Orange Coast College
Business Division
Computer Science Department

CS 116- Computer Architecture

Appendix B & More
Basics of Digital Electronics
& Logic Design
Topics

- Number System
- Binary arithmetic
- Boolean algebra
- Logic design
- Combinational & sequential logic
**Introduction**

- **SW / HW Analogy**

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Logic functions</td>
</tr>
<tr>
<td>Data structures</td>
<td>Logic components</td>
</tr>
<tr>
<td>Objects</td>
<td>Connections</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Signal applications and delays</td>
</tr>
<tr>
<td>Define variables</td>
<td>Create registers</td>
</tr>
<tr>
<td>Perform logic function</td>
<td>Create gates</td>
</tr>
<tr>
<td>Combine logic functions</td>
<td>Combine gated &amp; other modules</td>
</tr>
<tr>
<td>Function application</td>
<td>Apply signal on logic circuits</td>
</tr>
</tbody>
</table>
Introduction

- Logic circuit hierarchy
  - Lowest-level
    - Resistors, capacitors, transistors, voltage, current, . . .
  - Low level
    - Logic gates
      - AND, OR, Inverters, Amplifiers, . . .
  - Medium level
    - Logic modules
      - Adders, subtractors, multiplexors, decoders, counters, registers, . . .
  - High level
    - Integrated circuits chips (IC)
      - Memory, . . .
  - Very-High level
    - Very-large-scale-integration (VLSI)
      - Microprocessor, embedded computers
Number Systems
Number Systems

From: http://www.thefreedictionary.com/number%20system

- number system
  - Any notation for the representation of numbers
- number representation system
- numeration system
- system of numeration
  - mathematical notation - a notation used by mathematicians
- positional notation
- positional representation system - a numeration system in which a real number is represented by an ordered set of characters where the value of a character depends on its position
- Radix
  - base - (numeration system) the positive integer that is equivalent to one in the next higher counting place; "10 is the radix of the decimal system"
Number System

- Links:
  - [http://www.cstc.org/data/resources/60/convexp.html](http://www.cstc.org/data/resources/60/convexp.html)
  - a *positional* number system, because the value of the number depends on the position of the digits
  - The values of each position correspond to powers of the base of the number system.
  - For the decimal number system, which uses base 10, the place values correspond to powers of 10:
    
    | ... | 1000 | 100 | 10 | 1 |
    |-----|------|-----|----|---|
    | ... | $10^3$ | $10^2$ | $10^1$ | $10^0$ |

- Exercise:
  - Find out the weights for binary, octal, & hexadecimal
Number System

- Other systems
Number Systems

- Some of the practical number systems are
  - decimal (r=10)
  - binary (r=2)
  - octal (r=8)
  - hexadecimal (r=16).
- We can represent integers as well as fractional numbers in any radix.
Number Systems

- Number system of base (radix) $r$:
  - A system that uses distinct symbols for $r$ digits.
  - Examples: Decimal system
    - $r = 10$
    - Digits: 0, 1, 2, ..., 9

- Numbers are represented by a string of digit symbols.
  - Example:
    - 12345
Number Systems

- The digits equivalent to decimal values 11 to 15 in hexadecimal are A, B, C, D, E, & F respectively.
- To differentiate between numbers of different radix, we will use the following notation: 
  \[(\text{number})_{\text{radix}}\]

Fractions:

- Digits to the left of the fractional point are multiplied by positive values (0 included)
- Digits to the right of the fractional point are multiplied with negative values (Start from \(-1\))
Number Systems

- To determine the quantity that the number represents
  - Multiply each digit by an integer power of \( r \)
  - Form the sum of all weighted digits
  - Example:
    - \( 12345 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 \)

- The least-significant digit (LSD)
  - Represents the lowest-value digit of the number
  - Written on the rightmost position of the number.

- The most-significant digit (MSD)
  - Represents the highest-value digit
  - Written on the leftmost position of the number.
Number Systems

Examples

- Decimal
  - \((845.67)_{10}\) = \(8 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}\)

- Octal
  - \((45.67)_{8}\) = \(4 \times 8^1 + 5 \times 8^0 + 6 \times 8^{-1} + 7 \times 8^{-2}\)

- Hexadecimal
  - \((8F5.6B)_{16}\) = \(8 \times 16^2 + 15 \times 16^1 + 5 \times 16^0 + 6 \times 16^{-1} + 11 \times 16^{-2}\)

- Binary
  - \((1011.011)_{2}\) = \(1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-3}\)
From Decimal to Other Radixes

1. Separate the number into its integer & fraction parts.
2. For the integer part, convert by successive division by \( r \) and accumulate the remainders.
3. For the fraction part, successive multiplication by \( r \) and accumulate the integer digits obtained.
Binary Number System

- Use radix 2 (Only two symbols are used)
  - True/False or 0/1.
- Useful in digital circuits in which two states are involved (ON & OFF).
- For n-digit binary integer:
  - Least-significant bit/digit (LSB/LSD) \( (2^0) \) is on the rightmost position
  - Most-significant bit/digit (MSB/MSD) \( (2^{n-1}) \) on the leftmost position.
- Each 1 bit in the binary number generates a decimal equivalent for that place value.
Binary Number System

- **Integer Binary Numbers**
  - Examples: \((n=6)\)
    \[(101101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\]

- **Fractional Binary Numbers**
  - Examples:
    \[(101.101)_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}\]
Binary $\Leftrightarrow$ Decimal Conversion

Example: $(41.6875)_{10} \Rightarrow (\ ??? )_2$

<table>
<thead>
<tr>
<th>Integer = 41</th>
<th>Fraction = 0.6875</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>0.6875 x 2</td>
</tr>
<tr>
<td>20</td>
<td>1 LSB</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>MSB</td>
</tr>
</tbody>
</table>

$= (101001.1011)_2$
Binary ↔ Hexadecimal Conversion

- From Binary to Hexadecimal
  - Since $2^4 = 16$, 4 binary digits represent a hexadecimal digit.
  - For integer part:
    - Right-to-left
    - Partition the binary number into a group of 4 digits
    - If necessary add some leading zeroes to the leftmost digit.
  - For fractional part:
    - Left-to-right
    - Partition the digits from left-to-right.
    - If necessary add zeroes to the right

- Example:
  \[(1010111101100011)_2 = 1010 1111 0110 0011\]
  \[\text{Partition the number}\]
  \[= (A \ F \ 6 \ 3)_{16}\]
Binary ↔ Hexadecimal Conversion

- From Hexadecimal to Binary
  - Convert each hexadecimal digit to its equivalent binary number.

Example:

\((3A5FB.CD)_{16} = (0011 1010 0101 1111 1011. 1100 1101)_{2}\)
Binary ⇔ Octal Conversion

- **From Binary to Octal**
  - Since $2^3 = 8$, 3 binary digits represent an octal digit.
  - For integer part:
    - Partition the binary number into a group of 3 digits from right-to-left. If necessary add some leading zeroes to the leftmost digit.
  - For fractional part:
    - Partition the digits from left-to-right.

- **Example:**
  - Original number: $(1010111101100011)_2$
  - Partition the number: 001 010 111 101 100 011
  - Octal Equivalent: $(127543)_8$
Binary ↔ Octal Conversion

- From Octal to Binary
  - Convert each octal digit to its equivalent binary 3 digit combination.

