CSE140: Components and Design Techniques for Digital Systems

Introduction

Instructor: Mohsen Imani

Slides from:
Prof. Tajana Simunic Rosing & Dr. Pietro Mercati
Welcome to CSE 140!

- **Instructor:** Mohsen Imani
- **Email:** moimani@ucsd.edu;
  - please put “CSE140” in the subject line
- **Class:** Tue/Thr 11:00am-1:50pm at PCYNH 122
- **Office Hours:** Tue 3-5pm CSE 2217
- **Discussion sessions:** Tue/Thr 10-11am at PCYNH 122
- **Course website:**
  - [https://cseweb.ucsd.edu/classes/su17_2/cse140-a/index.html](https://cseweb.ucsd.edu/classes/su17_2/cse140-a/index.html)
- **Announcements and online discussion:** [https://piazza.com](https://piazza.com) → SIGN UP SOON !!!!
- **TAs and Tutors:** Office hours listed on the class website/Piazza
Grading

- Grading:
  - Class participation using iClicker: 5%
  - 4 Homeworks: 5% each
  - Midterm: 30%
  - Final: 45%

- Homeworks are out on Tuesdays and are due on the following Tuesday before 10am

- Tue discussion session will go over the HW solution
- Submission via gradescope: [https://gradescope.com/](https://gradescope.com/)
- Only a subset of problems will be graded
- **Late submissions will not be accepted!**
- Homework 1 will be out right after this class
Some Class Policies

• Academic Honesty
  – Studying together in groups is encouraged
  – Turned-in work must be completely your own
  – Both “giver” and “receiver” are equally culpable
  – Cheating on HW/ exams: F in the course.
  – Any instance of cheating will be referred to Academic Integrity Office
Textbooks and Recommended Readings

- **Recommended textbook:**
  - digital design by F. Vahid

- **Other recommended textbooks:**
  - Digital Design by F. Vahid
  - Digital Design & Computer Arch.
    - by David & Sarah Harris
  - Contemporary Logic Design
    - by R. Katz & G. Borriello

- Lecture slides are derived from the slides designed for all three books
Abstraction:
A way to simplify by hiding details from other layers

Layers of abstraction:
- Programs
- Device drivers
- Instructions
- Registers
- Data paths
- Controllers
- Adders
- Memories
- AND gates
- NOT gates
- Amplifiers
- Filters
- Transistors
- Diodes
- Electrons
Why Study Digital Design?

Look “under the hood” of your processors
You become a better programmer when you understand hardware your code runs on

Nvidia Tegra 2 die photo
The Scope of CSE140

- We start with Boolean algebra \( Y = A \) and \( B \)
- End up with the design of a simple CPU
- What’s next? CSE141 – more complex CPU architectures

Nvidia Tegra 2 die photo
• Number representations
  – Analog vs. Digital
  – Digital representations:
    • Binary, Hexadecimal, Octal
• Switches, MOS transistors, Logic gates
  – What is a switch
  – How a transistor operates
  – Logic gates
  – Building larger functions from logic gates
• Universal gates
• Boolean algebra
  – Properties
  – How Boolean algebra can be used to design logic circuits
What Does “Digital” Mean?

- **Analog signal**
  - Infinite possible values
  - Ex: voltage on a wire created by microphone

- **Digital signal**
  - Finite possible values
  - Ex: button pressed on a keypad

Which is analog?
A) Wind speed
B) Radio Signal
C) Clicker response
D) A) & B)
E) All of the above
Encoding Numbers – Base 10 & 2

- Each position represents a quantity; symbol in position means how many of that quantity
  - Base ten (decimal)
    - Ten symbols: 0, 1, 2, ..., 8, and 9
    - More than 9 -- next position
      - So each position power of 10
    - Nothing special about base 10 -- used because we have 10 fingers
  - Base two (binary)
    - Two symbols: 0 and 1
    - More than 1 -- next position
      - So each position power of 2

