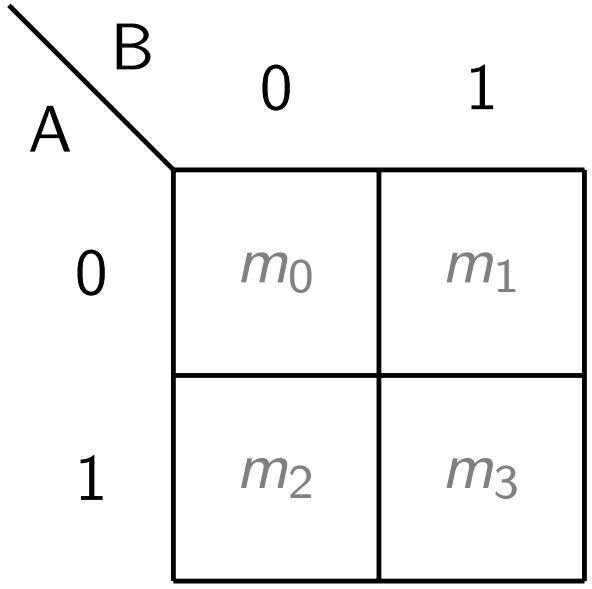
Limitations of Algebraic Simplification I

Simplification of Boolean algebraic expressions using theorems (like De Morgan's, Consensus, and Distributive laws) is non-systematic, lacks a deterministic stopping condition, and is highly prone to human error.

Key Points

- No structured procedure exists to guarantee that an expression has reached its minimal form.
- Simplification requires pattern recognition and trial-and-error algebraic manipulation.
- The Karnaugh Map (K-map) resolves this by providing a visual, systematic, and algorithmic approach to minimization.

Structure of a 2-Variable K-Map I



Structure of a 2-Variable K-Map II

Key Points

- A 2-variable K-map contains exactly $2^2 = 4$ cells, representing all possible combinations of two variables A and B.
- Variable A is mapped to the rows: Row 0 represents A=0 (or A') and Row 1 represents A=1 (or A).
- Variable B is mapped to the columns: Column 0 represents B=0 (or B') and Column 1 represents B=1 (or B).
- Cell addresses are concatenated binary indices mapping to minterms: m0 (00), m1 (01), m2 (10), and m3 (11).

Minterms and Cell Addresses I

Every cell in a 2-variable K-map represents a distinct minterm in the Sum-of-Products (SOP) form. The physical address of each cell corresponds to the specific states of variables A and B.

Key Points

- Cell m0 (binary 00) corresponds to the algebraic term A'B'.
- Cell m1 (binary 01) corresponds to the algebraic term A'B.
- Cell m2 (binary 10) corresponds to the algebraic term AB'.
- Cell m3 (binary 11) corresponds to the algebraic term AB.

The Adjacency Principle I

K-map simplification is mathematically rooted in grouping adjacent cells. Two cells are defined as adjacent if their binary coordinates differ by exactly one bit (Gray code order).

Key Points

- Adjacent cells share a common physical boundary (either a horizontal or vertical edge).
- Cell m0 (00) is adjacent to m1 (01) and m2 (10), but is not adjacent to m3 (11) because it differs by two bits.
- Grouping adjacent cells containing 1s allows us to factor out and eliminate the variable that changes state.

Mathematical Basis for Grouping I

Grouping adjacent cells is the visual representation of the Boolean algebraic simplification: XY' + XY = X(Y' + Y) = X(1) = X. This operation effectively eliminates the variable Y.

Key Points

- For example, grouping cells m0 (A'B') and m1 (A'B) yields: A'B' + A'B = A'(B' + B) = A'.
- The variable B changed from 0 to 1 across the group and was mathematically eliminated.
- The variable A remained constant at 0, thus appearing in the simplified output as literal A'.

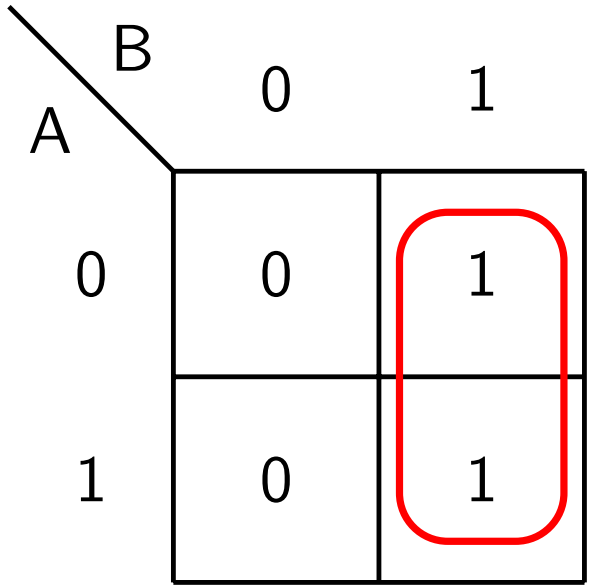
Grouping Rules and Constraints I

To systematically simplify a Boolean function using a K-map, we must adhere to specific rules for grouping cells containing 1s.

Key Points

- Group sizes must always be a power of two: 1, 2, or 4 cells in a 2-variable map.
- Groups must form a single rectangle or square; diagonal groupings are strictly invalid.
- All 1s must be covered by at least one group. Groups may overlap to maximize individual group sizes.
- The objective is to find the minimum number of groups with the largest possible dimensions.

Example 1: Simplification of F(A, B) = \{}sum m(1, 3) I



Example 1: Simplification of F(A, B) = \{}sum m(1, 3) II

Key Points

- Minterms m1 (01) and m3 (11) are mapped as 1s in the K-map.
- These cells are vertically adjacent and are grouped into a single loop of size 2.
- Along this vertical loop, variable A changes from 0 to 1 and is therefore eliminated.
- Variable B remains constant at 1, yielding the simplified minimal term: $F = B$.

Example 2: Overlapping Groups (F = \{}sum m(0, 1, 2)) I

When simplifying F(A, B) = $\sum m(0, 1, 2), wepopulatecells0, 1, and2with1s.Since3isnotapoweroftwo$

Key Points

- Group 1 covers m0 (00) and m1 (01). The constant variable is A = 0, giving the term A'.
- Group 2 covers m0 (00) and m2 (10). The constant variable is B = 0, giving the term B'.
- The simplified expression is the sum of these terms: F = A' + B'.
- This is logically equivalent to the NAND operation.

Summary of 2-Variable K-Maps I

The 2-variable K-map is the fundamental building block of graphical Boolean minimization.

Key Points

- A group of 1 cell represents a product term with 2 variables.
- A group of 2 cells eliminates 1 variable, representing a product term with 1 variable.
- A group of 4 cells covers the entire map and simplifies to 1.
- Understanding 2-variable K-maps prepares us for larger K-maps (3 and 4 variables).

Principles of Karnaugh Maps I

A Karnaugh Map (K-Map) is a graphical representation used to simplify Boolean expressions without using algebraic theorems.

Key Points

- It maps a truth table onto a grid where cells are arranged in Gray code sequence.
- Adjacent cells differ by exactly one binary variable (unit distance code), allowing physical adjacency to represent logical adjacency ($A + A' = 1$).
- It is standardly used for functions of 2, 3, 4, and sometimes up to 6 variables.

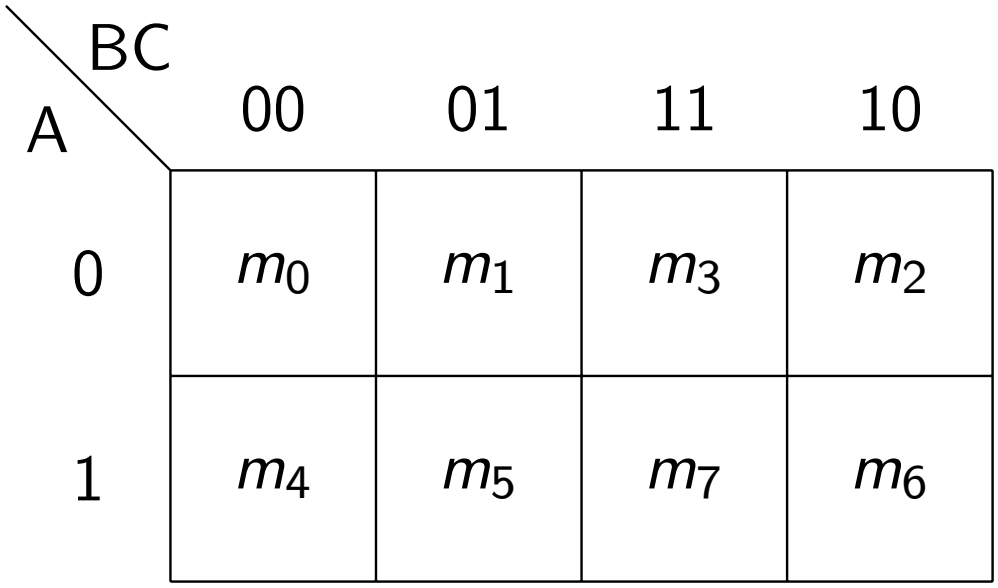
3-Variable K-Map Structure I

A 3-variable K-map simplifies functions of three variables (e.g., A, B, C) and consists of $2^3 = 8$ cells.

Key Points

- Typically organized as a 2x4 grid with rows representing one variable (A) and columns representing two variables (BC).
- Columns are ordered in Gray code sequence: 00, 01, 11, 10 to maintain adjacency between neighboring cells.
- Each of the 8 cells corresponds to a specific minterm: m_0 to m_7.

3-Variable K-Map Cell Mapping I



3-Variable K-Map Cell Mapping II

Key Points

- Rows represent the state of variable A (0 or 1).
- Columns represent the state of variables B and C in Gray code sequence (00, 01, 11, 10).
- Note the non-sequential minterm positioning: m_3 is before m_2 and m_7 is before m_6 due to Gray code column transitions.

Rules for Minimization and Grouping I

Simplification is achieved by grouping cells containing 1s (for Sum of Products form).

Key Points

- Group size must be a power of 2: 1, 2, 4, or 8 cells.
- Groups must be rectangular or square (diagonal grouping is strictly forbidden).
- The K-map wraps around: cells in the leftmost column are logically adjacent to cells in the rightmost column.
- Goal: Create the largest possible groups (least variables) with the minimum number of groups (least terms).

Example: 3-Variable Simplification I

Minimize the Boolean function: $F(A, B, C) = \sum m(1, 3, 5, 7)$.

Key Points

- Map 1s to cells m_1, m_3, m_5, and m_7.
- Identify a quad group of 4 adjacent cells: $\{m_1, m_3, m_5, m_7\}$.
- Within this group, variable A changes from 0 to 1, and variable B changes from 0 to 1, while variable C remains constant at 1.
- Eliminate changing variables (A and B) to obtain the simplified term: $F = C$.

4-Variable K-Map Structure I

A 4-variable K-map simplifies functions of four variables (e.g., A, B, C, D) and has $2^4 = 16$ cells.

Key Points

- Organized as a 4x4 grid where rows represent variables AB and columns represent variables CD.
- Both row and column headers follow the Gray code sequence: 00, 01, 11, 10.
- Cell indexing corresponds to minterms m_0 to m_{15}, reflecting the Gray code sequence in both directions.

4-Variable K-Map Layout I

AB \ CD	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

4-Variable K-Map Layout II

Key Points

- The 16-cell grid is indexed according to the intersections of AB and CD values.
- Notice the row jump: rows represent AB states in order (00, 01, 11, 10). Minterms m_8 to m_{11} are in the bottom row.
- Similarly, columns follow Gray code, placing minterms m_2, m_6, m_{14}, and m_{10} in the fourth column.

4-Variable Adjacency and Wrapping I

Adjacency in a 4-variable K-Map is multi-dimensional, extending horizontally, vertically, and through corner cases.

4-Variable Adjacency and Wrapping II

Key Points

- Top row cells are adjacent to bottom row cells (vertical wrap).
- Leftmost column cells are adjacent to rightmost column cells (horizontal wrap).
- The four corners $\{m_0, m_2, m_8, m_{10}\}$ are mutually adjacent and can form a group of 4.
- Grouping size hierarchy: Octet (8 cells, eliminates 3 variables), Quad (4 cells, eliminates 2 variables), Pair (2 cells, eliminates 1 variable).

Summary & Don't Care Conditions I

K-Maps are a powerful tool for visual optimization of Boolean logic functions.

Key Points

- Don't Care conditions (X) represent input combinations that never occur or whose output does not affect system behavior.
- Don't Cares can be treated as 0 or 1 to help create larger groups, leading to simpler expressions.
- Always identify Prime Implicants (PI) and Essential Prime Implicants (EPI) to systematically find the minimal Sum of Products (SOP).

Introduction to Don't Care Conditions I

In digital systems, some input combinations are either physically impossible or their resulting outputs are completely irrelevant to the system's operation.

Key Points

- Definition: Input states that cannot occur, or whose output values do not affect the system functionality.
- Representation: Denoted by 'X', 'd', or 'phi' in truth tables and K-Maps.
- Design Flexibility: During simplification, each don't care term can be treated as either a '0' or a '1'.
- Objective: To enlarge groupings (pairs, quads, octets) of 1s or 0s, resulting in a more simplified expression.

Mathematical Formulation & Representation I

A Boolean function with don't cares is defined by specifying both the active minterms/maxterms and the don't care conditions.

Mathematical Formulation & Representation II

Key Points

- Sum of Products (SOP) form: F(A, B, C, D) = $\sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$.
- Product of Sums (POS) form: F(A, B, C, D) = $\prod M(listofmaxterms) \cdot D(listofdon't careterms)$.
- Optional Coverage: We are not required to cover the don't care cells. They are only grouped if they simplify the main terms.
- Default Evaluation: Any don't care cell left ungrouped in SOP is effectively assigned a value of '0'.

Visualizing Don't Cares on a K-Map Grid I

AB \ CD	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

Visualizing Don't Cares on a K-Map Grid II

Key Points

- The grid represents a 4-variable K-Map for F(A,B,C,D) $= \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$.
- Don't cares are placed as 'X' in cells 0 (0000), 2 (0010), and 5 (0101).
- Standard minterms are placed as '1' in cells 1, 3, 7, 11, and 15.
- Empty/unused states default to '0'.

Rules for Grouping with Don't Cares I

Rules for logic minimization using K-Maps are modified to exploit the flexibility of don't cares.

Rules for Grouping with Don't Cares II

Key Points

- Rule 1: Group only '1's (or '0's for POS) and include 'X's if they enlarge the group size.
- Rule 2: Avoid making a group consisting entirely of 'X's. This would add an unnecessary term to the final expression.
- Rule 3: Not all 'X's need to be covered. We only care about covering the actual '1's.
- Rule 4: Standard adjacency rules (corners, edges wrapping) apply to 'X' cells just like regular cells.

Step-by-Step Minimization Procedure I

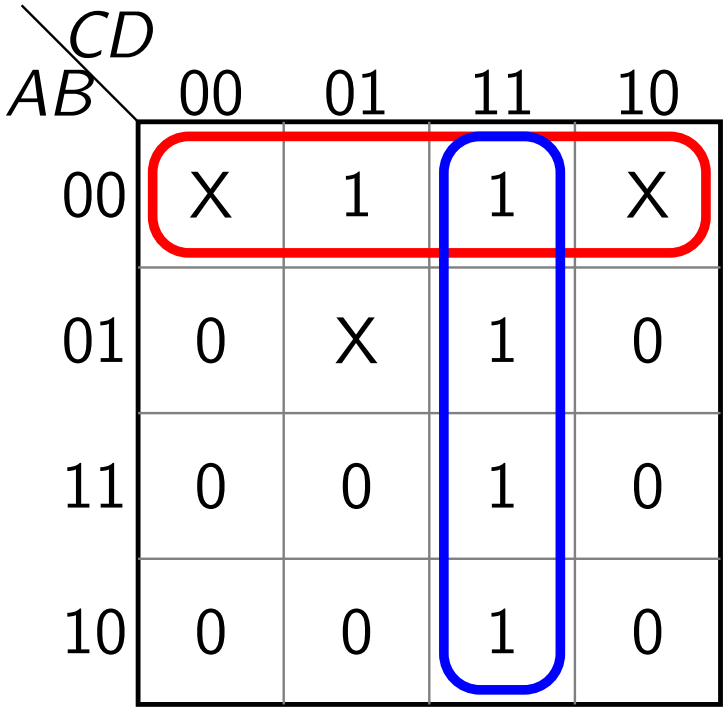
A systematic approach to minimizing a Boolean function with don't care conditions.

Step-by-Step Minimization Procedure II

Key Points

- Step 1: Plot all minterms as 1s, don't cares as Xs, and maxterms as 0s on the K-Map.
- Step 2: Identify any essential minterms (1s) that have only one possible grouping direction.
- Step 3: Form the largest possible groups (octets, quads, pairs) containing the 1s, using Xs as helpers.
- Step 4: Check if any 1s remain uncovered. Group them, prioritizing larger groups using remaining Xs.
- Step 5: Write the final Boolean expression by summing the simplified product terms.

Visualizing the Simplification & Groupings I



Visualizing the Simplification & Groupings II

Key Points

- Horizontal Quad (Red): Grouping cells 0, 1, 3, 2 treats cell 0 and 2 as '1'. Expression: A'B'.
- Vertical Quad (Blue): Grouping cells 3, 7, 15, 11 covers the column CD. Expression: CD.
- Ungrouped Don't Care: Cell 5 (0101) is not grouped and is treated as '0'.
- Final Optimized Boolean Function: F(A, B, C, D) = A'B' + CD.