- Example:
  - Octal value: \((5237)_8\)
  - Equivalent binary: \((101 010 011 111)_2\)
Representing Negative Numbers

- To represent values that are less than zero
- Two forms
  - Sign-magnitude
  - Complement
Representing Negative Numbers

- **Sign-magnitude form**
  - The number is divided into 2 parts
    - Sign
    - Magnitude.
Representing Negative Numbers

- Complement form
  - Used in digital circuits for simplifying subtraction operations & logical manipulation.
  - Forms:
    - Diminished-radix complements
    - Radix Complement
Diminished-radix Complements

- Obtained by subtracting each digit from \((\text{radix}-1)\)
- For decimal it is called 9's Complement
  - Subtract each decimal digit from \((\text{radix}-1)\)
  - \(10-1 = 9\)
- For binary it is called the 1’s complement
  - Subtract each binary digit from \((\text{radix}-1)\)
  - \(2-1 = 1\)
Radix Complement

- Obtained getting the diminished-radix complement then adding 1 to the resulting number
- For decimal it is called 10’s complement
- For binary it is called 2’s complement
Examples

9’s Complement

Example:

\[(546789)_{10} = 999999 - 546789 = (453210)_{\text{in 9's complement}}\]
Examples

- 1's Complement
  - The effect of subtracting the number from all 1’s number is that every digit is inverted to its complement (changing each 1 to 0 and each 0 to 1).
  - Disadvantage
    - Two values for zero
      - +0 = 0000 0000
      - -0 = 1111 1111

- Example:
  - \((1101)_2 = (0010)\) 1's complement
2's Complement

- The sign as well as the magnitude can be determined.
- The MSB is the sign bit
  - MSB = 0 => number is +ve,
  - MSB = 1 => number is -ve.
- The 2's complement representations of all positive values are the same as the binary equivalent for the decimal numbers
- Only the -ve number representation will differ.
- Advantage
  - Binary subtraction is carried out more easily as an addition operation
Binary ⇔ 2's Complement Conversion

Steps:
1. Separate the sign & the magnitude. -ve sign means that the sign bit =1 in 2's complement representation.
2. Convert the number to its 1's complement form.
3. Add +1 to the 1's complement.
### Examples

<table>
<thead>
<tr>
<th>Signed decimal</th>
<th>8-bit 2(^\text{nd}) complement representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 128</td>
<td>100000000</td>
</tr>
<tr>
<td>+ 127</td>
<td>01111111</td>
</tr>
<tr>
<td>- 127</td>
<td>10000001</td>
</tr>
<tr>
<td>+ 126</td>
<td>01111110</td>
</tr>
<tr>
<td>- 126</td>
<td>10000010</td>
</tr>
<tr>
<td>+ 125</td>
<td>01111101</td>
</tr>
<tr>
<td>- 125</td>
<td>10000111</td>
</tr>
<tr>
<td>+  2</td>
<td>00000010</td>
</tr>
<tr>
<td>-   2</td>
<td>11111110</td>
</tr>
<tr>
<td>+   1</td>
<td>00000001</td>
</tr>
<tr>
<td>-   1</td>
<td>11111111</td>
</tr>
<tr>
<td>+   0</td>
<td>00000000</td>
</tr>
</tbody>
</table>
Binary Codes

- Encoding:
  - Webster College Dictionary: “To transform one system of communication into another”
- Straight binary numbers are difficult for people to understand.
- Several other codes exist
  - 8421-Binary-Coded Decimal (BCD)
  - Excess-3 BCD Code (XS3)
  - Gray Code
8421-Binary-Coded Decimal (BCD)

- Is a weighted binary code for decimal digits 0-9.
- Although longer, BCD numbers are used in digital systems when numbers must be easily converted to decimals.
- The binary codes representing 10 to 15 aren't used in the BCD code. They are considered as don't care entries and help in reducing the circuit design.
## 8421-Binary-Coded Decimal (BCD)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>BCD (8 4 2 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>
BCD ⇔ Binary Conversion

- The binary point becomes the decimal point.
- **Integer part** of the number is grouped into 4-bit groups from **right-to-left**.
- **Fractional part** is grouped from **left-to-right**.
- Each 4-bit group is converted into its decimal equivalent.

**Examples:**

\[
(001\ 0000\ 0011\ .\ 0101)_2 = (1\ 0\ 3\ .\ 5)_{BCD}
\]

\[
(1\ 0\ 3\ .\ 5)_{BCD} = (001\ 0000\ 0011\ .\ 0101)_2
\]
Excess-3 BCD Code (XS3)

- A non-weighted binary code.
- Number is always 3 more than the 8421 BCD code.
- Has a significant value in arithmetic circuits.
- If each bit is complemented, the resulting 4-bit word will be the 9's complement of the number.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>BCD(8 4 2 1)</th>
<th>XS3 BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>
Gray Code

- Non-weighted binary code.
- The code for successive decimal equivalent change only by 1 bit. This is helpful in digital circuit implementations.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>BCD(8 4 2 1)</th>
<th>Gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>1 1 0 1</td>
</tr>
</tbody>
</table>
Alphanumeric Codes

- Codes used to represent alphanumerical characters.
- The most common type of non-numeric data is text (string of characters).
- Each character is represented by a bit-string following some established convention.
- Several codes exist:
  - ASCII
  - EBCDIC
  - Unicode
ASCII Code

- Abbreviation for “American Standard Code for Information Interchange”
- Uses 7-bits to represent a character
- $2^7 = 128$ different characters
- Used extensively in small computers to translate characters from the keyboard, printers, or video displays to computer language
EBCDIC Code

- Abbreviation for “Extended Binary-Coded Decimal Interchange Code”
- Uses 8-bits to represent a character
- \(2^8 = 256\) different characters
Unicode Code

- 16-bits to represent a character
- $2^{16}$ different characters
- Universal encoding of characters of most human languages
- Used by Java language and most operating systems
Binary Arithmetic

- Binary Addition
  - Adding Two Digits
    - 4 possible combinations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary Arithmetic

Possible operations
- Addition
- Subtraction
- Multiplication
- Division
## Binary Addition

Adding two single digit numbers
- Only 4 possible combinations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## Binary Addition

### Considering the Carry-In
- Adding 3 Digits
- 8 possible combinations

<table>
<thead>
<tr>
<th>Carry-In</th>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry-Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary Addition

Notes:
- When two 1's are added together, their sum is 10 in base 2
- For the LSDs of the numbers to be added, there are only two digits to consider
- For all the following digits, they may have a carry digit from the previous column
Binary Subtraction

- Subtracting Two digits
- 4 possible combinations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Difference</th>
<th>Borrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Binary Subtraction

- Subtracting 3 digits
  - Considering the borrow-in
  - 8 possible combinations
  - We don’t really need it
  - Self-exercise

<table>
<thead>
<tr>
<th>Borrow-In</th>
<th>A</th>
<th>B</th>
<th>Difference</th>
<th>Borrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Binary Subtraction

- Subtraction with borrow is not efficient for digital systems.
  - Extra logic is needed
  - Better solution is by using the 2’s complement for subtractions
Subtraction with Complements

If A and B are binary numbers, to subtract X - Y, we need to perform two steps:

- Convert Y to 1’s complement by inverting every 1 to 0 and every 0 to 1.
- Convert the result to 2’s complement by adding one to the inverted 1’s complement.
- Add the resulting 2’s complement to X.
Subtraction with Complements

Example:

Let $X = (21)_{10} = (10101)_2$

$Y = (19)_{10} = (10011)_2$

$X - Y = (21)_{10} - (19)_{10} = (2)_{10}$

$Z = 1's$ Complement of $Y = 01100$

$W = 2's$ complement of $Y = 01101$

$X + W = 10101$

$+ 01101$

00010

$= (2)_{10}$
Binary Multiplication

- The product of two binary numbers is 1
  - ⇔ both digits are 1's.
- The product of two binary numbers is 0
  - ⇔ any of the digits is 0.
- The result needs double the size of the operands
- How to multiply?
  - Can be done as a group of shifting and additions.
  - Can also be carried out with fractional numbers.
Binary Multiplication

