Combinational logic

- Number Representations
- Basic logic
  - Boolean algebra, proofs by re-writing, proofs by perfect induction
  - logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . . , minimal set
- Logic realization
  - two-level logic and canonical forms
  - incompletely specified functions
- Simplification
  - uniting theorem
  - grouping of terms in Boolean functions
- Alternate representations of Boolean functions
  - cubes
  - Karnaugh maps

Digital

- Digital = discrete
  - Binary codes (example: BCD)
  - Decimal digits 0-9
  - DNA nucleotides
- Binary codes
  - Represent symbols using binary digits (bits)
- Digital computers:
  - I/O is digital
    - ASCII, decimal, etc.
  - Internal representation is binary
    - Process information in bits

<table>
<thead>
<tr>
<th>Decimal Symbols</th>
<th>BCD Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>
The basics: Binary numbers

- Bases we will use
  - Binary: Base 2
  - Octal: Base 8
  - Hexadecimal: Base 16

- Positional number system
  - 101₂ = 1×2² + 0×2¹ + 1×2⁰
  - 63₈ = 6×8¹ + 3×8⁰
  - A₁₆ = 10×16¹ + 1×16⁰

- Addition and subtraction
  \[
  \begin{array}{c}
  1011 \\
  +1010 \\
  \hline
  10101
  \end{array}
  \]

- Conversion from binary to octal/hex
  - Binary: 10011110001
  - Octal: 10 | 011 | 110 | 001 = 2361₈
  - Hex: 100 | 1111 | 0001 = 4F₁₆

- Conversion from binary to decimal
  - 101₂ = 1×2² + 0×2¹ + 1×2⁰ = 5₁₀
  - 63₈ = 6×8¹ + 3×8⁰ + 4×8⁻¹ = 51.5₁₀
  - A₁₆ = 10×16¹ + 1×16⁰ = 161₁₀
Decimal→ binary/octal/hex conversion

<table>
<thead>
<tr>
<th>Binary</th>
<th></th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=56</td>
<td>56÷2=28</td>
<td>56÷8=7</td>
</tr>
<tr>
<td>28÷2=14</td>
<td>7÷8=0</td>
<td>0</td>
</tr>
<tr>
<td>14÷2=7</td>
<td>7÷2=3</td>
<td>1</td>
</tr>
<tr>
<td>7÷2=3</td>
<td>Q=N/2=56/2=111000/2=11100 remainder 0</td>
<td></td>
</tr>
<tr>
<td>3÷2=1</td>
<td>1÷2=0</td>
<td>1</td>
</tr>
<tr>
<td>1÷2=0</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Why does this work?
  - N=56₁₀=111000₂
  - Q=N/2=56/2=111000/2=11100 remainder 0
  - Each successive divide liberates an LSB

Number systems

- How do we write negative binary numbers?
- Historically: 3 approaches
  - Sign-and-magnitude
  - Ones-complement
  - Twos-complement
- For all 3, the most-significant bit (msb) is the sign digit
  - 0 ● positive
  - 1 ● negative
- Learn twos-complement
  - Simplifies arithmetic
  - Used almost universally
Sign-and-magnitude

- The most-significant bit (msb) is the sign digit
  - 0 ⇔ positive
  - 1 ⇔ negative
- The remaining bits are the number’s magnitude
- Problem 1: Two representations for zero
  - 0 = 0000 and also −0 = 1000
- Problem 2: Arithmetic is cumbersome

<table>
<thead>
<tr>
<th>Add</th>
<th>Subtract</th>
<th>Compare and subtract</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>4 0100</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
<td>− 4 1100</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>= 1 0000</td>
</tr>
</tbody>
</table>

↓

| 4    | 0100     | 4 0100 | 0100 |
| + 3  | + 0011   | − 3   | 1011 |
| 7    | 0111     | = 1   | 1111 |
|      |          | ≠ 1111 | = 0001 |
|      |          | − 1   | 1110 |
|      |          | ≠ 1111 | = 1001 |

Ones-complement

- Negative number: Bitwise complement positive number
  - 0011 ≡ 3_{10}
  - 1100 ≡ −3_{10}
- Solves the arithmetic problem

<table>
<thead>
<tr>
<th>Add</th>
<th>Invert, add, add carry</th>
<th>Invert and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>4 0100</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
<td>− 3 + 1100</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>= 1 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add carry: +1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 0001</td>
</tr>
</tbody>
</table>

- Remaining problem: Two representations for zero
  - 0 = 0000 and also −0 = 1111
Twos-complement

- Negative number: Bitwise complement plus one
  - $0011 = 3_{10}$
  - $1101 = -3_{10}$
- Number wheel
  - Only one zero!
  - msb is the sign digit
    - 0 = positive
    - 1 = negative

**Twos-complement (con’t)**

- Complementing a complement = the original number
- Arithmetic is easy
  - Subtraction = negation and addition
    - Easy to implement in hardware