Comparison of Simplification Results I

Comparing the logic circuits with and without using don't care conditions highlights the significant hardware savings.

Comparison of Simplification Results II

Key Points

- Without Don't Cares: Groupings are one quad (CD) and one pair (A'B'D) to cover cell 1. Output: F = CD + A'B'D.
- With Don't Cares: Groupings are two quads (CD and A'B'). Output: F = CD + A'B'.
- Literal Saving: The second term is reduced from 3 variables (A'B'D) to 2 variables (A'B').
- Hardware Benefit: Reduces logic gates from a 3-input AND gate to a 2-input AND gate, reducing propagation delay and power dissipation.

Real-World Application: BCD to 7-Segment Decoder I

Designing code converters is one of the most common applications of don't care simplification.

Real-World Application: BCD to 7-Segment Decoder II

Key Points

- BCD Constraints: Valid inputs are only decimal digits 0 to 9 (binary 0000 to 1001).
- Invalid States: The remaining 6 binary codes (1010 to 1111) are out-of-range and will never occur.
- Don't Cares mapping: The output lines for all segments (a to g) are configured as 'X' for inputs 10 through 15.
- Hardware Simplification: Using these 6 don't cares allows very simple 4-variable K-map expressions for all display segment drivers.

Summary & Design Rules of Thumb I

Key principles to keep in mind when designing digital circuits with don't care inputs.

Key Points

- Don't cares are flexible design resources that must be exploited for optimization.
- Always evaluate if treating an 'X' as '1' helps in doubling the group size (e.g. converting a pair to a quad).
- Never include an 'X' in a group if it doesn't help cover any active minterm (1).
- Leftover 'X's are automatically treated as '0' in SOP or '1' in POS; no hardware resources are wasted on them.

Introduction to Basic Logic Gates I

Digital circuits are constructed using three primary logic gates: AND, OR, and NOT. These gates form a functionally complete set, meaning any Boolean function can be implemented using combinations of these three basic gates.

Key Points

- AND Gate: Performs logical multiplication ($Y = A \cdot B$), where the output is HIGH only if all inputs are HIGH.
- OR Gate: Performs logical addition ($Y = A + B$), where the output is HIGH if at least one input is HIGH.
- NOT Gate: Performs logical inversion ($Y = \bar{A}$), which outputs the logical complement of the input signal.

Standard Forms for Hardware Implementation I

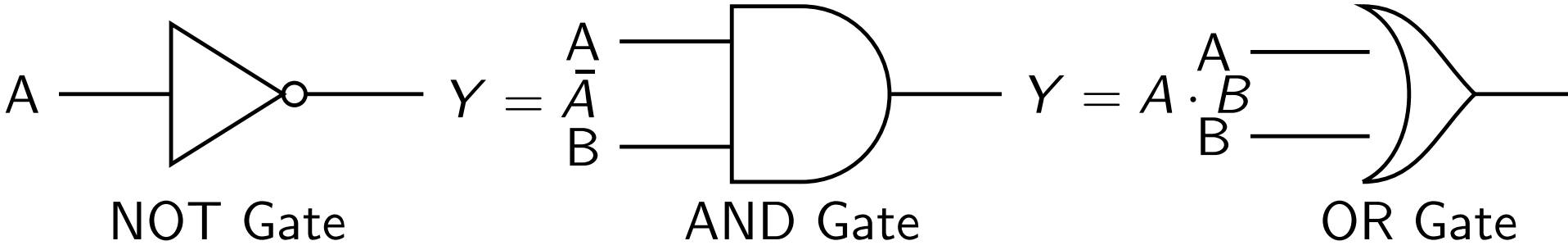
Before mapping a Boolean function to physical logic gates, the expression must be represented in a standard algebraic form. The two main standard representations are Sum-of-Products (SOP) and Product-of-Sums (POS).

Standard Forms for Hardware Implementation II

Key Points

- Sum-of-Products (SOP): Formed by ORing individual product terms. It directly maps to an AND-OR two-level gate structure.
- Product-of-Sums (POS): Formed by ANDing individual sum terms. It directly maps to an OR-AND two-level gate structure.
- Canonical Form: Min-terms (SOP) and Max-terms (POS) represent functions directly from truth tables before optimization.

Schematic Symbols of Basic Gates I



Schematic Symbols of Basic Gates II

Key Points

- IEEE/ANSI Standard graphic symbols for NOT, AND, and OR gates.
- The NOT gate contains an inversion bubble on its output node.
- The AND gate design features a flat input back-face and rounded output curve.
- The OR gate features a curved input back-face and a pointed output vertex.

Implementation of Sum-of-Products (SOP) I

To implement an SOP function such as $F = AB + C\bar{D}$: 1. Generate inverted literals using NOT gates. 2. Implement each product term using an AND gate. 3. Combine the outputs of the AND gates using a single multi-input OR gate.

Key Points

- Limits the logic path to a maximum of three levels: NOT level, AND level, and OR level.
- Preferred format for standard programmable logic arrays (PLAs).
- Ensures consistent propagation delay across all product term paths.

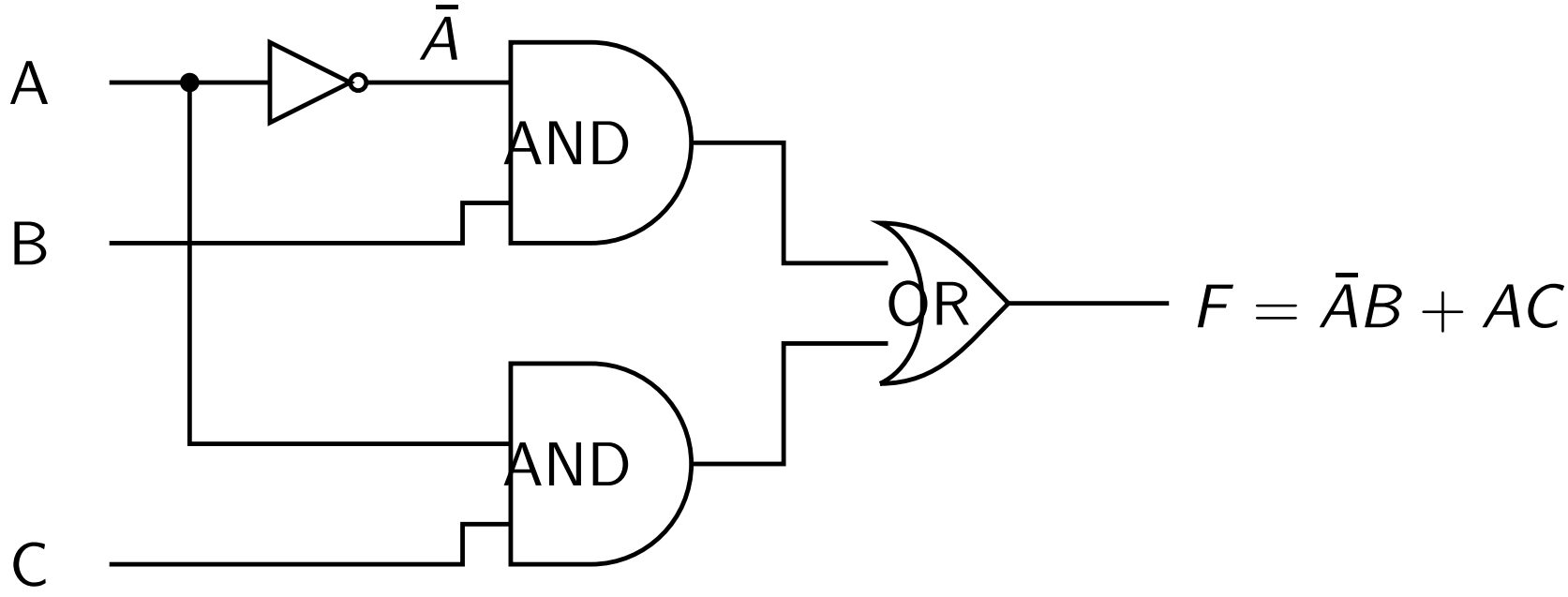
Implementation of Product-of-Sums (POS) I

To implement a POS function such as $F = (A + B) \cdot (\bar{A} + C)$: 1. Generate inverted literals using NOT gates. 2. Implement each sum term using an OR gate. 3. Combine the outputs of the OR gates using a single multi-input AND gate.

Key Points

- Configures logic in an OR-AND structure rather than AND-OR.
- Comprises level-1 NOT gates, level-2 OR gates, and a final level-3 AND gate.
- Provides a hardware-efficient alternative when the complement function contains fewer terms.

Practical Implementation of F = A'B + AC I



Practical Implementation of F = A'B + AC II

Key Points

- A clear, direct AND-OR implementation of the Boolean function $F = \bar{A}B + AC$.
- A single NOT gate generates the complemented input $\bar{A}$.
- Two-input AND gates isolate and compute the individual product terms.
- A final two-input OR gate integrates the terms to establish the output logic state.

Physical Constraints in Logic Gate Design I

Realizing Boolean equations in physical hardware requires adhering to electrical limits. Three critical parameters are Fan-In, Fan-Out, and Propagation Delay.

Key Points

- Fan-In: The maximum number of inputs a single logic gate can accept. High fan-in degrades switching speed.
- Fan-Out: The maximum number of gate inputs that a single gate output can drive without signal degradation.
- Propagation Delay (t_{pd}): The time delay between input change and output transition. Multi-level implementations increase overall delay.

The Importance of Simplification I

Directly implementing unsimplified Boolean functions leads to redundant gates, higher power consumption, and increased propagation delay. Simplification using Boolean Algebra or K-maps is mandatory before layout.

Key Points

- Example: Unsimplified $F = A\bar{B}C + AB\bar{C} + ABC$ requires three 3-input AND gates and one 3-input OR gate.
- Simplified form: $F = AB + AC$ requires only two 2-input AND gates and one 2-input OR gate.
- Fewer gates reduce the silicon area (cost) and dynamic power dissipation ($P = CV_{DD}^2 f$).

Summary of Logic Gate Implementation I

Implementing Boolean functions involves a systematic workflow: 1. Obtain Truth Table or Boolean expression. 2. Simplify using K-maps or algebraic laws. 3. Convert to SOP or POS form. 4. Map to NOT, AND, and OR gates, respecting fan-in/fan-out constraints.

Key Points

- Basic logic gates (AND, OR, NOT) are the building blocks of combinational circuits.
- A two-level design minimizes propagation delay at the cost of higher gate count or inputs.
- In practice, NAND and NOR gates are often used as universal gates to replace AND/OR/NOT configurations due to simpler CMOS construction.

Introduction to the OR Logic Operation I

The OR operation (logical disjunction) is one of the three fundamental Boolean operators used in digital systems.

Key Points

- Defined mathematically as $Y = A + B$ where $+$ denotes logical addition.
- The output of an OR gate is high (1) if at least one of its inputs is high (1).
- It is represented in circuit designs by a distinct curved input gate symbol.
- Used extensively in control logic, address decoding, and arithmetic units.

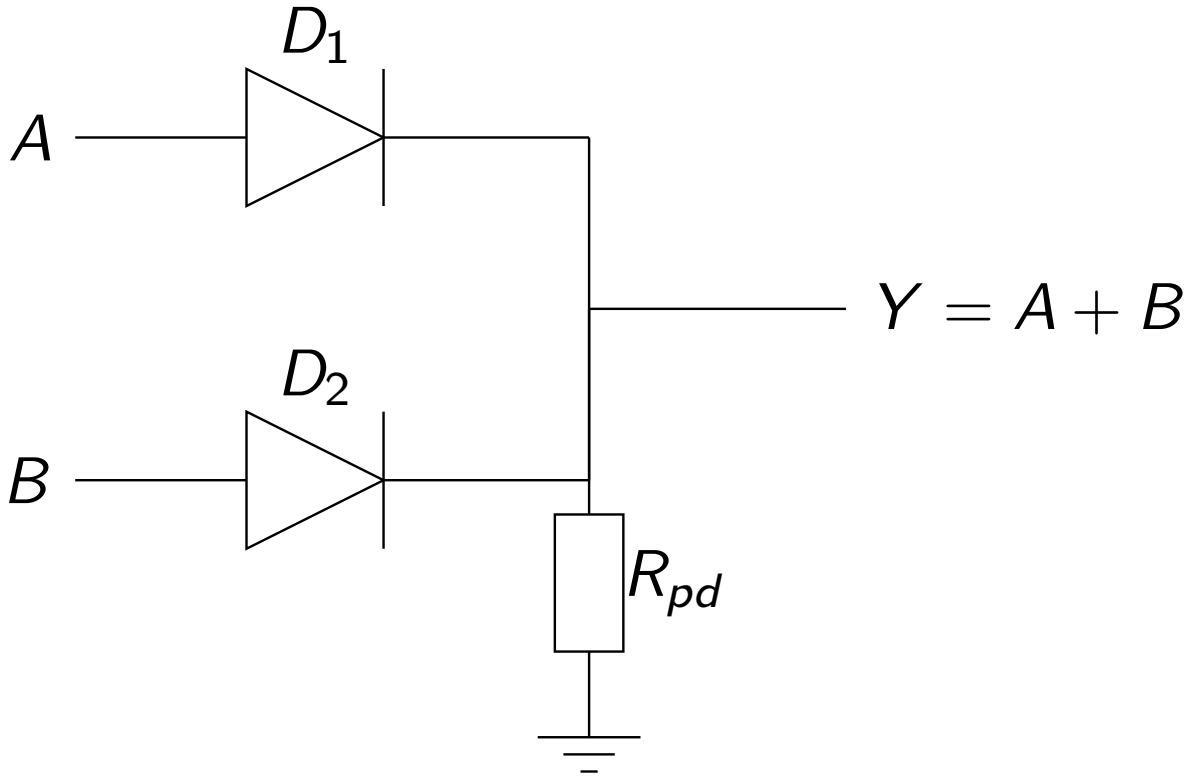
Truth Table and Algebraic Laws of OR I

An OR gate's behavior can be exhaustively characterized by its truth table and algebraic properties.

Key Points

- Truth table inputs: 0+0=0, 0+1=1, 1+0=1, 1+1=1.
- Identity Law: $A + 0 = A$.
- Null Law (Dominance): $A + 1 = 1$.
- Idempotent Law: $A + A = A$.
- Commutative and Associative properties: $A + B = B + A$, and $(A + B) + C = A + (B + C)$.

Diode Logic Realization of OR Gate I



Diode Logic Realization of OR Gate II

Key Points

- Uses two forward-biased PN junction diodes connected to a single pull-down resistor.
- If either input is high (e.g., 5V), the corresponding diode conducts, pulling the output high.
- The output voltage is $V_out = V_in$ - V_D, where V_D is the diode forward voltage drop (approx. 0.7V for Silicon).
- If both inputs are low (0V), both diodes are reverse-biased, and the output is pulled to 0V through R_pd.

Introduction to the NOT (Inverter) Operation I

The NOT operation (logical negation or inversion) is a unary operator that swaps the logic state of its input.

Key Points

- Represented mathematically as Y = A' or Y = barA.
- Truth table: Inverting 0 yields 1; inverting 1 yields 0.
- The circuit symbol is a buffer triangle followed by an inversion bubble (circle).
- Crucial for generating complementary signals in differential buses and register files.

Transistor Realization of NOT Gate I

Key Points

- Constructed using an NPN Bipolar Junction Transistor (BJT) in a common-emitter configuration.
- When V_in is High, the BJT is driven into saturation, pulling V_out down to $V_CE(sat)$ (approx. 0.2V).
- When V_in is Low, the BJT is in cutoff, and V_out is pulled up to V_CC through R_C.
- R_B limits base current to prevent transistor damage, and R_C defines the output pull-up strength.

Multi-Input OR Gates & Extension I

While basic OR gates have two inputs, practical systems require multi-input OR operations.

Key Points

- A multi-input OR gate yields a high output if any of its N inputs is high.
- Can be constructed by cascading two-input OR gates hierarchically.
- An N-input OR gate constructed with a tree of 2-input gates has a propagation delay proportional to log2(N).
- Direct multi-input realization is often done in CMOS using a parallel array of NMOS transistors.

Cascaded Logic: NOR as a Universal Gate I

Combining the OR and NOT operations yields the NOR (NOT-OR) operation, which is a universal logic gate.

Key Points

- NOR gate equation: Y = overlineA + B.
- Universality: Any Boolean function (AND, OR, NOT) can be constructed using only NOR gates.
- A NOT gate is created by tying all inputs of a NOR gate together (or tying unused inputs to GND).
- An OR gate is created by cascading a NOR gate with a NOT gate.

Physical Characteristics & Propagation Delay I

Real-world OR and NOT gates exhibit non-ideal behaviors such as propagation delay and power dissipation.

Physical Characteristics & Propagation Delay II

Key Points

- Propagation Delay (t_pd): The time required for a change in input to reflect at the output.
- Rise time (t_r) and Fall time (t_f) characterize the transition speed between high and low logic states.
- Static power dissipation is near zero in CMOS but significant in older TTL and PMOS/NMOS technologies.
- Dynamic power dissipation is proportional to switching frequency, load capacitance, and $V_C C$ squared.

Summary and Course Connection I

Understanding OR and NOT gates is foundational for higher-level Boolean simplification and digital design.