Example:

\[
\begin{align*}
(27)_{10} &= 11011 \\
(21)_{10} &= \times 10101 \\
&= 11011 \\
&\quad \downarrow \quad \downarrow \\
&00000 \\
&11011 \\
&\quad \downarrow \quad \downarrow \\
&00000 \\
&11011 \\
&\quad \downarrow \quad \downarrow \\
&100011 0111 \\
&\quad \downarrow \quad \downarrow \\
&\quad \downarrow \quad \downarrow \\
&= 1 + 2 + 4 + 16 + 32 + 512 = (567)_{10}
\end{align*}
\]
Binary Multiplication

Fractional Binary Multiplication

Example:

\[
(3/8)_{10} = 0.011 \\
(1/4)_{10} = \times 0.010 \\
0.00011 = 1/16 + 1/32 \\
= (3/32)_{10} \\
= (0.09375)_{10}
\]
Binary Division

- Similar to decimal number divisions.
- Converted to a group of shifts, subtractions, & comparisons.
- Can also be applied to fractional numbers.
- Causes overflow if the divisor is zero or the quotient needs more bits to express than the computer word (or register) size.
- There are special techniques for performing division directly on 2's complement numbers.
Binary Division

Example:

\[
\begin{array}{c}
(50)_{10} \\
(5)_{10} \\
(50)_{10} / (5)_{10}
\end{array}
\]

\[
\begin{array}{cccccccccc}
& & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
\hline
\text{dividend} & \text{shifted divisor} & \text{quotiont}
\end{array}
\]

\[
\begin{array}{c}
1010 \\
101 \\
0 0101 \\
101 \\
000
\end{array}
\]

\[
= 110010 \\
= 101 \\
= (10)_{10}
\]
Boolean Algebra

Matrix:

- **Algebra:**
  - Formal mathematical system with a set of objects and operations on those objects
  - Examples: Numerical algebra, set algebra, and Boolean Algebra

- **Boolean Algebra**
  - Invented by George Bool, (1815-1864)
  - mathematician & philosopher
Boolean Algebra

- Is the algebra of logic
- A way to express logic functions with logic equations.
- Extensively used in electronic circuits to build computers and other logic circuits
- Can be used to analyze and describe the behavior of logic circuits.
Elements of Boolean algebra

- Variables
- Values
- Operations
  - AND, OR, NOT
- Laws
  - a + 1 = 1
  - A + 0 = 0
  - Not (Not (a)) = a
  - ...
Variables & Values

- Variables
  - Usually given the names
    - a, b, c, ...
  - Should be of Boolean type
  - Can change values over time

- Values
  - Domain: 0 and 1 (True & False)
Operators

- AND, OR, NOT
- Both AND and OR are binary, commutative and associative operators
- The NOT operator is unary
- Follow fixed priority (precedence) scheme
Operators Precedence

- To determine order of evaluation
- To override the given precedence, use parentheses

1. NOT
   - Negation/inversion
   - Represented by: ', ! (in C-Language), ~, \( \neg \), or \( \neg \)

2. AND
   - Logical product
   - Represented by: \( \cdot \), *, && (in C-Language), or ^

3. OR
   - Logical sum
   - Represented by: +, | | (in C-Language), or v
Operators Precedence

Example:

- Evaluate: $X + Y \cdot ! Z$
- Order used: $(X + (Y \cdot (!Z)))$
  - First negate $Z$
  - Then AND with $Y$
  - The OR with $x$
Boolean Functions

- Logic functions are written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables combined with the logical operators on the right-hand side.

- Boolean function
  - A specific boolean expression
  - Usually given the name F with a possible subscript
  - Has a number of input variables
  - Has a unique output value.
  - Can be described by either a truth table or by an algebraic description.
  - Two functions are the same if the truth tables are the same
Boolean Functions

- Function as an algebraic expression
  - Example:
    - $F(x, y, z) = x \cdot y + z$
    - Simply $F = x \cdot y + z$

- Function as a truth table
  - Example:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
<th>$X \cdot Y$</th>
<th>$X \cdot Y + Z$ ($F$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean Functions

Examples:

- \( X = A \text{ AND } B \)
- **Meaning:**
  - \( X \) is true \( \iff \) (if and only if) both \( A \) and \( B \) are true
- \( X = A \text{ OR } B \text{ AND NOT } C \)
- **Meaning:**
  - \( X \) is true \( \iff \) Either \( A \) is true or \( B \) is true and \( C \) is false
Some Definitions

- **Boolean expression:**
  - A sequence of zeros, ones, and *literals* separated by boolean operators

- **Literal:**
  - A variable or a complement of a variable.
  - Examples: \( A, B, A', B' \)

- **Product Term**
  - A single literal or a logical *product* (AND) of two or more literals.
  - Examples:
    - \( A' \)
    - \( A \cdot B' \cdot C \)

- **Sum Term**
  - A single literal or a logical *sum* (OR) of two or more literals.
  - Examples:
    - \( A' \)
    - \( A + B + C \)
    - \( A + B' + C' \)
More Definitions

- **Product-of-sums:**
  - A logical product (AND) of Sum Terms.
  - Example:
    - \( A' \cdot (A + B + C) \cdot (A + B' + C) \cdot (A' + B' + C) \)

- **Sum-of-products:**
  - A logical sum (OR) of Product Terms.
  - Example:
    - \( A' + (A \cdot B \cdot C) + (A \cdot B' \cdot C) + (A' \cdot B' \cdot C) \)
More Definitions

- **Normal term:**
  - A Product (or Sum) term in which no variable appears more than once.

- **n-variable Minterm:**
  - A normal Product term with n-literals in which all variables appear exactly once, either complemented or un-complemented.
  - There are $2^n$ such product terms.

- **n-variable Maxterm:**
  - A normal Sum term with n-literals in which all variables appear exactly once, either complemented or un-complemented.
  - There are $2^n$ such sum terms.
Minterm / Maxterm Examples

- Normal term Examples:
  - $A \cdot B \cdot C'$ Normal term
  - $A \cdot B \cdot B \cdot C'$ Non-normal term

- 4-variable Minterm:
  - $A \cdot B \cdot C \cdot D$
  - $A \cdot B' \cdot C \cdot D$
  - $A \cdot B \cdot C' \cdot D'$
  - $A' \cdot B' \cdot C' \cdot D'$

- 4-variable Maxterm:
  - $A + B + C + D$
  - $A' + B' + C + D$
  - $A + B + C' + D'$
  - $A' + B' + C' + D'$
Are Boolean Functions Useful?

- Having all the elements of an algebra (values, variables, and operators), we can use these elements to derive a series of useful laws
  - Find equivalent functions
  - Simplify a given function
- Might provide for more efficient or cheaper circuit design
Laws of Boolean Algebra

Single-Variable Theorems

1. Identity
   \[ A + 0 = A \]
   \[ A \cdot 1 = A \]

2. Dominance
   \[ A + 1 = 1 \]
   \[ A \cdot 0 = 0 \]

3. Idempotent
   \[ A \cdot A = A \]
   \[ A + A = A \]

4. Inverse (Complement)
   \[ A + A' = 1 \]
   \[ A \cdot A' = 0 \]

5. Involution (Law of the double complement)
   \[ (A')' = A \]
Laws of Boolean Algebra

Two and Three Variable Theorems

6. Commutative
   \[ A + B = B + A \]
   \[ A \cdot B = B \cdot A \]

7. Associative
   \[ A + (B + C) = (A + B) + C \]
   \[ A \cdot (B \cdot C) = (A \cdot B) \cdot C \]

8. Distributive
   \[ A \cdot (B + C) = (A \cdot B) + (A \cdot C) \]
   \[ A + (B \cdot C) = (A + B) \cdot (A + C) \]

9. Covering (Absorption / Redundance)
   \[ A + A \cdot B = A \]
   \[ A \cdot (A + B) = A \]
   \[ A + A' \cdot B = A + B \]
   \[ A \cdot (A' + B) = A \cdot B \]
Laws of Boolean Algebra