### Decimal to binary:
- Divide by two
- Report the remainder
- Concatenate the reminders

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>REMINDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>523</td>
<td>1</td>
</tr>
<tr>
<td>261</td>
<td>1</td>
</tr>
<tr>
<td>130</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

523 = 1000001011
Encoding Numbers – Base 10 & 2

- Each position represents a quantity; symbol in position means how many of that quantity
  - Base ten (decimal)
    - Ten symbols: 0, 1, 2, ..., 8, and 9
    - More than 9 -- next position
      - So each position power of 10
    - Nothing special about base 10 -- used because we have 10 fingers
  - Base two (binary)
    - Two symbols: 0 and 1
    - More than 1 -- next position
      - So each position power of 2

Binary to decimal:
- Each position is a power of 2
- Sum up all the powers of 2 where you have a “1”

\[
\begin{align*}
9 & \times 2^9 + 8 & \times 2^8 + 7 & \times 2^7 + 6 & \times 2^6 + 5 & \times 2^5 + 4 & \times 2^4 + 3 & \times 2^3 + 2 & \times 2^2 + 1 & \times 2^1 + 0 & \times 2^0 \\
1 & \times 2^9 + 0 & \times 2^8 + 0 & \times 2^7 + 0 & \times 2^6 + 0 & \times 2^5 + 1 & \times 2^4 + 0 & \times 2^3 + 1 & \times 2^2 + 1 & \times 2^1 + 1 & \times 2^0 \\
\end{align*}
\]

\[
2 + 2 + 2 + 2 = 523
\]
Bases Sixteen & Eight

- **Base sixteen**
  - Used as compact way to write binary numbers
  - Basic digits: 0-9, A-F
  - Known as *hexadecimal*, or just **hex**

- **Base eight**
  - Basic digits: 0-7
  - Known as **octal**

<table>
<thead>
<tr>
<th>hex</th>
<th>binary</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>octal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>
Bases Sixteen & Eight

### Hexadecimal to binary:
- Expand each hexadecimal digit into the corresponding binary string

\[
8 \ A \ F = 1000 \ 1010 \ 1111
\]

### Binary to hexadecimal:
- Pad the binary string with zeros on the left until you have a number of digits multiple of 4
- Group digits 4-by-4 starting from the right and convert into the corresponding hexadecimal symbol

\[
101101 \quad \rightarrow \quad 00101101 \quad \rightarrow \quad 2 \ D
\]
Question

- **AB1** is a hex value. What’s the equivalent value in Octal?

A. 5341
B. 5261
C. AB1
D. 3421
E. None

\[ AB1 = 101010110001 \]
\[ \underline{5261} \]
Question

• **38** is a decimal value. What’s the equivalent value in hex?

A. 16
B. 2A
C. 014
D. 11
E. None

\[ 38 = \overline{100110} = \overline{60100110} \]
\[ \frac{60100110}{2} \]
Combinational circuit building blocks:
Switches & CMOS transistors
CMOS Switches

- CMOS circuit
  - Consists of N and PMOS transistors
  - Both N and PMOS are similar to basic switches

Silicon -- not quite a conductor or insulator: *Semiconductor*
Transistor Circuit Design

- **nMOS:**
  - Turns on when gate is connected to 1
  - When turned on, nMOS passes zeros well, but not ones, so connect source to GND
  - nMOS forms a pull-down network

- **pMOS:**
  - Turns on when gate is connected to 0
  - When turned on, pMOS passes ones well, but not zeros, so connect source to $V_{DD}$
  - pMOS forms a pull-up network

**Note:** Vahid’s textbook shows some circuits with pMOS connected to GND and nMOS to Vdd: this is NOT normally done in practice!
The following is true for CMOS switches:
A. nMOS turns on when gate is connected to logic 1
B. pMOS is an open switch when gate is connect to logic 1
C. All of the above
D. None of the above
Logic gates: CMOS NOT Gate