Key Points

- OR and NOT form a complete logic set when combined with AND (or on their own as NOR/NAND).
- In upcoming lectures, we will leverage these gates to build multiplexers, decoders, and adders.
- Boolean algebra simplification (Karnaugh Maps) directly maps expressions to networks of these gates.
- Hardware Description Languages (HDLs) compile behavioral OR/NOT statements into physical gate layouts.

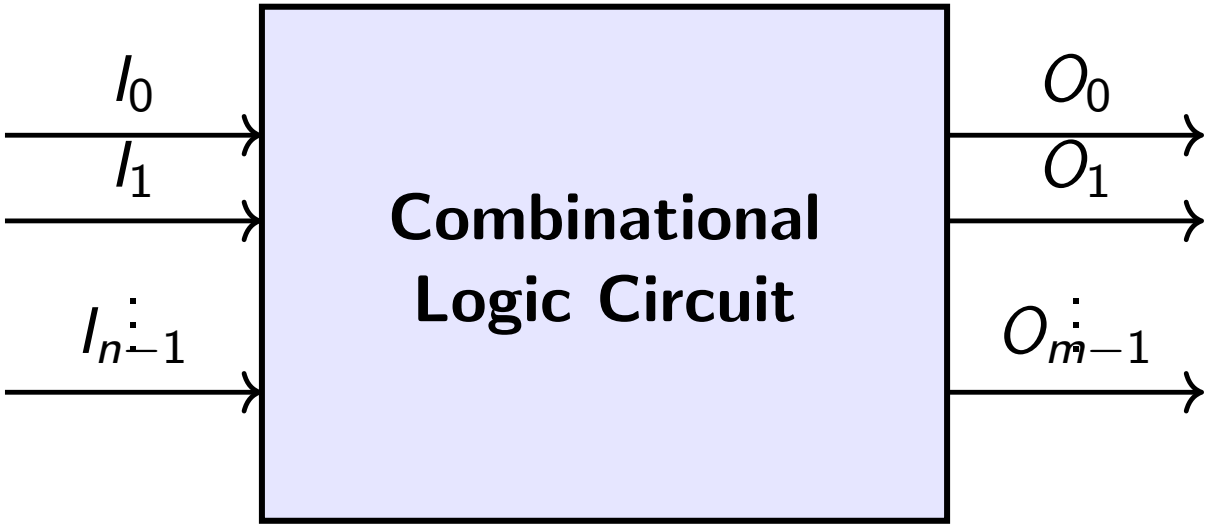
What is a Combinational Circuit? I

A combinational circuit is a system of interconnected logic gates whose outputs at any given instant are determined solely by the current combination of inputs.

Key Points

- Outputs respond immediately to input changes, delayed only by the propagation delay of the physical gates.
- Contains no storage or memory elements such as latches, flip-flops, or registers.
- There are no feedback loops from the output terminal back to any input terminal.
- Operates on binary variables, transforming n inputs into m outputs based on Boolean algebra.

Block Diagram Representation I



Block Diagram Representation II

Key Points

- An n-input circuit can receive up to 2^n possible input combinations.
- For each input combination, there is exactly one possible output state for each of the m outputs.
- Outputs are mathematically defined as: $O_i = f_i(I\_0, I\_1, ..., I\_n\text{-}1) for 0 <= i < m$.
- The logic design determines the layout and type of logic gates used internally.

Combinational vs. Sequential Logic I

Digital circuits are broadly classified into combinational and sequential circuits. The primary differentiating factor is the reliance on state.

Combinational vs. Sequential Logic II

Key Points

- Memory: Combinational circuits have no memory; sequential circuits require memory to store current state.
- Feedback: Combinational logic has no feedback loops; sequential logic relies on feedback path architectures.
- Clocking: Combinational circuits are asynchronous; sequential circuits are typically synchronous, regulated by a system clock.
- Complexity: Combinational design utilizes Boolean algebraic simplification; sequential design requires state tables.

Analysis of Combinational Circuits I

The analysis of a combinational circuit begins with a logic diagram and concludes with Boolean functions, truth tables, or timing diagrams.

Key Points

- Label all gate outputs that are functions of input variables only.
- Label intermediate gate outputs that depend on previously labeled outputs.
- Obtain Boolean functions for each output in terms of the primary inputs.
- Construct the system truth table for all 2^n input combinations to verify behavior.

Design of Combinational Circuits I

Designing combinational circuits is the reverse of analysis: it starts from a specification and ends with a optimized gate-level schematic.

Design of Combinational Circuits II

Key Points

- Step 1: Define the problem and determine the required number of input and output variables.
- Step 2: Assign literal symbols (e.g., A, B, X, Y) to all inputs and outputs.
- Step 3: Construct the truth table defining output values for all input combinations.
- Step 4: Obtain simplified Boolean expressions using Karnaugh Maps or tabular minimization.
- Step 5: Draw the gate schematic conforming to practical constraints (e.g., fan-in/fan-out, NAND/NOR only).

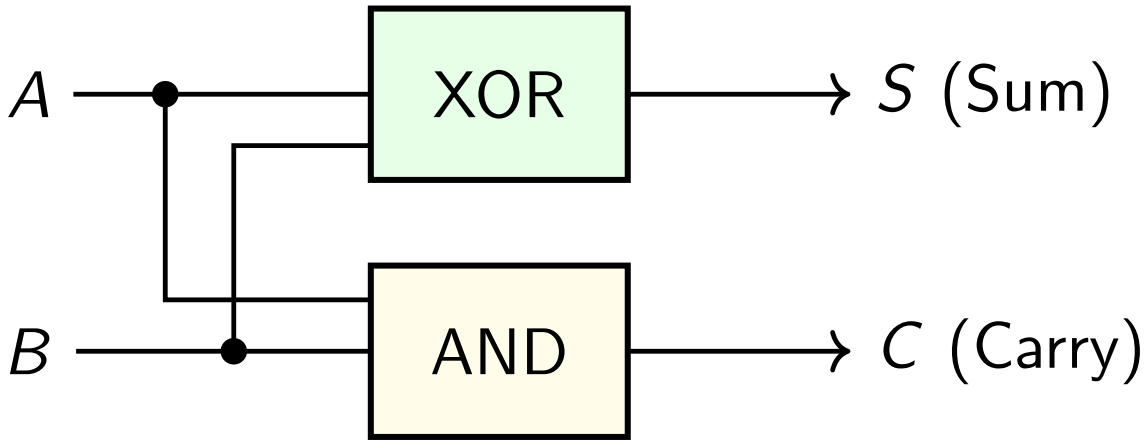
Common Combinational Components I

Standard combinational building blocks are used as Medium Scale Integration (MSI) devices in digital systems.

Key Points

- Arithmetic Blocks: Half Adder, Full Adder, Subtractors, Comparators, and Multipliers.
- Data Routers: Multiplexers (selectors) and Demultiplexers (distributors).
- Code Converters: Binary-to-BCD converters, BCD-to-7-segment decoders.
- Code Transformers: Encoders, Priority Encoders, Decoders, and Demultiplexers.

Arithmetic Logic: Half Adder Design I

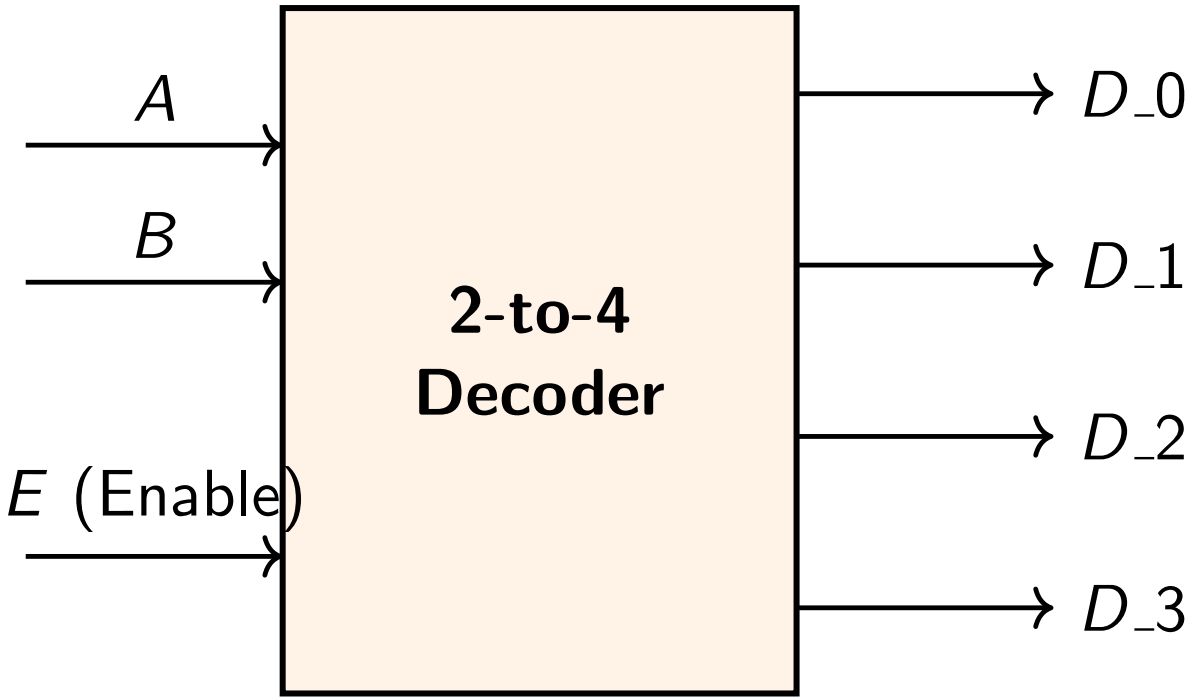


Arithmetic Logic: Half Adder Design II

Key Points

- A Half Adder is an arithmetic combinational circuit that adds two single-bit binary inputs.
- It outputs a Sum (S) bit and a Carry (C) bit.
- The Sum output corresponds to XOR logic: $S = A \oplus B$.
- The Carry output corresponds to AND logic: $C = A \cdot B$.
- Limitation: It cannot accept a Carry input from a previous column addition.

Decoder Circuits I



Decoder Circuits II

Key Points

- A decoder converts an n-bit input binary code into a maximum of 2^n unique output lines.
- Only one output line is active at any given time, depending on the value of inputs A and B.
- The Enable input (E) controls device activation; when E = 0, all output lines remain inactive.
- Decoders are widely used in memory addressing, data demultiplexing, and instruction decoding.

Summary of Combinational Logic I

Combinational circuits form the backbone of processing units, facilitating arithmetic calculation and hardware data routing.

Key Points

- Pure combinational outputs have no memory, depending solely on instantaneous inputs.
- Standard design flow: Specifications -¿ Truth Table -¿ Simplification -¿ Gate Implementation.
- Gate delays can cause temporary output errors, known as static or dynamic hazards.
- Next Lecture: The design of Full Adders, Carry Lookahead Adders, and Subtractors.

Fundamentals of Combinational Logic I

A combinational logic circuit is a class of digital circuits where the outputs at any instant are determined solely by the current combination of inputs.

Fundamentals of Combinational Logic II

Key Points

- Mathematically represented as $Y = f(X)$, where outputs respond to input changes after a brief propagation delay.
- Unlike sequential circuits, combinational circuits contain no memory elements (such as flip-flops or latches) and no feedback loops.
- Commonly implemented using logic gate networks such as AND, OR, NOT, NAND, NOR, and XOR.
- Forms the foundational components of digital processors, memory control units, and input/output interfaces.

Arithmetic vs. Logical Circuits I

Digital hardware functions are broadly divided into arithmetic units for numerical calculation and logical units for data routing and manipulation.

Arithmetic vs. Logical Circuits II

Key Points

- Arithmetic Circuits: Perform mathematical computations on binary representations (e.g., Adders, Subtractors, Comparators, Multipliers).
- Logical Circuits: Execute bitwise operations or perform routing and transformation of data (e.g., Multiplexers, Decoders, Encoders, Demultiplexers).
- Both classes rely on identical gate technologies but differ significantly in how carry or control signals propagate.
- Modern CPUs unify these operations into a single sub-system known as the Arithmetic Logic Unit (ALU).

Methodology of Combinational Design I

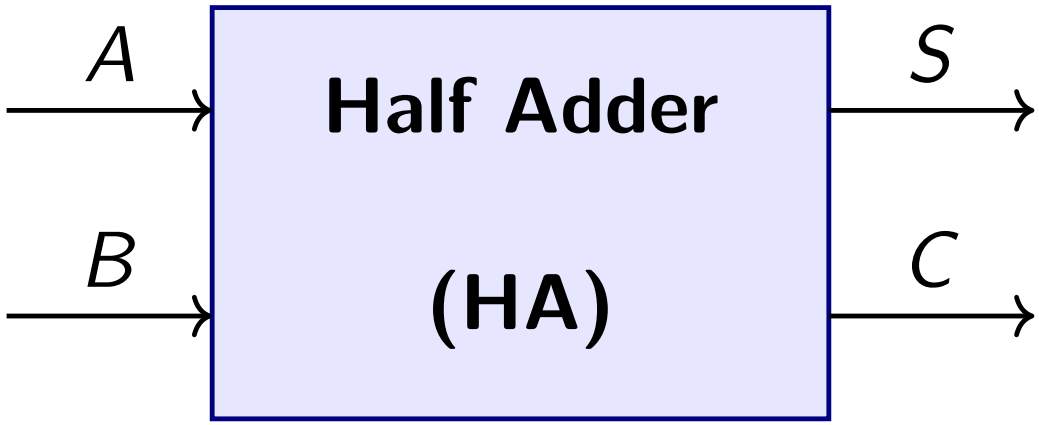
A rigorous, step-by-step engineering procedure ensures that behavioral specifications are correctly translated into optimized gate-level hardware.

Methodology of Combinational Design II

Key Points

- 1. Specification: Define the system requirements, identifying the number of binary input variables and output variables.
- 2. Formulation: Construct the truth table to map all 2^n input states to the desired outputs.
- 3. Optimization: Derive simplified Boolean equations using Karnaugh Maps (K-Maps) or Quine-McCluskey tabular reduction.
- 4. Mapping: Select the target gate library (e.g., NAND-only, standard cells) and generate the logic diagram.
- 5. Verification: Simulate the gate-level design to verify functional correctness and timing characteristics.

Half Adder: Block Diagram I



Half Adder: Block Diagram II

Key Points

- The Half Adder (HA) is the simplest arithmetic circuit designed to compute the sum of two single-bit binary inputs.
- Inputs: A (Augend) and B (Addend).
- Outputs: S (Sum) representing the least significant bit, and C (Carry) representing the most significant bit.
- The block diagram abstractly encapsulates the circuit boundaries and signal directions without showing internal gate topology.

Half Adder: Truth Table and Boolean Formulations I

Truth Table:

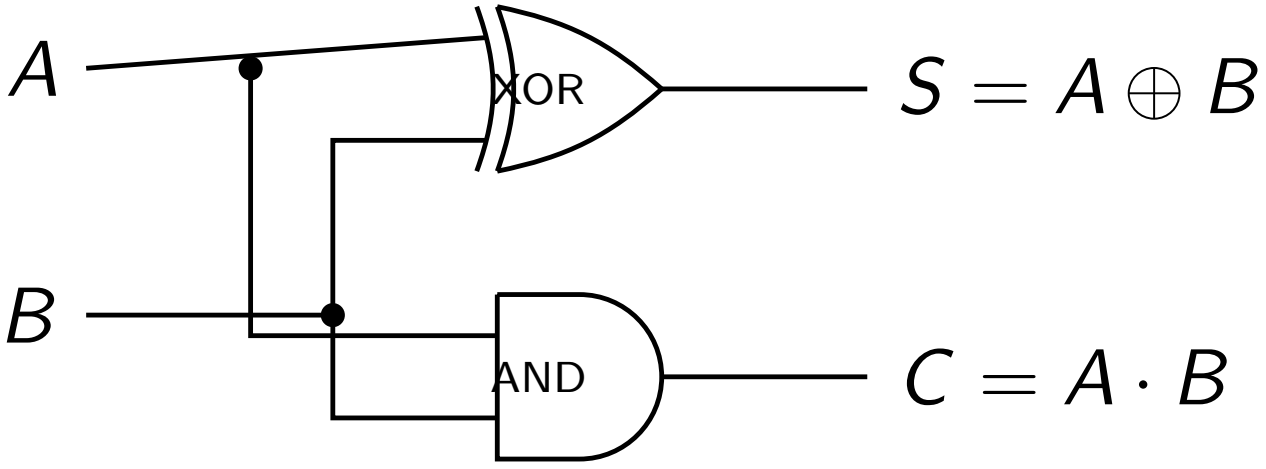
A	B	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adder: Truth Table and Boolean Formulations II

Key Points

- The truth table systematically enumerates all $2^2 = 4$ input combinations and their corresponding outputs.
- Sum (S) column matches the behavior of an Exclusive-OR (XOR) gate, where the output is 1 if and only if the inputs are different.
- Sum expression derived from Sum-of-Products (SOP): $S = \bar{A}B + A\bar{B} = A \oplus B$.
- Carry (C) column matches the behavior of a logical AND gate, where the output is 1 if and only if both inputs are 1.
- Carry expression: $C = A \cdot B$.