Two and Three Variable Theorems

10. Combining
   \[ A \cdot B + A \cdot B' = A \]
   \[ (A + B) \cdot (A + B') = A \]

11. Consensus
   \[ A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C \]
   \[ (A + B) \cdot (A' + C) \cdot (B + C) = (A + B) \cdot (A' + C) \]

12. Common identities
   \[ A + A' \cdot B = A + B \]
   \[ A \cdot (A' + B) = A \cdot B \]
Laws of Boolean Algebra

The most useful theorems

13. DeMorgan’s

\[(A + B)' = A' \cdot B' \quad \text{(NOR)}\]
\[(A \cdot B)' = A' + B' \quad \text{(NAND)}\]

DeMorgan's theorems apply to more than 2 variables
Laws of Boolean Algebra

You can check the correction of the above theorems by either of the following:
1. Building their equivalent logic circuits & checking the o/p of the circuit for all i/p combinations
2. Building the truth table of the function
3. Using Venn diagrams (Suitable for up to 3 variables)
4. By derivation – successive application of other laws
Duality principle

- All laws of Boolean algebra are stated in pairs
- They are obtained by:
  - swapping 0's and 1's, and
  - swapping +'s with ⋅'s
Examples

- Applying DeMorgan's

\[ X = ( ( A' + C) \cdot ( B + D') )' \]
\[ = ( A' + C)' + ( B + D')' \quad \text{By DeMorgan's} \]
\[ = ( A' \cdot C') + ( B' \cdot D') \quad \text{By DeMorgan's} \]
\[ = A \cdot C' + B' \cdot D \]

- Applying other laws

\[ X = A' \cdot B \cdot C + A' \cdot B \cdot C' \]
\[ = A' \cdot (B \cdot C + B \cdot C') \quad \text{By Distributive} \]
\[ = A' \cdot (B \cdot (C + C')) \quad \text{By Distributive} \]
\[ = A' \cdot (B \cdot 1) \quad \text{By Inverse} \]
\[ = A' \cdot B \quad \text{By Identity} \]
Truth Tables

- Completely describe any combinational logic function
- Describe the output with respect to the input
- Each entry specifies the value of the output for that particular input combination
- Each output column represents a different function
- Functions may be directly constructed from the truth table
- For a function with n-inputs, there is $2^n$ entries in the truth table
### Truth Tables

**Example:**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Deriving Functions From Truth Tables

- For each place where output is 1
  - Create an AND expression of the inputs
  - If input is 0, negate its value
  - Else, just use the variable name
  - OR all the expressions together

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ C = AB + AB \]
Deriving Functions From Truth Tables

Exercise:
- Write the functions for Sum and Carry
- Hint: This is the function for addition

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Karnaugh Maps

- A graphical representation of logic functions or truth tables.
- Allows combining product terms to minimize logic functions.
- Become impractical if we have more than 5 input variables.
Karnaugh Maps

- 2-Variables Karnaugh Map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Karnaugh Maps

- 3-Variables Karnaugh Map
- Idea:
  - The list of input values must follow a specific order
  - Each value is only one bit different than the next
  - This enabled minimization through adjacency

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
Karnaugh Maps

4-Variables Karnaugh Map

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>
Using Karnaugh Maps

Create table

- Each axis should be labelled with a variable and the value that variable can take
- For each cell in the table, insert a 1 where output is 1
- Just leave the other cells blank

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Karnaugh Maps

- Draw a line around adjacent cells with a 1
- Construct a logic function that represents the circled areas
  - Makes it easy to see where a function can be minimized

\[ \text{Output} = \overline{B} \]
Karnaugh Maps

- 3-Variables Karnaugh Map
  - Minimizing sum-of-products
  - Exercise: Write the minimized function

![Karnaugh Map Diagram]
Incompletely-Specified Functions

- Functions that have unspecified output for some input combination
- Unspecified Minterm’s of a function are usually called “Don’t care conditions”
- Don’t care terms:
  - Can have either 0’s or 1’s
  - Can be used in truth tables & Karnaugh maps for function simplification
Don't Cares

- In some situations we don't care what the value of the output would be.
- Don't cares are used in simplifying and thus optimizing the implementation of logic functions.
- Don't care terms are usually represented with an "X" or a "d" in the truth table.
- **Output don't cares:**
  - when we don't care about the value of an output for some input combination.
- **Input don't cares:**
  - When the output depends only on some of the inputs.
Don't Cares

Example:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D(Out1)</th>
<th>E(Out2)</th>
<th>F(Out3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

1. D is true if the value of A or C is true. We don't care about the value of B.
2. E is true if A or B is true. We don't care about the value of C.
3. F is true if exactly one of the inputs is true. We don't care if more than one input is true.
Don't Cares

Example:

- The previous truth table has output don't cares.
- If we simplify the truth table further, we will get both input and output don't cares.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D(Out1)</th>
<th>E(Out2)</th>
<th>F(Out3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
Don't Cares

Example: BCD Counter

- 4-bits needed
- Only 10 states are involved.
- States 10-15 are never used (Don’t care)
Don't Cares

Example:
- Design a logic circuit whose function is to Signal 1 when output reaches 9
- Exercise: Use K-Map to find the circuit

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
Logic Design

- Many mathematical & logical operations are difficult or impossible to perform with analog quantities
- Digital logic:
  - Maps real world values into two subsets corresponding to possible logic values — 0 and 1.
- Digital electronics:
  - Operate only with two voltage levels (HIGH & LOW)
  - All other voltages are temporary & occur while transitioning between the values.
  - The values & relationships between the two voltages differ from one logic family to the other.
  - Digital logic circuits can be analyzed and designed using truth tables.
Logic Design

- **Gates:**
  - The basic building blocks of logic.

- **Combinational logic systems (circuits):**
  - Systems containing no memory
  - Output depends on the input only

- **Sequential Logic Systems (circuits):**
  - Output depends on both inputs and past sequence of inputs (i.e. stored value)

- **State of a sequential circuit:**
  - A collection of state variables whose values at any one time contain information about the past to know the circuit's future state or behavior
  - Values are stored in the memory element.
Logic Design

- Positive logic circuit:
  - If 0 is assigned to LOW and 1 to HIGH

- Negative logic circuit:
  - If 1 is assigned to LOW and 0 to HIGH

- Asserted logic value:
  - Logical true (one).

- De-asserted logic value:
  - Logical false (zero)
Logic Design

- The behavior of a **combinational logic circuit** can be fully described by a **truth table** that lists all the combinations of input values and output values produced by each of the inputs.
- The behavior of **sequential circuits** can be described by a **state table** that specifies its output and next states as function of its current state and inputs.
Logic Gates vs. Logic Functions

- Any logic function can be described using three logic operators (AND, OR, NOT)
- Each logic operator can be represented as a logic gate
- Any logic function can be implemented into a logic circuit diagram
- Any logic circuit can be implemented using three types of gates (And, Or, Inverter)
- Any logic function can be represented as a sum of products (for output = 1's) or product of sums (for output = 0's) directly from the truth table.

<table>
<thead>
<tr>
<th>Logic function</th>
<th>Logic gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Inverter</td>
</tr>
<tr>
<td>AND</td>
<td>And</td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
</tr>
</tbody>
</table>
Logic Gates

- **AND gate** produces an asserted output if all its inputs are asserted.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A • B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **OR gate** produces an asserted output if one or more of its input is asserted.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A + B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic Gates

- NOT gate produces the opposite value of its input
  - NOT

<table>
<thead>
<tr>
<th>A</th>
<th>Not (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Buffer is used to amplify electrical signals
  - Buffer

<table>
<thead>
<tr>
<th>A</th>
<th>Buffered(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic Gates

- There are other types of gates that can be built out of the three essential gates.
  - **NAND (↑)**: The opposite of AND

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A NAND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **NOR[↓]**: The opposite of OR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A NOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logic Gates

Other gates

- **Exclusive-OR (⊕):** Produces asserted output if inputs differ

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Exclusive-NOR (Coincidence/⊙):** Produces asserted output if inputs are identical

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XNOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Combining Gates

- How do we build more complex circuits?
  - Link simple elements together
  - Examples:
    - \((A' + B) = A \cdot B'\)
    - \(A' \cdot B \cdot C \cdot (A + D)'\)

Logic Gates

Universality of NAND & NOR Gates

- NAND gates can be used to perform each of the Boolean operations.
- We can implement any logic expression using only NAND gates.
- De Morgan theorem can be used to convert all gates into NANDs.