Y = \overline{A}

<table>
<thead>
<tr>
<th>A</th>
<th>P1</th>
<th>N1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ON</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>OFF</td>
<td>ON</td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS Two Input NAND Gate

\[ Y = \overline{AB} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</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>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>P1</th>
<th>P2</th>
<th>N1</th>
<th>N2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>0</td>
</tr>
</tbody>
</table>
Common Logic Gates

\[ a \downarrow \quad a' \quad \begin{array}{l|l} a \quad \text{NOT} \\ \hline 0 & 1 \\ 1 & 0 \end{array} \quad \begin{array}{l} a \quad \text{BUF} \\ \hline 0 \quad 1 \end{array} \]

\[ (a \cdot b)' \]

\[ (a + b)' \]

\[ ab' + a'b \]

\[ ab + a'b' \]
Boolean algebra

- \( B = \{0, 1\} \)
- Variables represent 0 or 1 only
- Operators return 0 or 1 only
- Basic operators
  - Intersection: \( \cdot \) is logical **AND**: \( a \text{ AND } b \) returns 1 only when \( a=1 \) & \( b=1 \)
  - Union: \( + \) is logical **OR**: \( a \text{ OR } b \) returns 1 if either \( a=1 \) or \( b=1 \) (or both)
  - Complement: ’ is logical **NOT**: NOT \( a \) returns the opposite of \( a \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>AND</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>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>OR</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>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>BUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Derived operators:
  - **NAND**
  - **NOR**
  - **XOR**
  - **XNOR**
Boolean Algebra

X and Y are Boolean variables with X=1, Y=0
What is  X+X+Y?
A. 0
B. 1
C. 2
D. None of the above
BREAK !
Universal Gate: NAND

Any logic function can be implemented using just NAND gates. Boolean algebra’s basic operators are AND, OR and NOT.

Desired NOT Gate

\[ Q = \text{NOT}( A ) \]

NAND Construction

\[ Q = \text{NOT}( A \text{ AND } A ) \]

Desired AND Gate

\[ Q = A \text{ AND } B \]

NAND Construction

\[ Q = \text{NOT}( \text{NOT}( A \text{ AND } B ) \text{ AND } \text{NOT}( A \text{ AND } B ) ) \]

Desired OR Gate

\[ Q = A \text{ OR } B \]

NAND Construction

\[ Q = \text{NOT}( \text{NOT}( A \text{ AND } A ) \text{ AND } \text{NOT}( B \text{ AND } B ) ) \]

Truth Table

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output Q</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>

https://en.wikipedia.org/wiki/NAND_logic
Gate level Implementation

- $F = A + B'C$
Gate level Implementation

- F = (A+B).(C+D)

What’s the minimum number of NAND gates to implement this function?

A. 4  
B. 8  
C. 6  
D. 10
Universal Gate: NOR

Any logic function can be implemented using just **NOR** gates. Boolean algebra needs AND, OR and NOT.

https://en.wikipedia.org/wiki/NOR_logic
### Boolean Axioms & Theorems

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Dual</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>$B = 0 \text{ if } B \neq 1$</td>
<td>$A1' \quad B = 1 \text{ if } B \neq 0$</td>
</tr>
<tr>
<td>A2</td>
<td>$\bar{0} = 1$</td>
<td>$A2' \quad \bar{1} = 0$</td>
</tr>
<tr>
<td>A3</td>
<td>$0 \cdot 0 = 0$</td>
<td>$A3' \quad 1 + 1 = 1$</td>
</tr>
<tr>
<td>A4</td>
<td>$1 \cdot 1 = 1$</td>
<td>$A4' \quad 0 + 0 = 0$</td>
</tr>
<tr>
<td>A5</td>
<td>$0 \cdot 1 = 1 \cdot 0 = 0$</td>
<td>$A5' \quad 1 + 0 = 0 + 1 = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Dual</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>$B \cdot 1 = B$</td>
<td>$T1' \quad B + 0 = B$</td>
</tr>
<tr>
<td>T2</td>
<td>$B \cdot 0 = 0$</td>
<td>$T2' \quad B + 1 = 1$</td>
</tr>
<tr>
<td>T3</td>
<td>$B \cdot B = B$</td>
<td>$T3' \quad B + B = B$</td>
</tr>
<tr>
<td>T4</td>
<td>$\bar{\bar{B}} = B$</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>$B \cdot \bar{B} = 0$</td>
<td>$T5' \quad B + \bar{B} = 1$</td>
</tr>
</tbody>
</table>
Boolean theorems of multiple variables