Gate-Level Realization of Half Adder I



Gate-Level Realization of Half Adder II

Key Points

- This logic diagram is the physical implementation of the Boolean equations derived from the truth table.
- An XOR gate is used to generate the Sum bit (S), implementing the logic $S = A \oplus B$.
- An AND gate is used to generate the Carry bit (C), implementing the logic $C = A \cdot B$.
- Propagation Delay: The Sum bit is delayed by one XOR gate delay ($t_{pd,XOR}$), while the Carry bit is delayed by one AND gate delay ($t_{pd,AND}$).
- Hardware Cost: The standard implementation requires two basic gates, with a total transistor count depending on technology (e.g., CMOS uses 12 transistors).

Alternative Realization: NAND-Only Implementation I

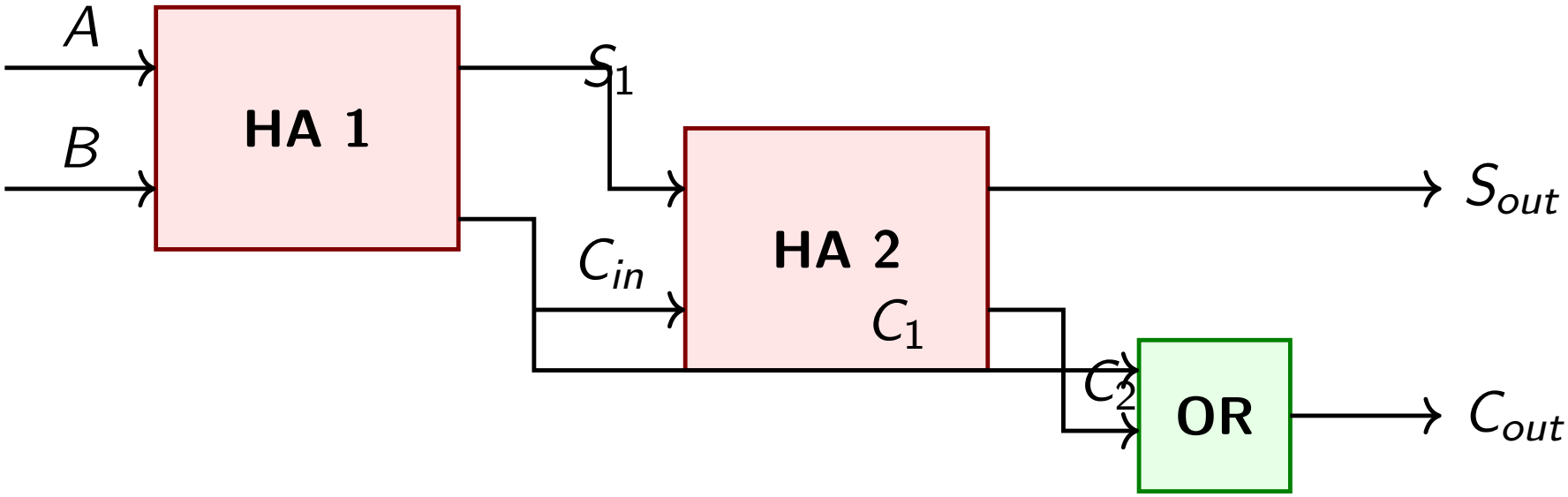
To achieve cost reduction and uniform propagation delays, circuits are often implemented using universal gates (NAND/NOR) instead of heterogeneous gate families.

Alternative Realization: NAND-Only Implementation II

Key Points

- NAND gates are universal gates, meaning any Boolean function can be implemented using only NAND gates.
- A standard XOR operation $S = A \oplus B$ requires 4 NAND gates: $N_1 = \text{NAND}(A, B)$, $N_2 = \text{NAND}(A, N_1)$, $N_3 = \text{NAND}(B, N_1)$, $S = \text{NAND}(N_2, N_3)$.
- The carry bit $C = A \cdot B$ is realized by inverting N_1 using a fifth NAND gate: $C = \text{NAND}(N_1, N_1)$.
- Using NAND-only logic reduces manufacturing complexity in CMOS fabrication because all logic cells are identical.
- Total gate count is 5, but the overall silicon area might be optimized compared to using separate XOR and AND gate cells.

Cascading and the Need for a Full Adder I



Cascading and the Need for a Full Adder II

Key Points

- Limitation: The Half Adder can only add two single-bit binary numbers and has no provision for a Carry-In from a lower-order stage.
- In multi-bit parallel addition (e.g. 8-bit ripple carry adder), each stage except the least significant bit (LSB) must handle three inputs: A_i, B_i, C_i.
- A Full Adder is required for these stages. It can be constructed using two Half Adders and one OR gate, as shown in the diagram.
- Equations for Full Adder: $S_{out} = A \oplus B \oplus C_{in}$, and $C_{out} = A \cdot B + C_{in} \cdot (A \oplus B) = C_1 + C_2$.
- Thus, the Half Adder serves as the fundamental building block for higher-order arithmetic units.

Lecture Summary & Real-World Applications I

Summary of combinational arithmetic design principles and the pivotal role of basic adders in modern computer architectures.

Lecture Summary & Real-World Applications II

Key Points

- Combinational circuits process binary inputs instantaneously (subject to propagation delays) without storing previous states.
- The Half Adder uses one XOR gate and one AND gate to perform basic two-bit binary addition.
- Half Adders are integrated into ALU designs, address generation units (AGUs) in processors, and multiplier arrays.
- Universal gate implementations (NAND/NOR) offer manufacturing and delay optimization pathways.
- The next lecture (Lecture 23) will expand on this foundation by examining the complete Full Adder design and multi-bit Ripple Carry Adders.

Limitations of the Half Adder I

Why we cannot build multi-bit adders using only Half Adders.

Key Points

- A Half Adder accepts only two 1-bit inputs: A and B.
- It produces a Sum (S) and a Carry (C) output.
- In multi-bit addition, every column (except the LSB) must add three bits: A_i, B_i, and the carry-in C_i from the previous stage.
- Since the Half Adder lacks a third input for the carry-in, it cannot be cascaded to perform addition on multi-bit operands.

The Full Adder: Definition and Truth Table I

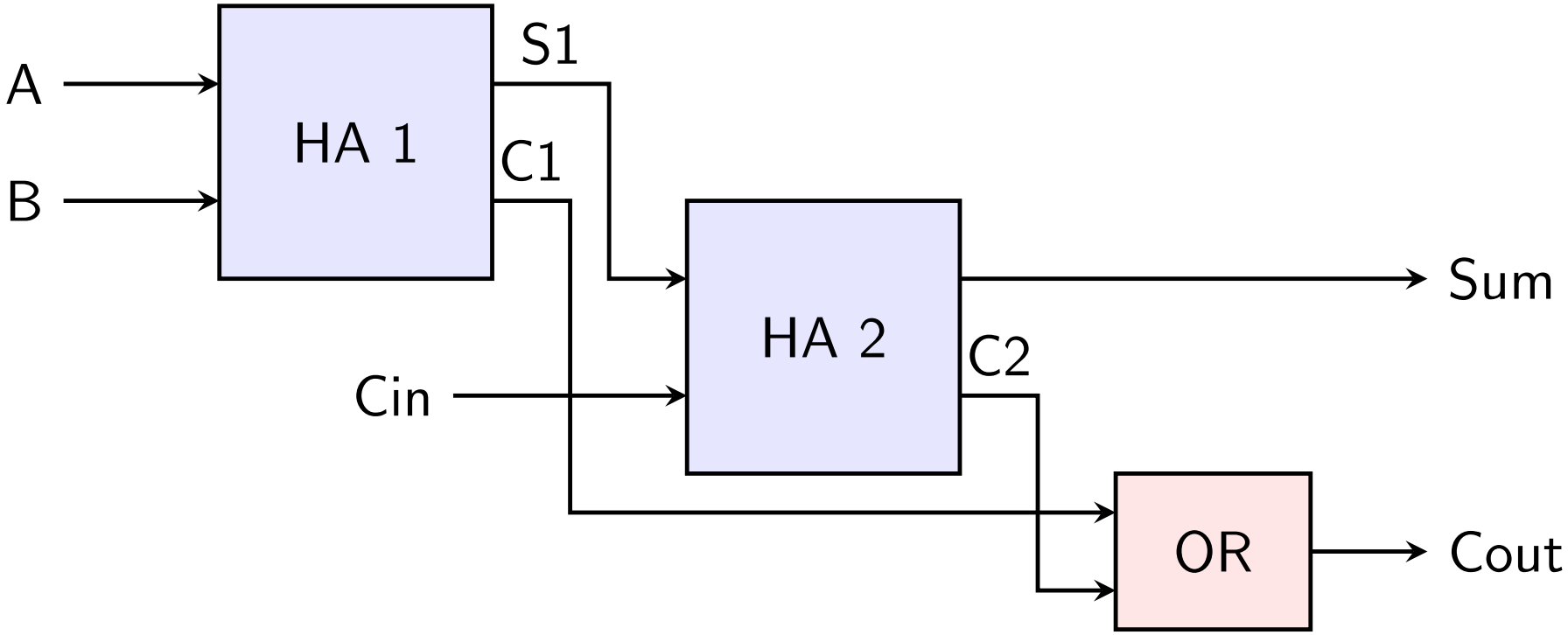
A combinational circuit that performs the arithmetic sum of three input bits.

The Full Adder: Definition and Truth Table II

Key Points

- Inputs: A (Augend), B (Addend), and Cin (Carry-in).
- Outputs: Sum (S) and Cout (Carry-out).
- Truth table behavior: Sum is high (1) when an odd number of inputs are high (1, 3).
- Cout is high (1) when two or more inputs are high (2, 3).
- Boolean equations: Sum = A $\oplus B \oplus Cin$ and Cout = $AB + BCin + ACin$.

Full Adder: Two Half Adders Block Diagram I



Full Adder: Two Half Adders Block Diagram II

Key Points

- A Full Adder can be constructed hierarchically using two Half Adders and a single 2-input OR gate.
- The first Half Adder computes the intermediate sum S1 = A $\oplus$ *Band* *carry* *C*1 = *AB*.
- The second Half Adder computes the final Sum by adding S1 and Cin: Sum = S1 $\oplus$ *Cin* = *A* $\oplus$ *B* $\oplus$ *Cin*.
- The final Carry-out Cout is generated if either Half Adder produces a carry: Cout = C1 + C2 = AB + Cin(A $\oplus$ *B*).

Mathematical Proof: Carry Equivalence I

Proof that AB + Cin(A $\oplus B$)*isequivalenttoAB* + *BCin* + *ACin*.

Mathematical Proof: Carry Equivalence II

Key Points

- Start with the carry equation from the block diagram: $\text{Cout} = \text{AB} + \text{Cin}(\text{A} \oplus B)$.
- Expand the XOR term: $\text{Cout} = \text{AB} + \text{Cin}(\text{A} + \text{B}) = \text{AB} + \text{ACin} + \text{BCin}$.
- Use Consensus Theorem or Boolean expansion: $\text{AB} = \text{AB}(1) = \text{AB}(1 + \text{Cin}) = \text{AB} + \text{ABCin}$.
- Combine terms: $\text{AB} + \text{ACin} = \text{A}(\text{B} + \text{Cin}) = \text{A}(\text{B} + \text{Cin}) = \text{AB} + \text{ACin}$.
- Similarly: $\text{AB} + \text{BCin} = \text{B}(\text{A} + \text{Cin}) = \text{B}(\text{A} + \text{Cin}) = \text{AB} + \text{BCin}$.
- Thus, $\text{Cout} = \text{AB} + \text{BCin} + \text{ACin}$, proving the logic equivalence of both implementations.

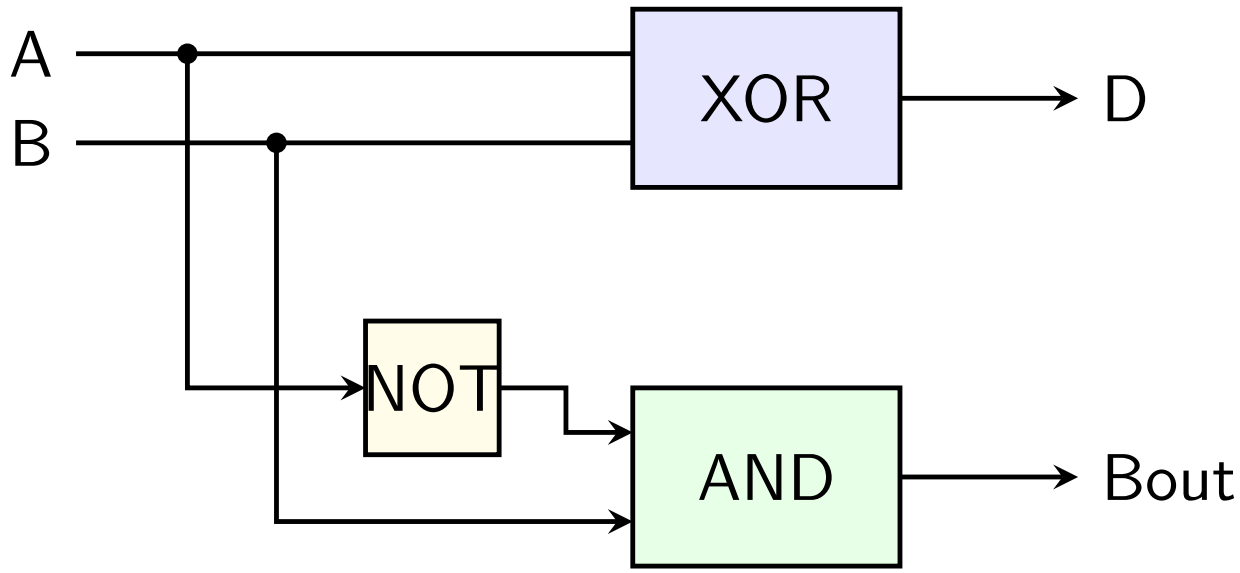
Introduction to Binary Subtractors I

Subtractor circuits perform binary subtraction by finding the difference between two bits.

Key Points

- Subtraction is non-commutative: A - B is not equal to B - A. We define A as the Minuend and B as the Subtrahend.
- The output signals are Difference (D) and Borrow-out (Bout).
- Difference (D) represents the arithmetic result: A - B.
- Borrow-out (Bout) is a signal indicating whether the minuend A had to borrow a '1' from the next higher-order bit stage to complete the subtraction.

The Half Subtractor Logic Circuit I



The Half Subtractor Logic Circuit II

Key Points

- A Half Subtractor performs subtraction on two 1-bit inputs: A (Minuend) and B (Subtrahend).
- It cannot handle a borrow-in from a lower-order stage, making it analogous to a Half Adder.
- The Difference output uses an XOR gate: $D = A \oplus B$. *ThisisidenticaltotheHalfAdderSumlogic*.
- The Borrow-out output requires an active-low input on the Minuend A: Bout = B, implemented using a NOT gate and an AND gate.

Half Subtractor Truth Table and Logic Equations I

Derivation of the algebraic expressions from the truth table.

Key Points

- Case 1: 0 - 0 -¿ Difference = 0, Borrow = 0.
- Case 2: 0 - 1 -¿ must borrow '1' from next stage, making it 2 - 1 = 1 -¿ Difference = 1, Borrow = 1.
- Case 3: 1 - 0 -¿ Difference = 1, Borrow = 0.
- Case 4: 1 - 1 -¿ Difference = 0, Borrow = 0.
- Difference equation from minterms: $D = B + A = A \oplus B$.
- Borrow equation from minterms: Bout = B.

Multi-bit Operations and Carry Propagation I

How full adders are cascaded to form multi-bit adders, and the limitations of propagation delay.

Multi-bit Operations and Carry Propagation II

Key Points

- A Ripple Carry Adder (RCA) is constructed by chaining N Full Adders in series.
- The carry-out of each stage Cout,i becomes the carry-in Cin,i+1 of the next stage.
- Propagation Delay: The sum and carry outputs of the N-th stage cannot settle until the carry has rippled through all preceding N-1 stages.
- Total delay is dominated by the carry propagation path: $t_total \approx (N-1)t_carry + t_sum$.
- Advanced designs like Carry Lookahead Adders (CLA) are used to overcome this delay by calculating carries in parallel.

Summary and Comparison I

Key comparisons between adder and subtractor circuits.

Summary and Comparison II

Key Points

- Half Adder: Adds 2 bits. S = A $\oplus B$, $C = AB$. *Gates* : $1XOR, 1AND$.
- Full Adder: Adds 3 bits. S = A $\oplus B \oplus Cin$, $Cout = AB + Cin(A \oplus B)$. *Gates* : $2XORs, 2ANDs, 1OR$.
- Half Subtractor: Subtracts 2 bits. D = A $\oplus B$, $Bout = \bar{A}B$. *Gates* : $1XOR, 1NOT, 1AND$.
- Full Subtractor: Subtracts 3 bits. D = A $\oplus B \oplus Bin$, $Bout = \bar{A}B + Bin(\overline{A \oplus B})$.