Exercise:

- Build the NAND / NOR equivalent for the 3 main logic gates (AND, OR, & NOT)
Rules for Evaluating Logic Circuit Function (Output)

1. Perform all NOT operations (inversions) of single terms
2. Perform all operations within parentheses
3. Perform AND operations
4. Perform OR operations, unless parentheses indicate otherwise
Equivalent Gate Symbols

Can be proven by DeMorgan’s theorems

\[(x+y)’ = x’ \cdot y’\]

\[x + y \equiv [x’ \cdot y’]’\]

\[[x \cdot y]’ = x’ + y’\]

\[x \cdot y = [x’ + y’]’\]
Generalization of Logic Gates

- Logic gates can have more than 2 inputs
- Fan-in
  - # input lines to the gate
- Fan-out
  - # o/p lies taken from the gate

\[
\begin{align*}
& A & B & C \\
& A \cdot B \cdot C \\
& A + B + C
\end{align*}
\]
Rules for Connecting Logic Circuits

- One direction from left-to-right
- Connected lines should be indicated by a node in the circuit diagram
- Every input to a gate must have either asserted or deasserted
- Never connect the output of two or more gates with direct wiring
Combinational Logic

- Combinational circuit:
  - May contain any number of logic gates.
  - No feedback loops.

- Feedback loop:
  - A signal path connecting circuit output back to the input of that same circuit.
Combinational Logic

- Circuit Analysis:
  - Starts with a logic diagram and proceeds to formal description of the function performed by that circuit (e.g. truth table or logic expression).

- Circuit design:
  - Starts with an informal description of the circuit, try to define the circuit input & output, specify its functional behavior by means of truth tables and equations, then obtain the logic diagram.
Combinational Logic

- Two-level logic gates:
  - If the gates are arranged in two levels, one including ANDs and the other ORs.

- Examples of logic circuits:
  - Adders (Half/ Full)
  - Decoders
  - Multiplexors
  - PLAs
Decoders

- A combinational circuit that converts binary information from n-bit coded input to maximum of $2^n$ unique outputs
  - **Input:**
    - n-bit input combination
  - **Output:**
    - $2^n$ outputs, one of which will be logically true depending on the input combination
  - **Enable line**
    - Controls the output
    - If Enable = 0, the i/p word is considered a don't care value and all o/p bits will be = 0
  - Each input word produces a different output word code
Decoders

Examples:
- 3x 8 Decoder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>O7</td>
</tr>
<tr>
<td>I2</td>
<td>O6</td>
</tr>
<tr>
<td>I3</td>
<td>O5</td>
</tr>
<tr>
<td></td>
<td>O4</td>
</tr>
<tr>
<td></td>
<td>O3</td>
</tr>
<tr>
<td></td>
<td>O2</td>
</tr>
<tr>
<td></td>
<td>O1</td>
</tr>
<tr>
<td></td>
<td>O0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>O7</th>
<th>O6</th>
<th>O5</th>
<th>O4</th>
<th>O3</th>
<th>O2</th>
<th>O1</th>
<th>O0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## Decoder Circuit

**Examples:**

- **2 x 4 Decoder circuit realization**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out 1</th>
<th>Out 2</th>
<th>Out 3</th>
<th>Out 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![Decoder Circuit Diagram]
Decoders

- Multiple binary decoders can be used to build larger decoders.
- Decoders can also be used as Minterm generators.
- Example:
  \[ F = \Sigma(1,2,4,7) \]
Decoder Application

- Put data into a specific register

![Diagram of decoder application](image-url)
Encoders

- Inverse of decoders
- Output code has fewer bits than the input code.
- Input:
  - $2^n$ (or less) lines
- Output:
  - $n$ lines, binary code corresponding to the input
Encoder Circuit

Example:

4X2 Binary encoder

- For O1:
  - $A = 0 \& B = 0$
- For O2:
  - $A = 0 \& C = 0$
- Value of D doesn’t affect the output
- We can also use K-map and don’t care to get another solution

Application:

- Convert decimal to BCD
  - (See Schaum’s Outline, p. 140)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Multiplexors (MUX / Selectors)

- A combinational circuit that selects binary information from one of many input lines and directs this information to a single output line.
- Selection of a particular input is controlled by selector lines.
- **Input:**
  - n-input signals
  - \(\lceil \log_2 n \rceil\) selectors
  - Enable input.
- **Output:**
  - 1 output only
- **Application:**
  - Used in building the CPU to choose specific signals.
Multiplexor Circuit

- Circuit realization of 2x1 MUX

- Another symbol representing MUX
Multiplexors Application

Application: Selecting a register from register file:

- Selector signals
- Read register number 1
- Read register number 2
- Read data 1
- Read data 2

nx1 MUX 32-bits wide

Diagram showing a multiplexer with register inputs and selector signals.
Demultiplexors (DEMUX)

- Inverse function of MUX
- Routing data to one of several destinations

Input:
- 1 input

Output:
- $2^n$ output lines

Select:
- n-select lines

A binary decoder with an enable input can be used as a DEMUX
**Demultiplexors Circuit**

- **Truth table**
  
  \[
  \begin{align*}
  O_0 &= I \cdot S_0' \cdot S_1' \\
  O_1 &= I \cdot S_0' \cdot S_1 \\
  O_2 &= I \cdot S_0 \cdot S_1' \\
  O_3 &= I \cdot S_0 \cdot S_1
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>s0</td>
</tr>
<tr>
<td>s1</td>
<td>o0, o1, o2, o3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0, 0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
</tr>
<tr>
<td>0</td>
<td>0, 1</td>
</tr>
<tr>
<td>1</td>
<td>1, 0</td>
</tr>
<tr>
<td>1</td>
<td>0, 0</td>
</tr>
<tr>
<td>1</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

- **Circuit realization**

  ![Demultiplexors Circuit Diagram](image-url)
Binary Adders

- Combinational circuits that perform arithmetic addition on two binary input values

- **Half-Adder:**
  - Adding 2 Bits
  - Generates Sum and Carry

- **Full adder:**
  - Adds 3 bits
  - Generates Sum & Carry
Binary Half Adders

- **Half-Adder:**
  - Sum = $A \oplus B$
  - Carry = $A \cdot B$
- **Implementation**

**Truth table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![Half-Adder Circuit Diagram]
Binary Adders

- **Full adder:**
  - Sum = $A \oplus B \oplus C_{\text{in}}$
  - $C_{\text{out}} = A \cdot B + B \cdot C_{\text{in}} + A \cdot C_{\text{in}}$

- **Implementation:**
  - Three possibilities
    - Use basic logic gates
    - Use 2 Half adders
    - Use ROM

- **Truth table**

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_{\text{in}}$</th>
<th>Sum</th>
<th>$C_{\text{out}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary Adders

- **Full adder Implementation:**
  - Using basic logic gates
  - Using 2 Half adders
  - Using ROM
Binary Adders

- Ripple-Carry adder:
  - For more than one binary bit we can cascade the full-adder modules
Binary Subtractors

- **Half-Subtractor:**
  - Difference = \( A \oplus B \)
  - Borrow = \( A' \cdot B \)

We can make use of the adder to perform subtraction.
Binary Subtractors

- We can make use of the adder to perform subtraction
  - Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Difference</th>
<th>Borrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Circuit realization
Sequential Circuits

- Circuits with memory
- Outputs depend on past sequence and possibly the input
- Usually represented with state diagrams or state tables