<table>
<thead>
<tr>
<th>Add</th>
<th>Invert and add</th>
<th>Invert and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 0100</td>
<td>4 0100</td>
<td>-4 1100</td>
</tr>
<tr>
<td>+ 3 + 0011</td>
<td>- 3 + 1101</td>
<td>+ 3 + 0011</td>
</tr>
<tr>
<td>= 7 0111</td>
<td>= l 1 0001</td>
<td>= l 1 1111</td>
</tr>
<tr>
<td></td>
<td>drop carry = 0001</td>
<td></td>
</tr>
</tbody>
</table>
There are 16 possible functions of 2 input variables:

- in general, there are $2^{2^n}$ functions of $n$ inputs

Different functions are easier or harder to implement

- each has a cost associated with the number of switches needed
- 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
- X (F3) and Y (F5): require 0 switches, output is one of inputs
- X' (F12) and Y' (F10): require 2 switches for "inverter" or NOT-gate
- X nor Y (F4) and X nand Y (F14): require 4 switches
- X or Y (F7) and X and Y (F1): require 6 switches
- X = Y (F9) and X ⊕ Y (F6): require 16 switches

thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice
Minimal set of functions

- Can we implement all logic functions from NOT, NOR, and NAND?
  - For example, implementing $X$ and $Y$ is the same as implementing $\neg(X \text{ nand } Y)$
- In fact, we can do it with only NOR or only NAND
  - NOT is just a NAND or a NOR with both inputs tied together

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>$X \text{ nor } Y$</th>
<th>$X \text{ nand } Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- and NAND and NOR are "duals", that is, it's easy to implement one using the other

$$X \text{ nand } Y = \neg(\neg X \text{ nor } \neg Y)$$
$$X \text{ nor } Y = \neg(\neg X \text{ nand } \neg Y)$$

- But let's not move too fast . . .
  - Let's look at the mathematical foundation of logic

An algebraic structure

- An algebraic structure consists of
  - a set of elements $B$
  - binary operations {$+, \cdot$}
  - and a unary operation {$'$}
  - such that the following axioms hold:

1. the set $B$ contains at least two elements: $a, b$
2. closure: $a + b$ is in $B$  \( a \cdot b \) is in $B$
3. commutativity: $a + b = b + a$ \( a \cdot b = b \cdot a \)
4. associativity: $a + (b + c) = (a + b) + c$ \( a \cdot (b \cdot c) = (a \cdot b) \cdot c \)
5. identity: $a + 0 = a$ \( a \cdot 1 = a \)
6. distributivity: $a + (b \cdot c) = (a + b) \cdot (a + c)$ \( a \cdot (b + c) = (a \cdot b) + (a \cdot c) \)
7. complementarity: $a + a' = 1$ \( a \cdot a' = 0 \)
Boolean algebra

- B = {0, 1}
- variables
- + is logical OR, • is logical AND
- ' is logical NOT
- All algebraic axioms hold

Logic functions and Boolean algebra

- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ', +, and •

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X•Y</th>
<th>X'</th>
<th>X'•Y</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

X, Y are Boolean algebra variables

Boolean expression that is true when the variables X and Y have the same value and false, otherwise
Axioms and theorems of Boolean algebra

- **Identity:**
  1. $X + 0 = X$
  2. $X + 1 = 1$
  3. $X + X = X$
  4. $(X')' = X$

- **Null:**
  5. $X + X = X$
  6. $X + Y = Y + X$
  7. $(X + Y) + Z = X + (Y + Z)$

- **Idempotency:**
  1. $X \cdot 1 = X$
  2. $X \cdot 0 = 0$
  3. $X \cdot X = X$
  4. $X \cdot X = 0$

- **Involution:**
  5. $X \cdot 1 = X$
  6. $X \cdot Y = Y \cdot X$
  7. $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

- **Complementarity:**
  8. $X \cdot (Y + Z) = (X \cdot Y) \cdot (X \cdot Z)$
  9. $X \cdot Y + X \cdot Y' = X$
  10. $(X + Y') \cdot Y = X \cdot Y$

- **Commutativity:**
  11. $X + X' = 1$
  12. $X + Y = Y + X$
  13. $(X + Y) + Z = X + (Y + Z)$

- **Associativity:**
  14. $X + 1 = 1$
  15. $X \cdot Y = Y \cdot X$
  16. $(X + Y) + Z = X + (Y + Z)$

- **Distributivity:**
  17. $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$
  18. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$

- **Uniting:**
  19. $X \cdot Y + X \cdot Y' = X$
  20. $(X + Y) \cdot (X + Y') = X$

- **Absorption:**
  21. $(X + Y) + Z = X + (Y + Z)$
  22. $(X \cdot Y) \cdot (X \cdot Z) = X \cdot Z + X' \cdot Y$

- **Factoring:**
  23. $X + X \cdot Y = X$
  24. $(X + X') \cdot Z = (X + Z) \cdot (X' + Y)$
  25. $(X + Y) \cdot (Y + Z) + (X' + Z) = X \cdot Y + X' \cdot Z$