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Dual</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T6</td>
<td>B • C = C • B</td>
<td>T6'</td>
</tr>
<tr>
<td>T7</td>
<td>(B • C) • D = B • (C • D)</td>
<td>T7'</td>
</tr>
<tr>
<td>T8</td>
<td>(B • C) + B • D = B • (C + D)</td>
<td>T8'</td>
</tr>
<tr>
<td>T9</td>
<td>B • (B + C) = B</td>
<td>T9'</td>
</tr>
<tr>
<td>T10</td>
<td>(B • C) + (B • C) = B</td>
<td>T10'</td>
</tr>
<tr>
<td>T11</td>
<td>(B • C) + (B • C) + (C • D)</td>
<td>T11'</td>
</tr>
<tr>
<td>T12</td>
<td>B_0 • B_1 • B_2... = (B_0 + B_1 + B_2...)</td>
<td>T12'</td>
</tr>
</tbody>
</table>
Boolean Duality

- Derived by replacing • by +, + by •, 0 by 1, and 1 by 0 & leaving variables unchanged

\[ X + Y + ... \iff X \cdot Y \cdot ... \]

- Generalized duality:

\[ f (X_1, X_2, ..., X_n, 0, 1, +, •) \iff f(X_1, X_2, ..., X_n, 1, 0, •, +) \]

- Any theorem that can be proven is also proven for its dual! Note: this is NOT deMorgan’s Law
Covering Theorem Explained

• Covering Theorem: $A^*(A+B) = A + A^*B = A$

Venn Diagrams
Combining Theorem Explained

- Combining Theorem: $AB + AB' = (A + B)(A + B') = A$
Proving theorems with Boolean Algebra

• Using the axioms of Boolean algebra:
  – e.g., prove the consensus theorem: $X \cdot Y + X \cdot Y' = X$

    | Axiom       | Expression                  | Simplified                                      |
    |-------------|-----------------------------|-------------------------------------------------|
    | Distributivity | $X \cdot Y + X \cdot Y' = X \cdot (Y + Y')$ |                                                  |
    | Complementarity | $X \cdot (Y + Y') = X \cdot (1)$  |                                                  |
    | Identity     | $X \cdot (1) = X$          | ✔                                                |

  – e.g., prove the covering theorem: $X + X \cdot Y = X$

    | Axiom       | Expression                  | Simplified                                      |
    |-------------|-----------------------------|-------------------------------------------------|
    | Identity     | $X + X \cdot Y = X \cdot 1 + X \cdot Y$ |                                                  |
    | Distributivity | $X \cdot 1 + X \cdot Y = X \cdot (1 + Y)$ |                                                  |
    | Identity     | $X \cdot (1 + Y) = X \cdot (1)$ |                                                  |
    | Identity     | $X \cdot (1) = X$          | ✔                                                |
Consensus Theorem of 3 Variables

- Consensus Theorem: \[ AB + B'C + AC = AB + B'C \]
Proof of Consensus Theorem with Boolean Algebra

• Consensus Theorem:
  \[-(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z\]
Applying Boolean Algebra Theorems

Which of the following is \( CB+BA+C'A \) equal to?