Representation
- State (Transition) diagram
- State Table
Sequential Circuits

State (Transition) diagram:
- Circles to represent states
- Directed line segments to represent transitions between the states
- One or more actions (outputs) may be associated with each transition
- Represents a finite state machine

Finite state machine:
- A function which maps an ordered sequence of input events into a corresponding sequence of (sets of) output events
Sequential Circuits

- State diagram examples:
  - Special counter: http://www.utdallas.edu/~frankd/SD1.html
  - Invoice: http://www.dcs.warwick.ac.uk/~ananda/lnotes/node278.html
  - Multiprocessor system: http://www.dca.fee.unicamp.br/~leopini/private/sib98/sld037.htm
  - T-Flip-flop: http://ranger.uta.edu/~carroll/cse2341/spring99/chapter8/sld015.htm
Sequential Circuit Models

- Mealy machine
  - O/p depends on both input & current state
  - O/p changes whenever i/p changes
Sequential Circuit Models

- Moore machine
  - O/p depends on current state only
  - O/p changes after clock edge
Sequential Circuits

Example: State diagram for Mealy machine
Sequential Circuits

Example: State diagram for Moore machine
Clocks

- Relative execution times of instructions on a computer are usually measured by number of clock cycles rather than seconds.
- Clock rates for various models of the computer may increase as technology improves.
Clocks

Definitions:

- **Clock**
  - A free-running signal with a fixed cycle time.
  - A processor's clock or one cycle thereof.

- **Synchronous system:**
  - A system in which state changes only occur in specific time controlled by a free-running clock.

- **Asynchronous system:**
  - A system in which state changes occur according to some other events.

- **Clock cycle time (period/interval):**
  - The time between successive transitions in the same direction, i.e. a complete period in which the signal has one high and one low signal levels.
Clock Definitions

- **Duty cycle:**
  - Is the percentage that the clock signal is at its asserted level.

- **Clock frequency:**
  - The inverse of the cycle time.

- **Edge-triggered clocking:**
  - State changes occur on a clock edge.

- **Rising edge:**
  - The edge that converts the signal from low to high.

- **Falling edge:**
  - The edge that converts the signal from high to low.

- **Set-up time:**
  - The minimum time that the input must be valid before the clock edge.

- **Hold time:**
  - The minimum time during which the input must be valid after the clock edge.

A clock signal oscillates between high & low values.
Clocks

- Clock edge determines when contents of state elements are updated.
- Clock cycle must be long enough for input values to stabilize.
- Edge triggered clocking is usually preferred because a state element can be used as both input and output to the same logic circuit.
- Clocks are needed in sequential logic to decide when an element that contains a state should be updated.
- The clock must have enough period that allows all signals to stabilize.

Assumptions:
- A clock signal is active high if state changes occur at the clock's rising edge.
- A clock signal is active low if state changes occur at the clock's falling edge.
Memory Elements

- Memory element:
  - Stores values
  - Controlled usually by clock
  - Can be static or dynamic, volatile or non-volatile
Bistable element

- The most fundamental element from which all flip flops are constructed
- A pair of inverters connected to each other. No matter whether the input to one of them is TRUE or FALSE, the circuit is always self consistent
- There is no way of controlling or changing the bistable element's state. It randomly comes up to a state when power is switched on and stays there forever.
- 2 possible stable states
Latch

- From [http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=latch](http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=latch)
- A digital logic circuit used to store one or more bits
- Has a data input and an output
- When the input is applied, data on the input is "latched" or stored and transferred to the output
- Output will then retain its value until the clock goes active again Simplest type of memory element
- No clock involved
- Change can occur any time as long as the input is asserted
- After the input is applied, the latch remains in its state.
Flip-Flop (FF)

- A digital logic circuit
- Flip-Flop’s and latches are the basic building blocks of most sequential circuits. Their function is to store signals.
- Can be in one of two states which it switches (or "toggles") between under control of its inputs.
- Considered as a one bit memory.
- Changes occur only at a clock edge.
- Reading input and generating output are separate events.
Flip-Flop (FF)

- Four types:
  - SR flip-flop
  - JK flip-flop
  - D-type flip-flop (or latch)
  - T flip-flop. Non-transparent
Set-Reset (S-R) Latches

- Built from a pair of NOR or NAND gates.
- Two-Inputs:
  - S for set & R for reset.
- Two-Outputs:
  - Q output & Q' (inverted output).

Implementation Using NANDs

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?'</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Q</td>
</tr>
</tbody>
</table>

Implementation Using NORs

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>Q'</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q</td>
<td>?</td>
</tr>
</tbody>
</table>
Set-Reset (S-R) Latches

- If only S is asserted
  - Latch is switched to the set state \((Q=1\) and \(Q'=0\)).

- If only R is asserted
  - Latch is switched to the reset state \((Q=0\) and \(Q'=1\)).

- If both S & R are not asserted
  - Latch stays in its current state.

- If both S and R are asserted
  - **Undefined Dangerous !!**)
  - Circuit behaves like a bistable element.
  - Can lead to incorrect operation or oscillate in an unstable state.
Delay (D) Latch/FF

- Stores the value of its data input signal.
- Implementation
  - Using gates
  - Using an inverter with SR flip flop
- Applications:
  - Set or reset flags in response to some condition
  - Build registers and store bits of information Input using an array of D-FFs
D Flip-Flop

Output:
- The value of the internal state (Q)
- Complement of internal state (Q')

When the input is
- Asserted
  - Latch is open and the value of the output becomes the value of the input.
- Deasserted:
  - Latch is closed and the output holds its value.
D Flip-Flop

Truth table:

<table>
<thead>
<tr>
<th>Clock</th>
<th>D</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>Q</td>
<td>Q'</td>
</tr>
</tbody>
</table>

Operation of a D-latch
Assumption: output initially deasserted
Master-Slave FFs

Also called **Edge-Triggered or Falling-Edge FF**

Example: Two D-FFs:

- The Master is open and follows the input (D) when the clock (C) is asserted.
- The slave is open when the clock falls, while the master will be closed. It gets the input from the output of the master FF.
Master-Slave Flip-flops

- During each clock period at most one state change will take place
  - Race conditions is avoided
- Especially useful when input of a FF is a function of its own output
- We can also have Master-Slave S-R or J-K FFs

Operation of a D-FF with a falling edge trigger, assuming the output is initially deasserted
Master-Slave FFs

- Racing:
  - Multiple internal variables change states as a result of a single input changing state

- Non-critical racing
  - If exactly one final state is reached regardless of the order and speed of internal variable changes

- Critical racing:
  - If two or more different final states could be reached depending on the order and speed of the internal variable changes

- Critical racing should be avoided, since the behavior of the sequential circuit cannot be predicted
Other Flip-Flops

- **J-K-FFs**
  - A modified version of SR-FF
  - Prevent both inputs from being assigned 1 simultaneously

- **T-FFs**
  - A single-input version of JK-FF
  - Both inputs are tied together
  - The output toggles whenever input is applied
## Characteristics Tables of FFs

### J-K FF

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q[t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>q(T) No change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 Reset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 Set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>q'(T) Complement</td>
</tr>
</tbody>
</table>

### S-R FF

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q[t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>q(T) No change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 Reset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 Set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>? Unpredictable</td>
</tr>
</tbody>
</table>

### D FF

<table>
<thead>
<tr>
<th>D</th>
<th>Q[t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>Set</td>
</tr>
</tbody>
</table>

### T FF

<table>
<thead>
<tr>
<th>T</th>
<th>Q[t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q(t) No change</td>
</tr>
<tr>
<td>1</td>
<td>q'(t) Complement</td>
</tr>
</tbody>
</table>
## Excitation Tables of FFs

### S-R FF

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q(t)</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

### J-K FF

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q(t)</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

### D FF

<table>
<thead>
<tr>
<th>D</th>
<th>Q(t)</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### T FF

<table>
<thead>
<tr>
<th>T</th>
<th>Q(t)</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Sequential Circuits Design Steps

1. State the description of the circuit behavior, using state diagram, timing diagram, or any other potential information.
2. Obtain the state table.
3. If possible, reduce the number of states by some state-reduction method.
4. Assign binary value to each state.
5. Determine the number of FFs needed for the state combination.
6. Choose the type of the FFs used.
7. From the state table, derive the excitation table.
8. Derive the circuit output functions anf FFs input functions, using Karnaugh map, or any other simplification method.
9. Draw the logic diagram.
Sequential Circuits Analysis Steps

1. Given the logic diagram of the circuit, determine the value of the input function, in terms of the present state and input variables, for each FF in the circuit.