Full Subtractor: Mathematical Analysis and Truth Table I

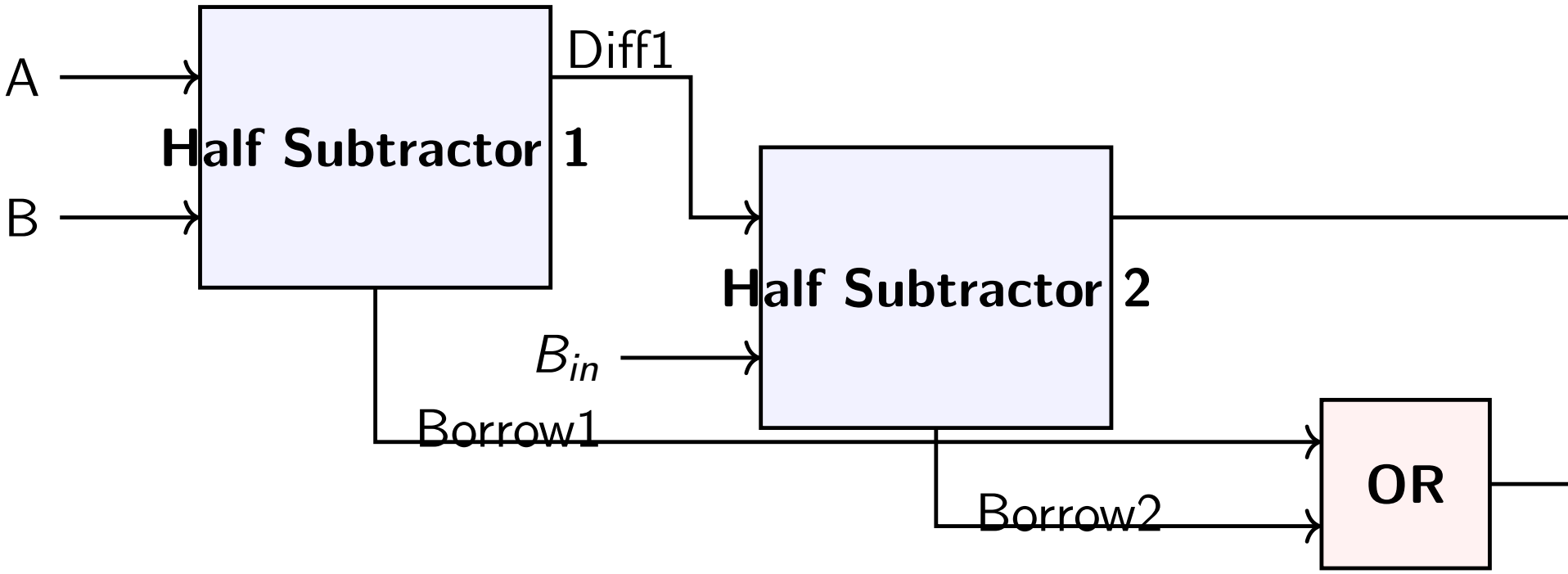
A Full Subtractor performs subtraction on three single-bit inputs: the minuend (A), the subtrahend (B), and the borrow-in (Bin) from a previous lower-order stage.

Full Subtractor: Mathematical Analysis and Truth Table II

Key Points

- The circuit generates two outputs: Difference (D) and Borrow-Out (Bout).
- The Boolean expression for Difference is D = A $\oplus B \oplus$ *Bin*, *whichmatchestheexpressionfortheSuminaFullAdder*.
- The Boolean expression for Borrow-Out is Bout = A'B + A'Bin + BBin = A'(B $\oplus$*Bin*) + *BBin*, *whichisactivewhentheminuendAissmallerthanthecombina*
- The truth table consists of 8 rows: for input combinations (0,0,0) to (1,1,1), producing output pairs (0,0), (1,1), (1,1), (0,1), (1,0), (0,0), (0,0), (1,1) for (D, Bout).

Full Subtractor Block Implementation I



Full Subtractor Block Implementation II

Key Points

- A Full Subtractor can be constructed using two Half Subtractors and a single OR gate.
- The first Half Subtractor subtracts B from A to yield Difference-1 (Diff1 = A $\oplus B) and Borrow - 1 (Borrow1 = A'B)$.
- The second Half Subtractor subtracts Bin from Diff1 to yield the final Difference (D = Diff1 $\oplus Bin) and Borrow - 2 (Borrow2 = Diff1' \cdot Bin)$.
- The overall Borrow-Out (Bout) is the OR combination of Borrow-1 and Borrow-2: Bout = A'B + (A $\oplus B)' \cdot Bin$.

Introduction to Data Transmission Combinational Circuits I

Data transmission combinational circuits route, direct, compress, or decompress binary data packets over a communication bus or system network.

Introduction to Data Transmission Combinational Circuits II

Key Points

- These circuits do not store information (being combinational) but facilitate efficient channel sharing and routing.
- Key categories include Encoders, Decoders, Multiplexers (Data Selectors), and Demultiplexers (Data Distributors).
- Encoders convert active-line input state signals into a more compact encoded binary output format.
- Decoders accept binary inputs and activate a corresponding output line, reverting the encoded signals back to single-active line formats.

Basic Function of Encoders: Theoretical Analysis I

An encoder is a combinational logic circuit that performs the inverse operation of a decoder.

Key Points

- It features 2^n input lines and n output lines, generating the n-bit binary code representing the active input line.
- Under the basic encoder model, only one input line is active at any given time (one-hot state format).
- If multiple inputs become active simultaneously, a standard encoder produces an invalid, garbled output code.
- To resolve conflicting active lines, Priority Encoders are designed to prioritize the highest-indexed active line.

4-to-2 Binary Encoder: Logic Design I

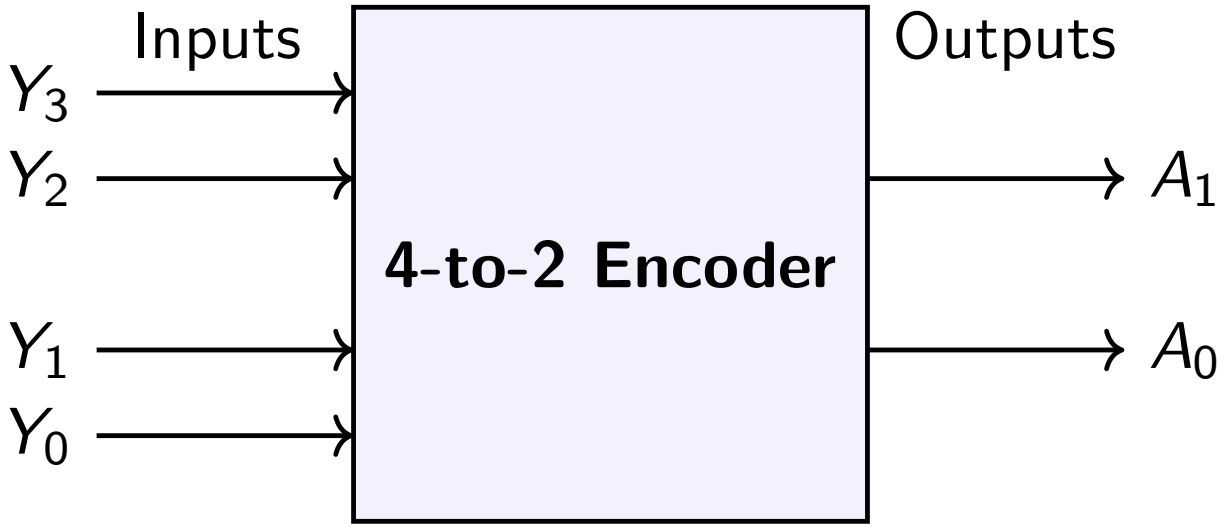
A 4-to-2 encoder takes four input lines (Y0, Y1, Y2, Y3) and maps them to a two-bit binary code (A1, A0).

4-to-2 Binary Encoder: Logic Design II

Key Points

- The inputs are mutually exclusive: Y0 = 1 yields A1A0 = 00; Y1 = 1 yields A1A0 = 01; Y2 = 1 yields A1A0 = 10; Y3 = 1 yields A1A0 = 11.
- Boolean equations are derived from OR combinations of active states: A0 = Y1 + Y3 (active for odd-numbered input channels).
- Boolean equation for output A1: A1 = Y2 + Y3 (active for the upper half of input channels).
- In simple logic gates, this encoder can be constructed using just two 2-input OR gates.

4-to-2 Encoder Block Diagram I



4-to-2 Encoder Block Diagram II

Key Points

- The block diagram illustrates four input lines representing signals Y0, Y1, Y2, Y3.
- Two binary output lines correspond to the bits A1 and A0.
- This configuration reduces a 4-wire physical transmission bus into a 2-wire encoded digital bus.
- It acts as a fundamental component in system control panel interfaces and data collection nodes.

Basic Function of Decoders: Theoretical Analysis I

A decoder detects the presence of a specific n-bit binary code on its inputs and activates the single unique output line matching that combination.

Basic Function of Decoders: Theoretical Analysis II

Key Points

- It contains n input lines and up to 2^n output lines; only the output line corresponding to the input minterm is active.
- Decoders are essentially minterm generators: each output represents one unique product term of the inputs.
- Most commercial decoders incorporate an Enable (E or EN) input to control the active state of the whole chip.
- If the Enable input is disabled, all outputs remain inactive (0 for active-high, 1 for active-low) regardless of the address inputs.

2-to-4 Decoder: Logic Design and Expressions I

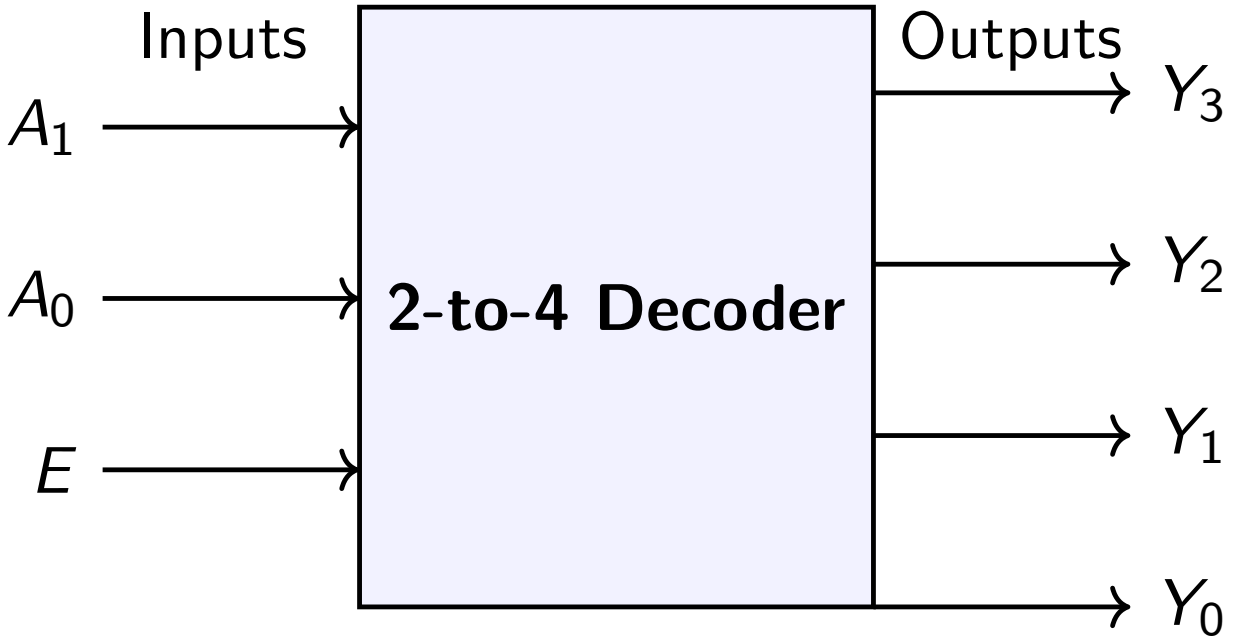
A 2-to-4 decoder maps two inputs (A1, A0) to four outputs (Y0, Y1, Y2, Y3) with an active-high Enable signal (E).

2-to-4 Decoder: Logic Design and Expressions II

Key Points

- The output equations are direct representations of the four minterms of A1 and A0, gated by E.
- Boolean expression for output Y0: $Y0 = E \cdot A1' \cdot A0'(activewheninputsare00)$.
- Boolean expression for output Y1: $Y1 = E \cdot A1' \cdot A0(activewheninputsare01)$.
- Boolean expression for output Y2: $Y2 = E \cdot A1 \cdot A0'(activewheninputsare10), and Y3 = E \cdot A1 \cdot A0(activewheninputsare11)$.

2-to-4 Decoder Block Diagram I



2-to-4 Decoder Block Diagram II

Key Points

- The block diagram illustrates two select inputs (A1, A0), one Enable line (E), and four outputs (Y0, Y1, Y2, Y3).
- This decoder acts as a 1-of-4 selector block, widely applied in peripheral device mapping.
- Cascading enable inputs allows multiple decoders to be combined (e.g., configuring two 2-to-4 decoders into a 3-to-8 decoder).
- Decoders are the standard building blocks for memory block select logic (RAM chip-select pins).

What is a Multiplexer? I

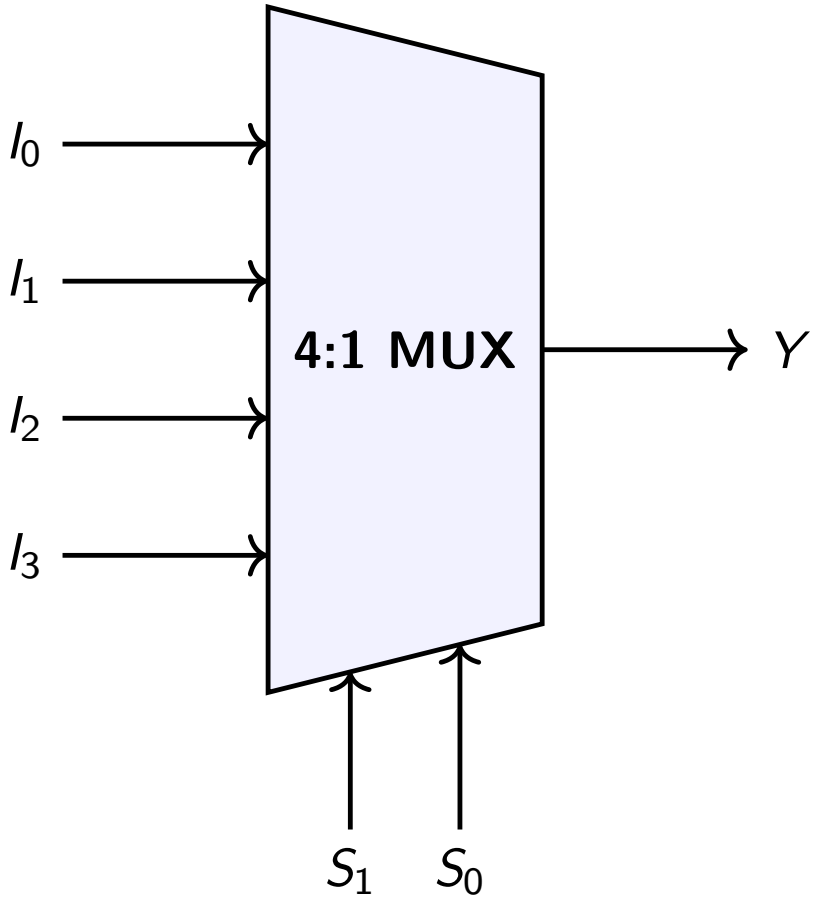
A multiplexer (commonly abbreviated as MUX) is a digital combinational logic circuit that selects one of many analog or digital input sources and routes it to a single output line. It acts as a digitally controlled multi-position switch.

What is a Multiplexer? II

Key Points

- Enables data routing from multiple sources to a single destination link.
- Consists of 2^n inputs, n selection lines, and exactly 1 output line.
- The control input binary value determines which input is connected to the output.
- Crucial for optimizing buses, communication paths, and processor data-paths.

Block Diagram of a 4-to-1 MUX I



Block Diagram of a 4-to-1 MUX II

Key Points

- Inputs I_0, I_1, I_2, I_3 represent the data sources.
- Outputs Y is the single selected destination line.
- Selection lines S_1 (MSB) and S_0 (LSB) control the switching logic.
- Total lines: 4 Data Inputs, 2 Select Inputs, 1 Output.

Truth Table of 4-to-1 MUX I

The operational behavior of the 4-to-1 MUX is formally defined by its Truth Table. By configuring the selection lines S_1 and S_0, we choose which input channel is mapped to Y.

Key Points

- When $S_1S_0 = 00$, $Y = I_0$ (Input channel 0 selected).
- When $S_1S_0 = 01$, $Y = I_1$ (Input channel 1 selected).
- When $S_1S_0 = 10$, $Y = I_2$ (Input channel 2 selected).
- When $S_1S_0 = 11$, $Y = I_3$ (Input channel 3 selected).
- Selection inputs are active-high by default.

Boolean Equation of 4-to-1 MUX I

The logic expression for the output Y of a 4-to-1 multiplexer can be derived using the Sum-of-Products (SOP) formulation from the truth table.