A. \( AB+AC' \)
B. \( BC+AC' \)
C. \( AB+BC \)
D. None of the above
DeMorgan’s Theorem
(Bubble Pushing)

- $Y = \overline{AB} = \overline{A} + \overline{B}$

- $Y = \overline{A + B} = \overline{A} \cdot \overline{B}$
Example of Transforming Circuits with Bubble Pushing
Implement using only NORs

\[ F = X'Y + Z \]
### Proving Theorems with Perfect Induction

- Using perfect induction = complete the truth table:
  - e.g., de Morgan’s:

\[
(X + Y)' = X' \cdot Y'
\]

NOR is equivalent to AND with inputs complemented

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>X'</th>
<th>Y'</th>
<th>(X + Y)'</th>
<th>X' \cdot Y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
(X \cdot Y)' = X' + Y'
\]

NAND is equivalent to OR with inputs complemented

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>X'</th>
<th>Y'</th>
<th>(X \cdot Y)'</th>
<th>X' + Y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Specifying Logic Problems with Truth Tables

- **Half Adder**: Truth table -> Boolean Eq. -> Logic Circuit

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>carry</th>
<th>sum</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>0</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>
Going from a Truth Table to a Boolean Equation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>carry</th>
<th>sum</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>0</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>

Truth Table

Boolean Equation:

\[
\text{Sum (a, b)} = a' \cdot b + a \cdot b' = a \text{ XOR } b
\]

\[
\text{Carry (a, b)} = a \cdot b
\]
How do we get a Boolean Equation from the Truth Table?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>carry</th>
<th>sum</th>
<th>Boolean Equation</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>1</td>
<td>0</td>
<td>1</td>
<td>Sum ( \text{(a, b)} = a' \cdot b + ab' )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Carry ( a, b) = ab )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Need Some Definitions

- **Complement**: variable with a bar over it or a ‘A’, B’, C’
- **Literal**: variable or its complement A, A’, B, B’, C, C’
- **Implicant**: product of literals ABC, AC, BC
- **Implicate**: sum of literals (A +B +C), (A +C), (B +C)
- **Minterm**: AND that includes all input variables ABC, A’BC, AB’C
- **Maxterm**: OR that includes all input variables (A +B +C), (A’+B +C ), (A’+B’+C)
# Minterms of Two Input Functions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Minterm</th>
<th>Maxterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A’B’</td>
<td>A+B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A’B</td>
<td>A+B’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>AB’</td>
<td>A’+B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>AB</td>
<td>A’+B’</td>
</tr>
</tbody>
</table>
CSE140: Components and Design Techniques for Digital Systems

Canonical representation
Canonical Form -- Sum of Minterms

• Truth tables are too big for numerous inputs
• Use standard form of equation instead
  – Known as **canonical form**
  – Regular algebra: group terms of polynomial by power
    • \( ax^2 + bx + c \)  \((3x^2 + 4x + 2x^2 + 3 + 1 \rightarrow 5x^2 + 4x + 4)\)
  – Boolean algebra: create a sum of **minterms** (or a product of **maxterms**)


Is \( F(a,b) = ab + a' \) in canonical form?

A) True

B) False

\[
\sigma b + \sigma'(b + b') = \text{canonical form}
\]

\[
\sigma b + \sigma'b + \sigma'b' = \text{all inputs}
\]
Sum-of-products Canonical Form

- Also known as disjunctive normal form
- Minterm expansion:

\[
F = A'B'C' + A'BC' + AB'C'
\]

Sum-of-products Canonical Form

- Also known as disjunctive normal form
- Minterm expansion:

\[
F = 001 \quad 011 \quad 101 \quad 110 \quad 111
\]

\[
F = A'B'C + A'BC + AB'C + ABC' + ABC
\]

\[
F' = A'B'C' + A'BC' + AB'C'
\]
Sum-of-products canonical form (cont’d)