2. Use the corresponding FF characteristics table to determine the next state.

3. Build the State table, from the information known, with columns indicating the present state, input, next state, and output.

4. Draw the state diagram.
Registers

- High-speed memory locations in a computer's CPU.
- Only a small number of registers available ("Register set" or "Register file")
- Typically 32 in a modern processor (Some processors, e.g. SPARC, have as many as 144)
Registers

- May be directly addressed with a few bits
- Fast access; typically, two registers can be read and a third written -- all in a single cycle
Special Registers

- **SPIM:**
  - **$0:**
    - Contains always zero.
    - No write logic required for this register.
  - **$31:**
    - Contains return address (link) for procedure calls.
    - Can only be loaded via ALU
Special Registers

- **Shift Registers**
  - **Serial shift-register:**
    - The o/p of one FF is the i/p of the next
  - We can add more circuit to allow the shift in the other direction too

4-Bit Shift-register

Input
Clock

Q0
Q1
Q2
Q3

Clock (Clear)’
In1
In2
In3
In4

4-Bit Register

Out1
Out2
Out3
Out4
Special Registers

- Parallel-Load register
  - The input is entered in parallel
  - The parallel load input can also be added to the previous register to generate a universal register that allows:
    - Shift left
    - Shift right
    - Parallel load
    - Clear
Register Files (Register Sets)

- Set of 32 registers
  - Indexed by register number

- Input signals:
  - Register numbers (source & destination)
  - Data
  - Write signal

- Output signals
  - Result
Implementation

Implementation options:
- Decoder for each read or write port
- Multiplexor to choose the read port
- Array of registers
Operations

- Reading from a specific register.
  - Input: Supply the register number.
  - Output: Data contained in the indicated register.
  - Reading a register doesn’t change its state.

- Writing into a register.
  - Input: Supply the data, the register number, clock
  - Output: No output is specified.
A register file with 2 read ports & 1 write port
5 inputs & 2 outputs
Implementation of Register Files

- Implementation of 2 read ports

![Diagram of register file implementation with 2 read ports and a multiplexer (MUX) to select between register 0 to register (n-1) and register n.]
Implementation of Register Files

- Implementation of the Write ports
  - Decoder is used to generate clock input to registers
Counters

Counter:

- A register that goes through a prescribed sequence of states when i/p is applied
- Input may be a clock pulse or other source
- Input may be random or at fixed time intervals
Binary Counters

- A digital circuit
- Has a clock input
- Has a number of count outputs which give the number of clock cycles
- Output may change either on rising or falling clock edges
- May have a reset input which sets all outputs to zero when asserted
- May be either a synchronous counter or a ripple counter
Other Counters

- **n-bit binary counter:**
  - Counts from 0 to \(2^n - 1\)
  - Needs \(n\)-FFs
- **Synchronous counter:**
  - All FFs receive a common clock input
  - Change of state is determined from the present state of the counter
- **Ripple counter:**
  - The output of one FF is used to trigger another FF
Counter Examples

- 2-bit binary counter:
  - 2 Flip-flops are needed

Exercise
- Draw the sequential circuit

<table>
<thead>
<tr>
<th>F0 (t)</th>
<th>F1 (t)</th>
<th>f0 (t+1)</th>
<th>f1 (t+1)</th>
<th>F1 S</th>
<th>F1 R</th>
<th>F2 S</th>
<th>F2 R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Counters Examples

- Ripple counter:
  - See this website
Programmable Logic devices (PLD)

- Programmable Logic:
  - A logic element whose function is not restricted to a particular function
  - May be programmed at different points of the life cycle.
    - At the earliest, it is programmed by the semiconductor vendor, by the designer prior to assembly, or by the user, in circuit.

- PLD Compiler:
  - A software tool to program PLD’s
  - Converts functional descriptions into connection
PLD Details

- A large-scale integration (LSI) chip
  - Contains regular circuit structure
  - Can be customized for a specific application
  - Programmed by the purchaser
  - Densities range from 1,000 to 10,000 gates
  - Reduces cost for both customer and manufacturer
  - Many modern PLDs are erasable and can be reprogrammed

- Implementation:
  - A programmable AND array
  - Followed by a fixed fan-in OR gates
  - Followed by flip-flops.
  - Product lines can be any input combination
  - Flip-flops can be fed back to input
  - Programmable element can be fuse or a transistor
Basic PLD block diagram

- **i/p buffer**
- **Product (AND) Matrix**
- **Sum (OR) Matrix**
- **o/p buffer**

**Inputs:**
- \((A_0, A_{m})\) m i/p lines
- \((A'_0, A'_m)\) 2 m i/p lines & complements

**Outputs:**
- \((B_0, B_p)\) n o/p lines
Programmable Logic Array (PLA)

- A combinational, 2-level, AND-OR device programmed to realize any sum-of-product logic expressions
- Has both programmable AND and OR planes.
- Programmed by establishing the connections that are actually needed.
Programmable Logic Array (PLA)

- A set of inputs with corresponding input complements
- Two stages of logic.
  1. First stage:
     1. Array of AND gates (AND-plane) forming a set of product terms
  2. Second stage:
     1. Array of OR gates (OR-plane), forming a logical sum of any number of the product terms
- Can directly implement truth table
  Number of OR gates correspond to the number of truth table entries for which the output is true.
Implement the function:

\[ D = (A' \cdot B' \cdot C) + (A' \cdot B \cdot C') + (A \cdot B' \cdot C') + (A \cdot B \cdot C) \]

Exercise:
- Find the equations for E and F

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D (Out1)</th>
<th>E (Out2)</th>
<th>F (Out3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Read-Only Memories (ROMs)

- Nonvolatile memory
- After they are created, they can only be read but not written
- A combinational circuit that encodes a collection of logic functions directly from truth tables
- Manufactured with fixed contents
- Internal structure works with diodes or transistors
  - Presence or absence of a diode or a transistor distinguishes between 0 and 1
- Modern ROMs are fabricated as a single IC chip.
ROMs

Advantages:
- Low-cost
- Inherently non-volatile

Application:
- Programs for embedded systems
  - Microprocessor controlled (e.g. washers, microwave, ...)
- Storage of the lowest level bootstrap software (firmware) in a computer.
- Permanent storage of computer programs
- Look-up tables
ROMs Characteristics

- **Input (height):**
  - n-input lines
  - $2^n$ addressable entries (Address lines)

- **Output (Width):**
  - Number of bits (d) in each addressable entry.
  - Total number of bits = Height x Width = $d \times n$
ROMs Variations

- Programmable ROMs (PROMs):
  - Can be programmed electronically, when the designer knows their contents.