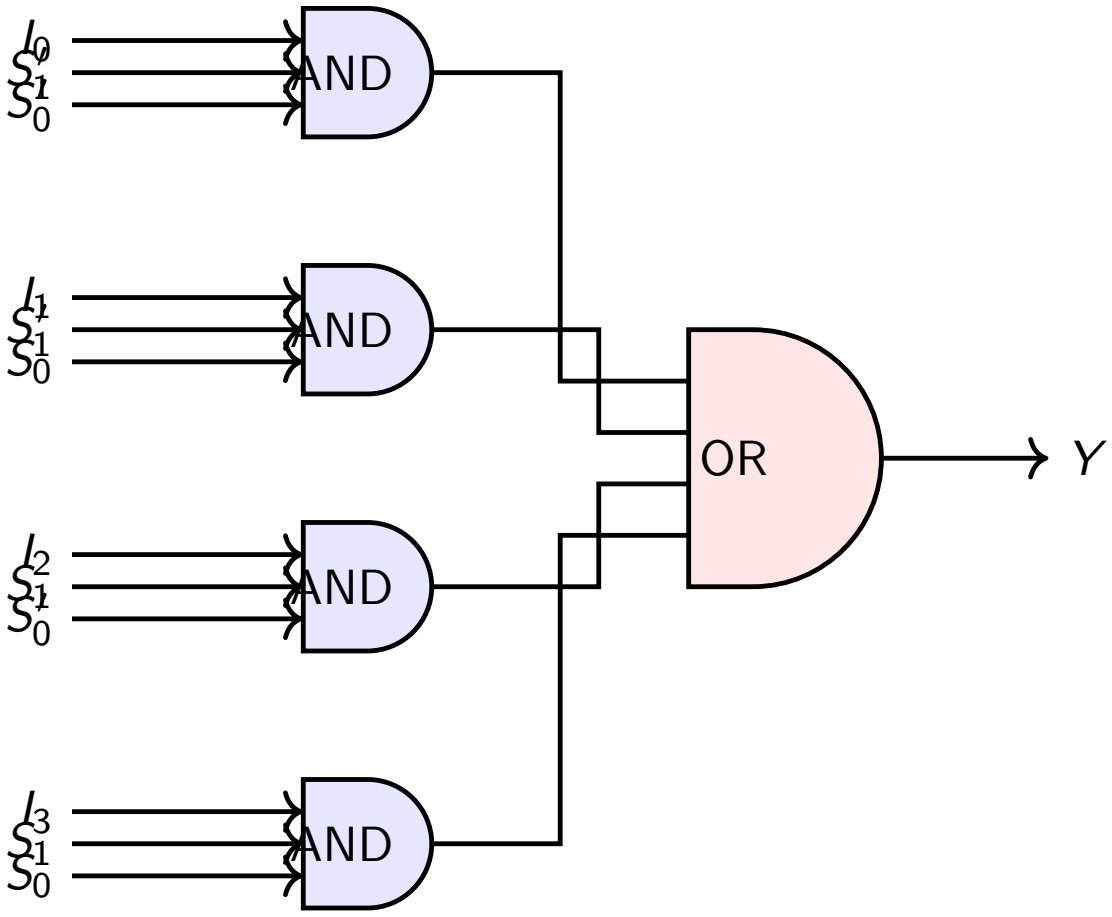
$$Y = S_1' S_0' I_0 + S_1' S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$$

Boolean Equation of 4-to-1 MUX II

Key Points

- Each product term represents one selection configuration.
- Only one product term can be logic high at any given time.
- The control variables act as enable switches for each data channel.
- If a selection code is active, the corresponding data line is directly routed to the output.

Gate-Level Logic Diagram I



Gate-Level Logic Diagram II

Key Points

- Implemented using four 3-input AND gates and one 4-input OR gate.
- Inverters (NOT gates) generate the complemented selection signals S_1' and S_0'.
- Each AND gate is enabled by a specific combination of selection inputs.
- The OR gate combines the outputs of all AND gates to produce the single final output Y.

Propagation Delay in a 4-to-1 MUX I

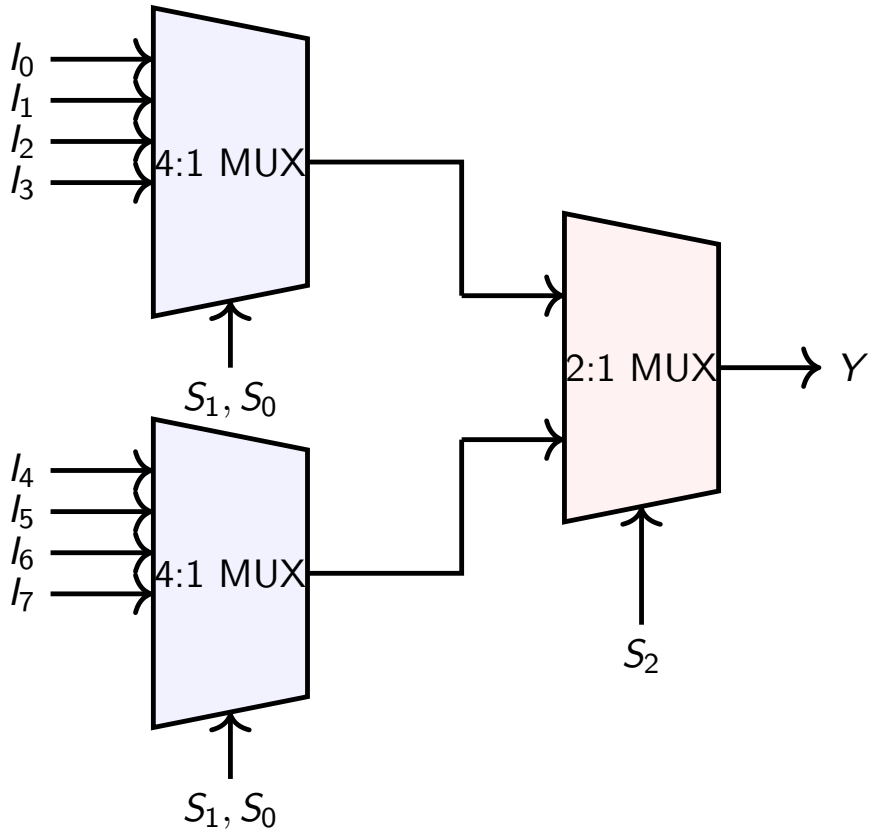
Propagation delay (t_{pd}) defines the time required for a signal transition at an input to propagate to the output.

Propagation Delay in a 4-to-1 MUX II

Key Points

- Data-to-Output ($I_x \rightarrow Y$): Passes through 2 gate levels (AND-OR). Typically shorter delay.
- Select-to-Output ($S_x \rightarrow Y$): Passes through 3 gate levels (NOT-AND-OR) if inversion is needed. Typically represents the critical path.
- Unequal delays can lead to transient 'glitches' or hazard states during select transition.
- Using high-speed logic families (like 74AHC) minimizes these timing disparities.

Cascading: Building 8-to-1 MUX from 4-to-1 MUXes I



Cascading: Building 8-to-1 MUX from 4-to-1 MUXes II

Key Points

- An 8-to-1 MUX requires selecting from 8 inputs, requiring 3 select lines (S_2, S_1, S_0).
- Two 4-to-1 MUXes decode the lower select lines (S_1, S_0) in parallel.
- A final 2-to-1 MUX uses the MSB select line (S_2) to choose between the outputs of the two 4-to-1 MUXes.
- This hierarchical tree structure scales modularly to construct 16-to-1 MUXes or larger.

Standard MUX IC: The 74153 I

Commercial integrated circuits package multiple multiplexers into a single chip. A classic TTL standard is the 74LS153.

Key Points

- Contains two independent 4-to-1 multiplexers in a single 16-pin Dual In-line Package (DIP).
- Features common Select Inputs (A, B) shared by both MUX circuits.
- Includes individual active-low Strobe/Enable inputs $(1G', 2G')$ for disabling or enabling each MUX independently.
- If Strobe is high, the output is forced low regardless of select and data inputs.

Applications of Multiplexers I

Multiplexers are critical components in both simple and highly complex digital systems, including CPU design and communications.

Applications of Multiplexers II

Key Points

- Data Routing: Directing register outputs to a shared Arithmetic Logic Unit (ALU) bus.
- Boolean Function Generator: Implementing logic gates and arbitrary boolean equations directly without logic gate reduction.
- Parallel-to-Serial Data Converter: Taking a byte/word input and streaming it bit-by-bit over a serial channel.
- Time-Division Multiplexing (TDM): Sharing a single communication medium among multiple independent users.

The Role of Truth Tables in Combinational Logic I

A truth table defines the operational behavior of a combinational logic circuit by listing all possible combinations of inputs and their corresponding output states.

The Role of Truth Tables in Combinational Logic II

Key Points

- For n inputs, the truth table contains exactly 2^n unique rows representing binary combinations from 0 to 2^n - 1.
- Truth tables are independent of physical implementation, serving as the mathematical specification of the circuit's boolean functions.
- They provide the starting point for minimization techniques like Karnaugh Maps (K-maps) and the Quine-McCluskey method.
- Outputs are explicitly defined as 0, 1, or Don't Care (X) for each input vector combination.

Representing Circuits with Logic Diagrams I

A logic diagram is a schematic representation of a digital circuit using standardized symbols for logic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) and lines representing electrical connections.

Representing Circuits with Logic Diagrams II

Key Points

- Translates abstract Boolean algebraic equations into physical hardware topologies.
- Shows the signal propagation path from inputs (typically on the left) to outputs (typically on the right).
- Helps engineers analyze propagation delay, fan-in/fan-out, and physical routing constraints.
- Provides a visual blueprint for structural hardware description languages (HDLs) and schematic capture tools.

Introduction to Demultiplexers (DEMUX) I

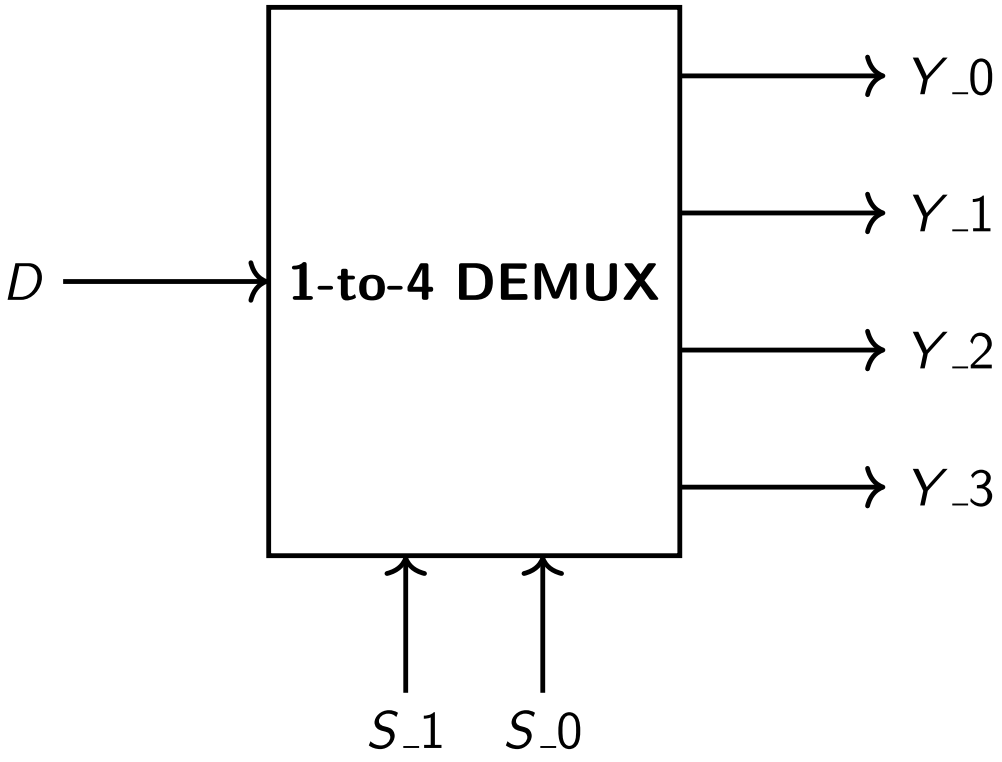
A Demultiplexer (DEMUX) is a combinational logic circuit that takes a single input signal and routes it to one of several outputs based on the state of control select lines.

Introduction to Demultiplexers (DEMUX) II

Key Points

- Known as a data distributor or one-to-many spatial switch.
- Operates as the functional inverse of a Multiplexer (MUX).
- Has 1 data input (D), 's' select inputs, and 2^s outputs (Y_0 to Y_2^s-1).
- The select inputs determine which specific output port will mirror the logic state of the input line, while all other outputs remain inactive.

1-to-4 DEMUX Block Diagram I



1-to-4 DEMUX Block Diagram II

Key Points

- The block diagram shows one data input D, two select inputs S_1 and S_0, and four outputs Y_0, Y_1, Y_2, Y_3.
- The selection code S_1 S_0 forms a 2-bit binary number indicating the active output index.
- For example, when S_1 $S\_0 = 10$, the output Y_2 is connected to the input D, while Y_0, Y_1, and Y_3 are held at logic level 0 (active-high).

Truth Table of a 1-to-4 DEMUX I

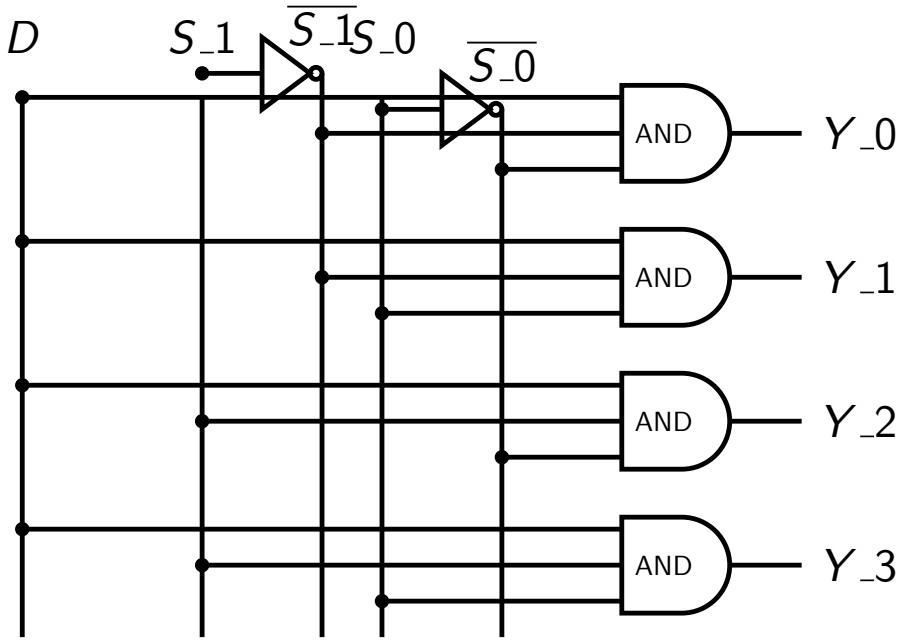
Select Inputs		Input	Outputs			
S_1	S_0	D	Y_0	Y_1	Y_2	Y_3
0	0	D	D	0	0	0
0	1	D	0	D	0	0
1	0	D	0	0	D	0
1	1	D	0	0	0	D

Truth Table of a 1-to-4 DEMUX II

Key Points

- The data input D can be either a 0 or 1; the selected output follows D.
- Non-selected outputs are forced to an inactive state (0 for active-high outputs).
- Boolean expressions for the outputs: $Y\_0 = \overline{S\_1} \cdot \overline{S\_0} \cdot D$,Y_1=$\overline{S_1} \cdot$S_0$\cdot D$,Y_2=S_1$\cdot \overline{S_0} \cdot D$,Y_3=S_1$\cdot$S_0$\cdot D$.
- The circuit effectively acts as a decoder with D serving as the enable line.

Logic Diagram of a 1-to-4 DEMUX I



Logic Diagram of a 1-to-4 DEMUX II

Key Points

- The logic diagram shows the implementation using two inverters (NOT gates) and four 3-input AND gates.
- Each AND gate generates a minterm of the select variables combined with the data input D.
- Since only one AND gate can be enabled at any given time (depending on S_1 and S_0), D is routed to exactly one output.
- This structural architecture is scalable for larger demultiplexers like 1-to-8 or 1-to-16 by adding more select lines.

Practical Applications of DEMUX I

Demultiplexers are crucial in digital communication, computer architecture, and memory systems where a single channel needs to be shared among multiple destinations.

Practical Applications of DEMUX II

Key Points

- Data Routing: Directing audio/video streams from a single receiver source to multiple speaker/display destinations.
- Address Decoding in Memories: Selecting a specific memory bank or row in SRAM/DRAM based on address bus states.
- Serial-to-Parallel Conversion: Reconstructing parallel data bytes from a serial bit stream (used in combination with synchronous counters).
- Bus Demultiplexing: Sharing a common microprocessor data bus with multiple peripheral devices (e.g., input/output ports).

Multiplexer vs. Demultiplexer Comparison I

— Feature — Multiplexer (MUX) — Demultiplexer (DEMUX) — ——————— — Type of Operation — Many-to-One (Data Selector) — One-to-Many (Data Distributor) — — Inputs — 2^n Data inputs, n Select lines — 1 Data input, n Select lines — — Outputs — 1 Data output — 2^n Data outputs — — Logic Structure — Sum-of-Products (SOP) — AND gates driven by decoders — — Equation Form — $Y = \sum(\text{Minterm\_i} \cdot \text{D\_i}) | Y\_i = \text{Minterm\_i} \cdot D |$

Multiplexer vs. Demultiplexer Comparison II

Key Points

- Both circuits utilize select lines to control data routing, acting as electronic switches.
- A MUX consolidates multiple channels into a single transmission medium, while a DEMUX reconstructs the original channels at the receiving end.
- They are frequently paired in telecommunication links to enable Time-Division Multiplexing (TDM).

Cascading for Larger Demultiplexers I

Standard 1-to-4 DEMUX units can be cascaded to construct larger demultiplexing networks, such as a 1-to-8 or 1-to-16 DEMUX, without requiring custom high-input gates.