- **Product minterm**
  - ANDed product of literals – input combination for which output is 1
  - each variable appears exactly once, true or inverted (but not both)
  - You can write a canonical form in multiple ways

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>minterms</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A’B’C’ m0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A’B’C m1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A’BC’ m2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A’BC m3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB’C’ m4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB’C m5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC’ m6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC m7</td>
<td>1</td>
</tr>
</tbody>
</table>

F in canonical form:

\[
F(A, B, C) = \Sigma m(1,3,5,6,7) = m1 + m3 + m5 + m6 + m7 = A’B’C + A’BC + AB’C + ABC’ + ABC
\]

canonical form ≠ minimal form

\[
F(A, B, C) = (A’B’ + A’B + AB’ + AB)C + ABC’
= ((A’ + A)(B’ + B))C + ABC’
= C + ABC’
= ABC’ + C
= AB + C
\]

short-hand notation for minterms of 3 variables
Sum of Products Canonical Form

I. \( \text{sum}(A, B) = A'B' + AB' = \sum m(1, 2) \)
   \[ = (A + B)(A' + B') = \prod M(0, 3) \]

II. \( \text{carry}(A, B) = AB = \sum m(3) \)
    \[ = 1A + B' (A' + B')(A + B') = \prod M(0, 1, 2) \]
Does the following SOP canonical expression correctly express the above truth table:

\[ Y(A,B) = \Sigma m(2,3) \]

A. Yes  
B. No
Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion
- Implements “zeros” of a function

\[
F = (A + B + C) (A + B' + C) (A' + B + C)
\]

\[
F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')
\]
### Product-of-sums canonical form (cont’d)

- **Sum term (or maxterm)**
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>maxterms</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B+C</td>
<td>M0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A+B+C’</td>
<td>M1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A+B’+C</td>
<td>M2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B’+C’</td>
<td>M3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A’+B+C</td>
<td>M4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’+B+C’</td>
<td>M5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’+B’+C</td>
<td>M6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A’+B’+C’</td>
<td>M7</td>
</tr>
</tbody>
</table>

F in canonical form:

\[
F(A, B, C) = \Pi M(0, 2, 4) = M0 \cdot M2 \cdot M4 = (A + B + C)(A + B’ + C)(A’ + B + C)
\]

canonical form ≠ minimal form

\[
F(A, B, C) = (A + B + C)(A + B’ + C)(A’ + B + C)
\]

\[
= (A + B + C)(A + B’ + C)
\]

\[
(A + B + C)(A’ + B + C)
\]

\[
= (A + C)(B + C)
\]

short-hand notation for maxterms of 3 variables
Summary

• What we reviewed thus far:
  – Number representations
  – Switches, logic gates
  – Universal gates
  – Boolean algebra
  – Using Boolean algebra to simplify Boolean equations
    • There is an easier way!

• What is next:
  – Combinational logic:
    • Minimization (the easier way)
    • Designs of common combinational circuits
      – Adders, comparators, subtractors, multipliers, etc.
Claude Shannon:

- In 1936, Shannon began his graduate studies in electrical engineering at MIT.
- While studying the complicated ad hoc circuits of this analyzer, Shannon designed switching circuits based on Boole’s concepts.
- In 1937, he wrote his master’s degree thesis, *A Symbolic Analysis of Relay and Switching Circuits*,[9]
- A paper from this thesis was published in 1938.[10] In this work, Shannon proved that his switching circuits could be used to simplify the arrangement of the electromechanical relays that were used then in telephone call routing switches. Next, he expanded this concept, proving that these circuits could solve all problems that Boolean algebra could solve. In the last chapter, he presents diagrams of several circuits, including a 4-bit full adder.[9]

- Using this property of electrical switches to implement logic is the fundamental concept that underlies all electronic digital computers.

https://en.wikipedia.org/wiki/Claude_Shannon#Logic_Circuits