- Erasable PROMs (EPROMs):
  - Requires slow erasure process using ultraviolet light.
  - Special devices are needed for reprogramming

- Electrically Erasable PROM (EEPROM):
  - Stored bits may be electrically erased.
## ROMs vs. PLAs

<table>
<thead>
<tr>
<th></th>
<th>ROM</th>
<th>PLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully decoded:</td>
<td>Contains a full output word for each possible input combination.</td>
<td>Partially decoded.</td>
</tr>
<tr>
<td>Number of entries</td>
<td>Number of entries grows exponentially with number of inputs.</td>
<td>Number of product terms grows more slowly.</td>
</tr>
<tr>
<td>Less efficient</td>
<td>Less efficient for implementing combinational logic.</td>
<td>More efficient for implementing combinational logic.</td>
</tr>
<tr>
<td>Ability to</td>
<td>Ability to implement any logic function with the matching number of</td>
<td>To implement a different function, modification is required.</td>
</tr>
<tr>
<td>implement any</td>
<td>input and outputs.</td>
<td></td>
</tr>
<tr>
<td>Easy to change</td>
<td>Easy to change ROM's contents if the logic function changes.</td>
<td>To change of content requires modification in the connections and/or</td>
</tr>
<tr>
<td>ROM's contents</td>
<td></td>
<td>number of gates.</td>
</tr>
<tr>
<td>Size doesn't change</td>
<td>Size doesn't change.</td>
<td>Size might change.</td>
</tr>
</tbody>
</table>

**PLA**: Partially decoded.

**Contains a full output word for each possible input combination.**

**Number of entries grows exponentially with number of inputs.**

**Less efficient for implementing combinational logic.**

**Ability to implement any logic function with the matching number of input and outputs.**

**Easy to change ROM's contents if the logic function changes.**

**Size doesn't change.**
Static RAMs (SRAM)

- Registers & register files provide the basic building blocks for small memories.
- Large memories are built using SRAMs or DRAMs.
SRAM

- IC-chips memory arrays with read/write port
- Each bit of storage is a bistable flip-flop (pair of inverting gates)
- Retains a value as long as power is supplied
- It is however, still volatile
  - Will lose its contents when the power is switched off
- Usually faster than DRAM
- Compared to DRAM, less bits of SRAM can be put in the same area
- Costs more per bit than DRAM
  - Used for the most speed-critical parts of a computer (e.g. cache memory) or other circuit
SRAM

- Type of SRAM is defined by:
  1. Height: Number of addressable locations
  2. Width: Width of each addressable location
- SRAMs have fixed access time (5-25ns).
- Address line depends on the first number
- To initiate read/write access, the Chip select signal must be active.
- The Output enable signal allows the output data to be accessed.
SRAM Example

- $256k \times 1$ SRAM $\equiv 256k \ (=2^{18})$ entries, each 1-bit wide
  $\Rightarrow$ needs
  18 address
  1 data input
  1 data output lines

- $32k \times 8$ SRAM $\equiv 32k \ (=2^{15})$ entries, each 8 bits wide
  $\Rightarrow$ Needs
  15 address
  8 data output
  8 data input lines are needed
SRAM Operations

1. Reading from a specific location.
   - input: Register number.
   - output: Data contained in the indicated register.

2. Writing into a specific location.
   - input: Data, location, write enable signal, and Chip select signals.
   - output: The Chip select & Output enable signals should be activated.
SRAM Operations

- There are set-up and hold-time requirements for the address & data lines.
- Write enable signal isn't a clock, but a pulse with minimum width requirements.
- Instead of using MUXs, large memories are implemented with a shared o/p line (Bit line).
- Bit line:
  - A shared line that multiple memory cells in a memory array can assert.
Basic structure of an 4 x 2 SRAM
A 32 x 82 SRAM

[Diagram of a 32 x 82 SRAM with address decoder and multiplexers]
More SRAMs

See more detailed diagrams at
http://www.ece.gatech.edu/research/labs/ccss/education/CmpE1700/day28/examples.html#ex2
Dynamic RAMs (DRAMs)

- The value kept in a cell is stored as a charge in a capacitor.
  - A single transistor is used to access the stored charge.
- The value can't be kept indefinitely
  - Must be periodically refreshed
  - Charge can be kept for few milliseconds
- To refresh the cell, we read it and then write it back
- Two-level decoding structure is used, that allows an entire row to be refreshed.
DRAM

- To save pins & reduce cost, the same address lines are used for both rows & column addresses.
- Access is slower than SRAMs
  - Typical DRAM access times range from 60 - 110 ns
- Row Access Strobe (RAS):
  - Signal row addressing.
- Column Access Strobe (CAS):
  - Signal column addressing.
- See more detailed diagrams at [http://www.ece.gatech.edu/research/labs/ccss/education/CmpE1700/day28/examples.html#ex2](http://www.ece.gatech.edu/research/labs/ccss/education/CmpE1700/day28/examples.html#ex2)
- **DRAM**

- A single transistor DRAM

- A 4M x 1 DRAM with a 2048 x 2048 array

Diagram:

- **Row decoder 11-to-2048**
  - Input: Address[10–0]
  - Output: 2048 x 2048 array

- **2048 x 2048 array**

- **Column latches**

- **Mux**

- **Dout**

- **Pass transistor**

- **Capacitor**

- **Bit line**

- **Word line**
Error Detection & Correction

- Most computer systems use some sort of error-checking code to detect possible corruption of data
- A collection of methods to detect errors in transmitted or stored data and to correct them
- Involves some form of coding

Detection methods:
- Single Parity
- Multiple Parity

Detection & Correction methods:
- Hamming code
- Other methods
Single Parity

- The simplest form of error detection
- A single added parity bit or a cyclic redundancy check
- Parity can only detect, but not correct errors
- Only odd number of errors can be detected

Parity code:
- The number of 1's in a word is counted.

Odd parity:
- If the number of 1's is odd.

Even parity:
- If the number of 1's is even
Error Detection & Correction

Multiple parity:

- Detect that an error has occurred and also which bits have been inverted
  - That bit should therefore be re-inverted to restore the original data
- The more extra bits are added, the greater the chance that multiple errors will be detectable and correctable
Error Detection & Correction

- Single Error Correction, Double Error Detection (SECDEC)
  - Hamming code: One of the most commonly used
- Other error-correcting codes
  - Allow detection as well as correction of errors
  - More bits are used to encode the data.
Hamming code

- An error-detecting and error-correcting binary code, used in data transmission, that can (a) detect all single-and double-bit errors and (b) correct all single-bit errors.
  - For more details see [Summary of Hamming code properties](http://www.ee.unb.ca/tervo/ee4253/hamming.htm)

- Example:
  - see [http://www.ee.unb.ca/tervo/ee4253/hamming.htm](http://www.ee.unb.ca/tervo/ee4253/hamming.htm)
Buses

**Bus:**
- A set of electrical connections (wires) through which signals (and power) can pass. The signals may be either synchronous or asynchronous depending on the type of bus.
- Within a computer bus signals are usually synchronous, and controlled by the system clock.

**Types of bus in a typical computer:**
- Data bus
- Address bus
- Control Bus

**For a good introduction see:**
- [http://www.chm.bris.ac.uk/~cijpm/CM1/lt6.htm#intro](http://www.chm.bris.ac.uk/~cijpm/CM1/lt6.htm#intro)
Data Bus

- 8, 16, 32, 64, 80 (or more) wires
- For transmission of data
- The wider the bus the more data can be moved around
- Bi-directional
- Parallel
Address Bus

- Governs the amount of memory that a computer can address
  - 16 bit is 64 Kilobytes
  - 20 bits is a Megabyte
  - 24 bits is 16 Megabytes
  - 32 bit is 4 Gigabytes.
- Unidirectional, only the processor can 'address' memory
- Parallel
Control Bus

- Includes all the different wires to:
  - Carry power
  - Earth
  - Clock signals
  - Interrupts
  - Controls for other buses
  - Connections to other processors and chips
  - Logic signals
  - Any other electrical connections at 5V or below
- Consists (mostly) of individual wires
- Serial
Buses

- CPU
- Memory
- I/O

Data
Address
Control
Bus Notation

- A bus usually has its own name
- Either drawn with a double arrow or a thick line
- Assumption:
  - A bus with no label is assumed to be of 32-bits width
  - If the width of the bus differs from the 32-bit, it will be explicitly indicated on the graph.