Cascading for Larger Demultiplexers II

Key Points

- To build a 1-to-8 DEMUX, two 1-to-4 DEMUX units are combined with an additional select line (acting as an enable control).
- The most significant select bit (S_2) is routed to the enable inputs, activating only one of the two 1-to-4 DEMUX chips at any time.
- Cascading provides a modular approach to circuit design, minimizing design time and propagation delay analysis.
- This modular expansion technique is widely used in Field Programmable Gate Array (FPGA) logic blocks.

Levels of Design Abstraction I

Digital systems are represented at different levels of abstraction to manage complexity. Each representation serves a unique purpose during specification, design, simulation, and implementation.

Key Points

- Truth Table: Behavioral specification detailing output response for all possible input combinations.
- Block Diagram: Structural/functional abstraction showing system modules, ports, and data paths without gate-level details.
- Logic Diagram: Gate-level schematic mapping physical logic gates (AND, OR, XOR, NOT) and their exact interconnections.

The Truth Table I

A Truth Table is the most basic behavioral representation of a combinational circuit, mapping binary inputs to binary outputs.

Key Points

- Defines the complete input-output relationship mathematically.
- For N inputs, the table contains exactly 2^N rows representing all possible input states.
- Independent of physical realization, layout, or gate selection.
- Serves as the starting point for minimization techniques (Boolean algebra, Karnaugh Maps, Quine-McCluskey).

The Block Diagram I

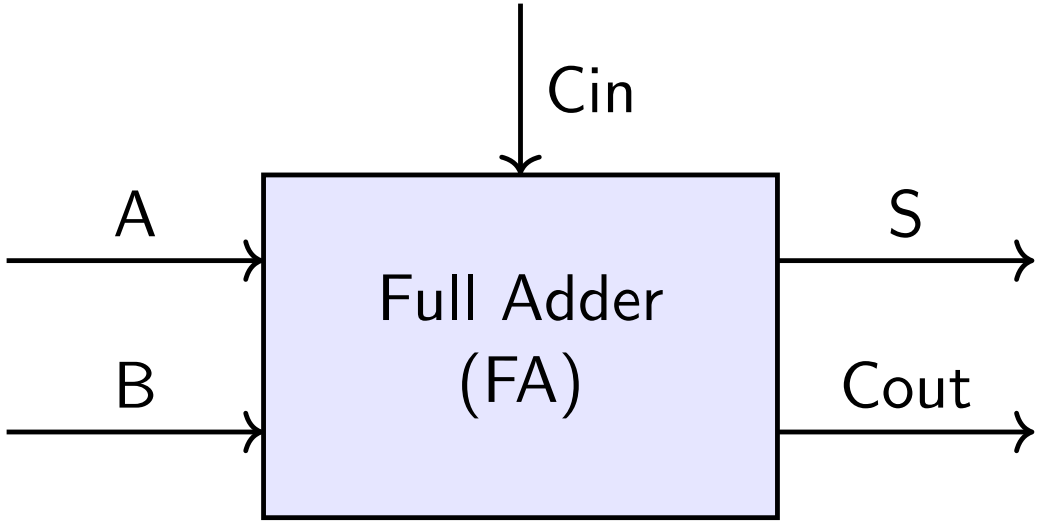
A Block Diagram represents a system at the highest level of structural abstraction. It defines the system boundaries and interface ports.

The Block Diagram II

Key Points

- Uses blocks (typically rectangles) to represent complex functional units (e.g., ALU, Multiplexer, Adder).
- Specifies input and output signals, indicating their direction and width (single wire vs. multi-bit bus).
- Hides internal implementation details to allow designers to focus on system integration.
- Critical for modular design, dividing a large system into manageable, testable sub-blocks.

Visualizing a Block Diagram: Full Adder I



Visualizing a Block Diagram: Full Adder II

Key Points

- The block diagram treats the Full Adder as a 'black box'.
- Inputs (A, B, Cin) enter from the left and top; outputs (S, Cout) exit from the right.
- No internal gates are shown; only interface signals and functional labels are defined.
- Facilitates cascade configurations (e.g., connecting multiple FAs to form a Ripple Carry Adder).

The Logic Diagram I

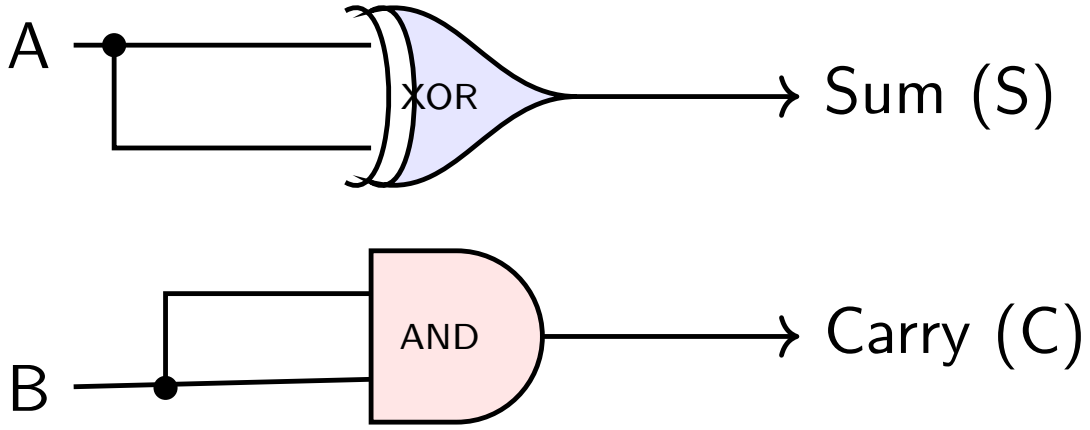
A Logic Diagram translates the behavioral specification (Truth Table) or functional block into an interconnected network of physical logic gates.

The Logic Diagram II

Key Points

- Shows the exact gate types (AND, OR, NAND, NOR, XOR, NOT) used to implement the circuit.
- Represents physical wiring and signal routing between components.
- Directly correlates with hardware implementation (transistors, standard cells, or FPGA Look-Up Tables).
- Enables analysis of propagation delays, critical path timing, and potential hazards/glitches.

Visualizing a Logic Diagram: Half Adder I



Visualizing a Logic Diagram: Half Adder II

Key Points

- Shows the exact implementation of a Half Adder using one XOR gate and one AND gate.
- Line intersections with dots indicate electrical connections (nodes).
- Illustrates how parallel logic paths generate multiple outputs simultaneously.
- Allows verification of Boolean expressions: $S = A \oplus B$ and $C = A \cdot B$.

Translation Methodology I

Designing digital systems requires systematic translation from behavior (Truth Table) to structure (Logic Diagram) and abstraction (Block Diagram).

Translation Methodology II

Key Points

- Step 1: Define system requirements and construct the Truth Table representing behavioral requirements.
- Step 2: Derive Boolean expressions for each output using SOP (Sum of Products) or POS (Product of Sums).
- Step 3: Simplify expressions using Karnaugh Maps or algebraic manipulation to minimize gate count.
- Step 4: Draw the Logic Diagram based on the minimized expressions, then package the circuit into a Block Diagram for modular reuse.

Comparison of Representations I

Each representation highlights different characteristics of a digital system and is used during different stages of the design cycle.

Key Points

- Truth Table: Best for verification, unambiguous behavioral reference, but scales poorly (2^N growth).
- Block Diagram: Best for system-level design, interconnect visualization, and functional partitioning.
- Logic Diagram: Best for hardware debugging, delay path estimation, and physical netlist generation.

Summary & Key Takeaways I

Understanding block diagrams, logic diagrams, and truth tables is fundamental to managing complexity in Computer Engineering.

Key Points

- Digital design relies on moving between behavioral, functional, and structural domains.
- Block Diagrams define the boundaries and interfaces (I/O) of system modules.
- Logic Diagrams specify the gate-level implementation details of these modules.
- Truth Tables provide the absolute mathematical behavioral specification of the circuit.

Review: Combinational Logic Circuits I

A combinational circuit's output at any time is determined solely by its current inputs.

Review: Combinational Logic Circuits II

Key Points

- No memory elements are present in the circuit structure to store state
- No feedback paths exist from outputs back to inputs
- Mathematically defined as $Y = f(X)$, where X is the input vector and Y is the output vector
- Examples: Adders, subtractors, multiplexers, decoders, and encoders
- Propagation delay defines the time taken for outputs to settle after input transition

Defining Sequential Logic Circuits I

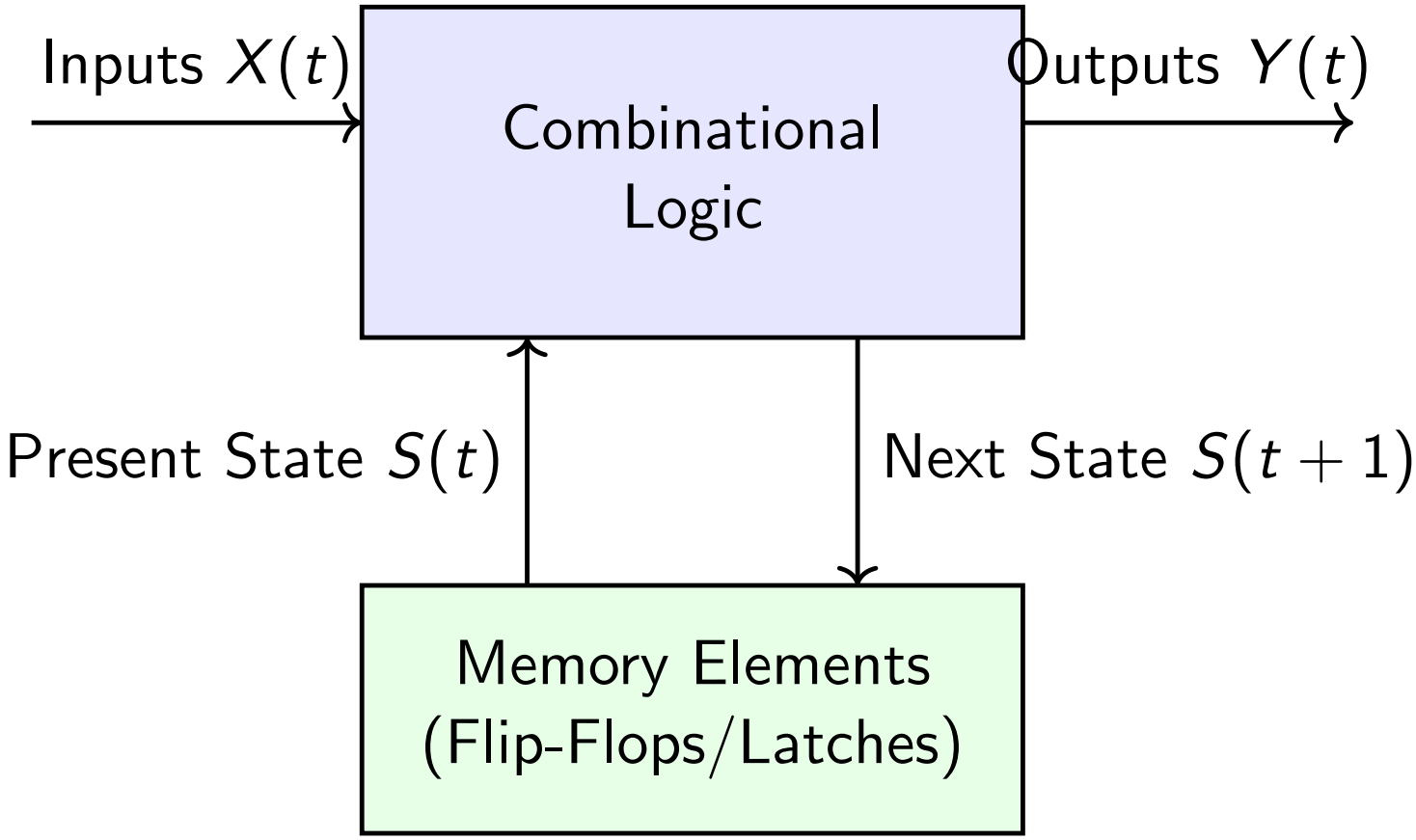
Sequential circuits employ memory elements to store past state information, influencing future outputs.

Defining Sequential Logic Circuits II

Key Points

- Outputs depend on both current inputs and the present state of the system
- Contains a combinational logic section and a memory/feedback path section
- Mathematically: $Y(t) = f(X(t), S(t))$ and $S(t+1) = g(X(t), S(t))$
- Memory elements store binary state information (bits)
- Enables the construction of counters, registers, state machines, and CPU control units

Block Diagram of a Sequential Circuit I



Block Diagram of a Sequential Circuit II

Key Points

- Combinational logic maps input vector X and present state S to outputs Y and next state S+
- Memory elements update state based on a control/clock signal
- Feedback path is critical: without it, the circuit cannot retain state over time
- Memory elements are typically synchronized via a clock signal in synchronous systems

Contrast: Combinational vs. Sequential Circuits I

Operational comparison between memoryless combinational logic and state-based sequential logic.

Contrast: Combinational vs. Sequential Circuits II

Key Points

- Memory: Combinational has no memory; Sequential contains latches or flip-flops
- Feedback: Combinational does not utilize feedback; Sequential relies on feedback paths
- Behavior: Combinational is instantaneous (ignoring propagation delay); Sequential behavior is historical
- Clock: Combinational is clock-free; Sequential often requires a clock for synchronization
- Complexity: Combinational is designed using truth tables; Sequential requires state diagrams and excitation tables

Synchronous vs. Asynchronous Sequential Circuits I

Sequential circuits are classified by how state changes are triggered and synchronized.

Synchronous vs. Asynchronous Sequential Circuits II

Key Points

- Synchronous circuits update memory elements at specific clock transitions (edge-triggered)
- Asynchronous circuits update state immediately when input changes occur without a clock
- Synchronous designs are highly predictable, widely used, and immune to timing races
- Asynchronous designs can be faster but suffer from hazards, races, and complex design constraints
- Clock skew and clock distribution networks are major design challenges in synchronous sequential systems

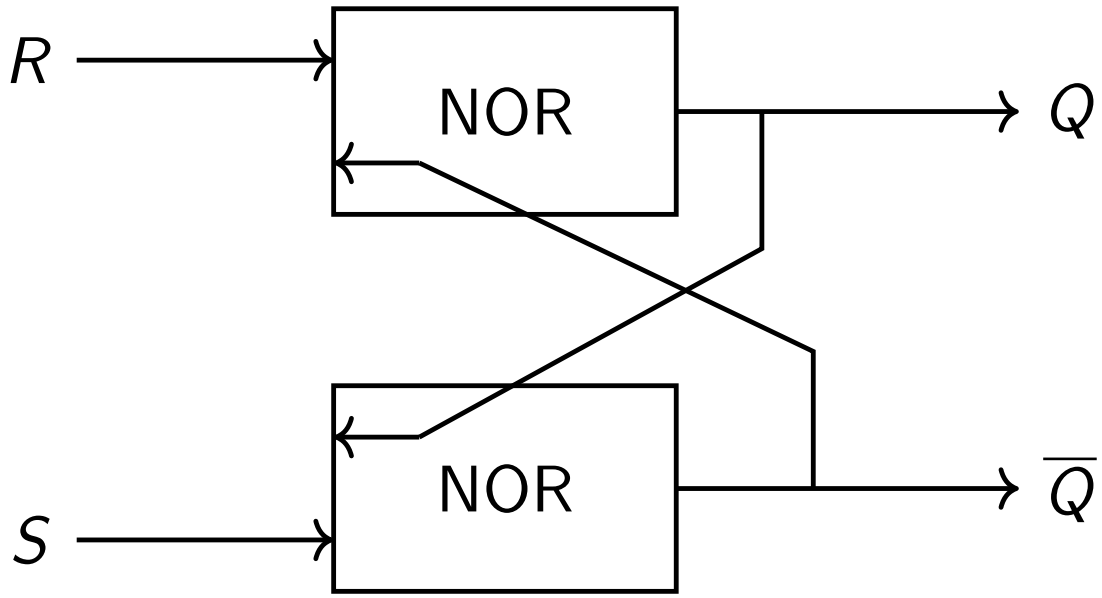
The State Memory Elements: Latches vs. Flip-Flops I

The core unit of a sequential circuit is the 1-bit memory cell.

Key Points

- Latches are level-sensitive memory elements (output responds directly to input when enabled)
- Flip-Flops are edge-sensitive/edge-triggered memory elements (updates state only at clock transitions)
- Level-sensitive latches are prone to race conditions if feedback paths are not carefully controlled
- Edge-triggered flip-flops are constructed using master-slave configurations or edge-detector gates
- Common flip-flop types: SR, JK, D, and T flip-flops

The SR Latch: Cross-Coupled NOR Gates I



The SR Latch: Cross-Coupled NOR Gates II

Key Points

- Cross-coupling is the core mechanism that provides regenerational feedback
- When R=0 and S=0, the circuit remains in its previous state (Latch / Hold)
- R=1 and S=0 forces Q=0 and Q'=1 (Reset state)
- R=0 and S=1 forces Q=1 and Q'=0 (Set state)
- R=1 and S=1 is an invalid state that results in unpredictable behavior when inputs are simultaneously negated

Sequential Models: Mealy vs. Moore Machines I

Synchronous sequential circuits are formally modeled as Finite State Machines (FSMs).

Sequential Models: Mealy vs. Moore Machines II

Key Points

- Mealy Machine: Outputs are a function of both the present state and current inputs
- Moore Machine: Outputs are a function of the present state only
- Mealy machines react faster to input changes (within the current clock cycle)
- Moore machines require state transition for outputs to change, yielding cleaner, glitch-free outputs
- Choosing between Mealy and Moore depends on timing constraints and system design style

Lecture Summary & Course Roadmap I

Understanding sequential fundamentals is crucial for designing complex digital processors.

Lecture Summary & Course Roadmap II

Key Points

- Sequential circuits differ from combinational circuits due to state memory and feedback
- Latches and flip-flops act as the fundamental memory elements for state storage
- Synchronous systems utilize clock signals to eliminate hazards and achieve deterministic behavior
- Next Lecture (Lecture 29): Detailed analysis of Flip-Flops (SR, JK, D, T) and timing parameters
- Reference reading: Chapter 5 of 'Digital Design' by M. Morris Mano

Introduction to Sequential Logic I

Sequential circuits depend on both current inputs and past states, requiring memory elements.

Key Points

- Unlike combinational circuits, sequential logic incorporates feedback loops to store state information.
- Memory elements can be either level-triggered (latches) or edge-triggered (flip-flops).
- The state of a sequential circuit is represented by the binary value stored in its memory elements.
- Flip-flops act as the fundamental 1-bit storage cell in synchronous digital systems.

Comparison: Latches vs. Flip-Flops I

Understanding the distinction between level-triggering and edge-triggering is crucial for synchronous design.

Comparison: Latches vs. Flip-Flops II

Key Points

- Latches are transparent; their output changes as soon as the input changes while the enable signal is active (level-sensitive).
- Flip-flops are edge-triggered; they sample the input and update the output only at a specific transition (rising or falling edge) of the clock signal.
- Latches are prone to race conditions and are harder to analyze in complex synchronous pipelines.
- Flip-flops provide robust synchronization, making them the preferred choice for registers, counters, and state machines.

The Clocked/Gated SR Flip-Flop I

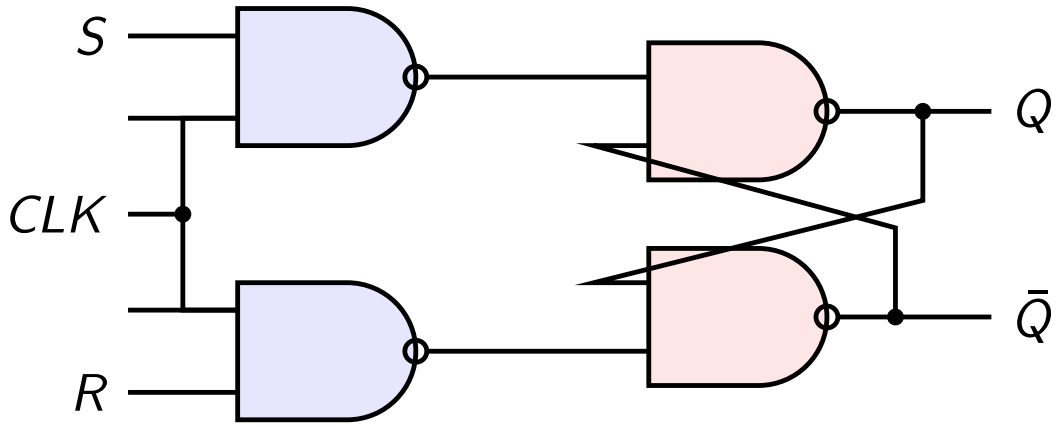
Adding a clock input to the SR latch controls when the inputs S and R can affect the state.

The Clocked/Gated SR Flip-Flop II

Key Points

- S stands for Set (forces output Q to 1) and R stands for Reset (forces output Q to 0).
- The clock signal (CLK) acts as an enable control; inputs are only read when the clock edge/level is active.
- Truth table behavior: When S=0, R=0 (Hold); S=1, R=0 (Set); S=0, R=1 (Reset); S=1, R=1 (Invalid/Forbidden).
- The S=1, R=1 state leads to an unstable or invalid condition where both Q and Q_bar try to be 0 or 1 simultaneously.

Clocked SR Flip-Flop Schematic I



Clocked SR Flip-Flop Schematic II

Key Points

- Shows a standard active-high gated SR flip-flop using NAND logic gates.
- The input NAND gates act as a steering circuit, enabling S and R inputs only when the CLK line is high.
- The cross-coupled NAND gates form the latch memory cell.
- If both S and R are logic high when CLK is high, both input NANDs output 0, driving both Q and Q_bar outputs high (invalid state).

The JK Flip-Flop: Motivation I

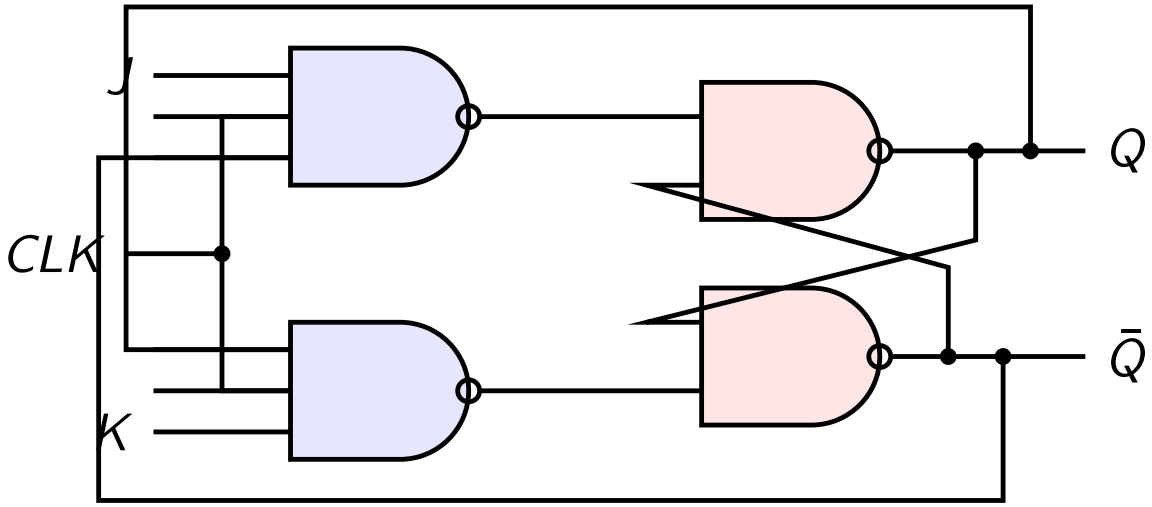
The JK flip-flop eliminates the invalid state of the SR flip-flop by feeding back the outputs to the input gates.

The JK Flip-Flop: Motivation II

Key Points

- J (Jump) corresponds to Set, and K (Kill) corresponds to Reset.
- By feeding Q_bar back to the input gate of J, and Q to the input gate of K, we ensure the input gates are selectively enabled.
- When J=1 and K=1, the flip-flop enters the Toggle state (Q(t+1) = $Q_bar(t)$), switching outputs on every clock pulse.
- This feedback loop completely prevents the forbidden condition associated with the SR flip-flop.

JK Flip-Flop Schematic with Feedback I



JK Flip-Flop Schematic with Feedback II

Key Points

- Illustrates a 3-input NAND gate implementation of the JK flip-flop.
- The outer feedback lines route the current output states back to the input gates to alter the behavior when J=K=1.
- Feedback from Q connects to the bottom input NAND gate (associated with K).
- Feedback from Q_bar connects to the top input NAND gate (associated with J).

JK Flip-Flop Behavior & Analysis I

The characteristic table defines the next state Q(t+1) based on inputs J, K, and current state Q(t).

JK Flip-Flop Behavior & Analysis II

Key Points

- For inputs J=0, K=0: Next state Q(t+1) = Q(t) (No Change / Hold state).
- For inputs J=0, K=1: Next state Q(t+1) = 0 (Reset state).
- For inputs J=1, K=0: Next state Q(t+1) = 1 (Set state).
- For inputs J=1, K=1: Next state Q(t+1) = $Q_bar(t)$ (Toggle state).
- The characteristic equation of the JK flip-flop is derived using a K-map: Q(t+1) = J.$Q_bar(t)$ + K_bar.Q(t).

The Race-Around Condition in JK Flip-Flops I

A significant issue arises in level-triggered JK flip-flops when both J and K inputs are kept high simultaneously.

The Race-Around Condition in JK Flip-Flops II

Key Points

- If J=1, K=1, and the clock pulse width (tp) is larger than the propagation delay of the flip-flop (td), the output will toggle repeatedly.
- The output Q becomes unstable and oscillates between 0 and 1 during the active clock period.
- At the end of the clock pulse, the final state of Q is unpredictable, depending on whether the pulse terminates on a high or low transition.
- To avoid this, we must satisfy the condition: tp ¡ td ¡ T (where T is the period of the clock), or use edge-triggered designs or Master-Slave JK flip-flops.

Solutions to Race-Around & Summary I

Master-Slave configurations and Edge-Triggering are the key techniques used to construct reliable sequential circuits.

Solutions to Race-Around & Summary II

Key Points

- A Master-Slave JK flip-flop cascades two flip-flops, one acting as the master (level-triggered) and one as the slave (clock-inverted).
- Since only one stage is active at any time, the feedback loop is broken during the active clock phase, eliminating race-around.
- Modern digital electronics relies almost exclusively on edge-triggered D and JK flip-flops for synchronous logic design.
- Summary: We have covered the transition from latches to SR flip-flops, identified the invalid state, and analyzed how JK flip-flops resolve this through toggle feedback.

Introduction to the D Flip-Flop I

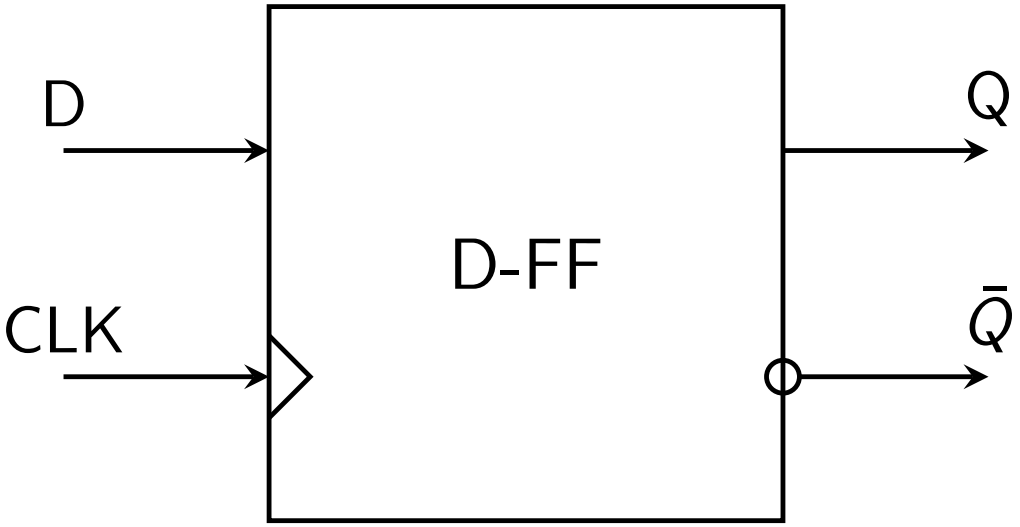
The D (Data or Delay) Flip-Flop is the most fundamental memory element in synchronous sequential design. Unlike a latch, which is level-sensitive, a flip-flop is edge-triggered, meaning it changes state only at a specific transition (rising or falling edge) of the clock signal.

Introduction to the D Flip-Flop II

Key Points

- Eliminates the 'indeterminate state' vulnerability found in SR flip-flops by forcing complementary D and D-bar inputs.
- Acts as a basic 1-bit delay element: output Q at the next clock cycle equals the input D at the current active clock edge.
- Features a single data input (D) and a clock input (CLK), along with complementary outputs Q and Q-bar.

D Flip-Flop Schematic Symbol I



D Flip-Flop Schematic Symbol II

Key Points

- The triangular wedge at the CLK input denotes edge-triggered operation (dynamic indicator).
- The bubble on the Q-bar output indicates an active-low/complementary output signal.
- This symbol represents a positive-edge-triggered D flip-flop; a bubble on CLK would mean negative-edge-triggered.

D Flip-Flop Characteristic Table and Equation I

The next-state behavior of a D flip-flop is mathematically described by its characteristic equation: $Q(t+1) = D$. This signifies that whatever value is present at input D is transferred to output Q on the next active clock edge.

Key Points

- Truth Table mapping: If D $= 0$, next state $Q(t+1) = 0$ (Reset state).
- Truth Table mapping: If D $= 1$, next state $Q(t+1) = 1$ (Set state).
- The current state $Q(t)$ has no influence on the next state $Q(t+1)$, making it ideal for data storage registers.

Critical Timing Parameters I

Physical flip-flops require stable inputs around the clock edge to prevent metastability. Setup time (t_{su}) and Hold time (t_h) define the window of vulnerability around the active clock transition.

Key Points

- Setup Time (t_{su}): Minimum time the D input must remain stable BEFORE the active clock edge.
- Hold Time (t_h): Minimum time the D input must remain stable AFTER the active clock edge.
- Propagation Delay (t_{pcq}): Time taken for the output Q to change after the active clock edge occurs.

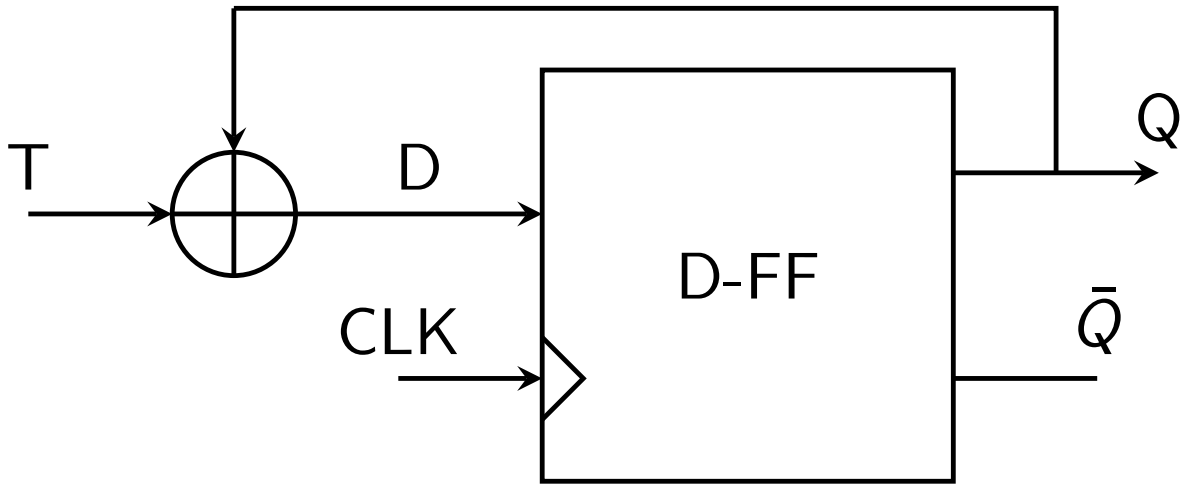
Introduction to the T (Toggle) Flip-Flop I

The T (Toggle) Flip-Flop is a sequential element that changes its state on every clock cycle if its control input T is asserted. If T is low, it maintains its current state. It is highly useful in counters and frequency dividers.

Key Points

- Derives its name from 'Toggle' action, meaning invert the current output.
- Can be constructed by feeding back the output of a D Flip-Flop through an XOR gate with the T input.
- Operating modes: T = 0 -¿ Hold (no change); T = 1 -¿ Toggle (state inversion).

T Flip-Flop Circuit Diagram I



T Flip-Flop Circuit Diagram II

Key Points

- Shows D input driven by the XOR of the external T input and the current state Q.
- Algebraic derivation: $D = T \oplus Q$, which forces $Q(t+1) = T \oplus Q(t)$.
- This feedback structure forms the standard implementation of a T flip-flop using a D flip-flop.

T Flip-Flop Characteristic Table and Equation I

The state transition behavior of the T Flip-Flop is defined by the characteristic equation: $Q(t+1) = T \oplus Q(t)$. This XOR relationship governs whether the state toggles or remains constant.

Key Points

- When T $= 0$: $Q(t+1) = 0 \oplus Q(t) = Q(t)$ (Hold state, memory is preserved).
- When T $= 1$: $Q(t+1) = 1 \oplus Q(t) = \bar{Q}(t)$ (Toggle state, output inverts).
- Useful for state machine design where transitions depend on conditional toggle events.

Practical Applications in Digital Systems I

Flip-flops form the building blocks of almost all synchronous digital systems. Their varying behaviors make them suited for different functions in computer processors and controllers.

Key Points

- D Flip-Flops: Extensively used in multi-bit registers, shift registers, and CPU pipeline buffers.
- T Flip-Flops: Used in binary counters, frequency dividers (halving the clock frequency when T=1), and ripple counters.
- Modern FPGA architectures consist of logic cells that bundle a Look-Up Table (LUT) with a D Flip-Flop.

Summary: D vs. T Flip-Flops I

While both are edge-triggered storage elements, they serve different functional roles. The D flip-flop stores data directly, whereas the T flip-flop acts as a state toggling control unit.

Key Points

- D flip-flop: $Q(t+1) = D$. Excellent for registers and buffer storage.
- T flip-flop: $Q(t+1) = T \oplus Q(t)$. Ideal for counters, frequency division, and sequential state control.
- Understanding the timing constraints (setup and hold times) is critical to avoiding metastability in real designs.