- **Consensus:**
  26. $X + X' = 1$
  27. $(X + Y) \cdot (Y + Z) + (X' + Z) = X \cdot Y + X' \cdot Z$
  28. $(X + Y) \cdot (Y + Z) \cdot (X' + Z) = (X + Y) \cdot (X' + Z)$
Axioms and theorems of Boolean algebra (cont’d)

- de Morgan’s:
  14. \((X + Y + ...)' = X' \cdot Y' \cdot ...\)
  14D. \((X \cdot Y \cdot ...)’ = X’ + Y’ + ...\)
- generalized de Morgan’s:
  15. \(f'(X_1, X_2, ..., X_n, 0, 1, +, \cdot) = f(X_1', X_2', ..., X_n', 1, 0, \cdot, +)\)
- establishes relationship between \(\cdot\) and \(+\)

---

Axioms and theorems of Boolean algebra (cont’d)

- Duality
  - a dual of a Boolean expression is derived by replacing
    \(\cdot\) by \(+\), \(+\) by \(\cdot\), 0 by 1, and 1 by 0, and leaving variables unchanged
  - any theorem that can be proven is thus also proven for its dual!
  - a meta-theorem (a theorem about theorems)
- duality:
  16. \(X + Y + ... \Leftrightarrow X \cdot Y \cdot ...\)
- generalized duality:
  17. \(f (X_1, X_2, ..., X_n, 0, 1, +, \cdot) \Leftrightarrow f(X_1', X_2', ..., X_n', 1, 0, \cdot, +)\)

- Different than deMorgan’s Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions
Using the axioms of Boolean algebra:
- e.g., prove the theorem: \(X \cdot Y + X \cdot Y' = X\)
  - distributivity (8): \(X \cdot Y + X \cdot Y' = X \cdot (Y + Y')\)
  - complementarity (5): \(X \cdot (Y + Y') = X \cdot (1)\)
  - identity (1D): \(X \cdot (1) = X\)

- e.g., prove the theorem: \(X + X \cdot Y = X\)
  - identity (1D): \(X + X \cdot Y = X \cdot 1 + X \cdot Y\)
  - distributivity (8): \(X \cdot 1 + X \cdot Y = X \cdot (1 + Y)\)
  - identity (2): \(X \cdot (1 + Y) = X \cdot (1)\)
  - identity (1D): \(X \cdot (1) = X\)

Proving theorems (rewriting)

Activity

- Prove the following using the laws of Boolean algebra:
  - \((X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z\)
Proving theorems (perfect induction)

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

\[
\begin{array}{c|c|c|c|c|c}
X & Y & X' & Y' & (X + Y)' & X' \cdot Y' \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

NOR is equivalent to AND with inputs complemented

\[
\begin{array}{c|c|c|c|c|c}
X & Y & X' & Y' & (X \cdot Y)' & X' + Y' \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

NAND is equivalent to OR with inputs complemented

A simple example: 1-bit binary adder

- Inputs: A, B, Carry-in
- Outputs: Sum, Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
<th>S</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>1</td>
</tr>
<tr>
<td>0</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>0</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

S = A’ B’ Cin + A’ B Cin’ + A B’ Cin’ + A B Cin

Cout = A’ B Cin + A B’ Cin + A B Cin’ + A B Cin
Apply the theorems to simplify expressions

- The theorems of Boolean algebra can simplify Boolean expressions
  - e.g., full adder’s carry-out function (same rules apply to any function)

\[
\begin{align*}
\text{Cout} & = A' \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin} \\
& = A' \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin} \\
& = A' \cdot B \cdot \text{Cin} + A \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' \\
& = (A' + A) \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' \\
& = (1) \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin} \\
& = B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin} \\
& = B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' \\
& = B \cdot \text{Cin} + A \cdot (B' + B) \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin} \\
& = B \cdot \text{Cin} + A \cdot \text{Cin} + A \cdot B \cdot (\text{Cin}' + \text{Cin}) \\
& = B \cdot \text{Cin} + A \cdot \text{Cin} + A \cdot B \cdot (1) \\
& = B \cdot \text{Cin} + A \cdot \text{Cin} + A \cdot B
\end{align*}
\]

Adding extra terms creates new factoring opportunities

Activity

- Fill in the truth-table for a circuit that checks that a 4-bit number is divisible by 2, 3, or 5

<table>
<thead>
<tr>
<th>X8</th>
<th>X4</th>
<th>X2</th>
<th>X1</th>
<th>By2</th>
<th>By3</th>
<th>By5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>
<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>
<td>0</td>
</tr>
</tbody>
</table>

- Write down Boolean expressions for By2, By3, and By5
Activity

From Boolean expressions to logic gates

- **NOT** X'  \(\overline{X}\)  \(\neg X\)  
  \[ \begin{array}{c|c|c}
  X & Y \\
  \hline
  0 & 1 \\
  1 & 0 \\
  \end{array} \]

- **AND** X \(\cdot Y\)  X\(Y\)  X \(\land Y\)  
  \[ \begin{array}{c|c|c|c}
  x & y & z \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
  \end{array} \]

- **OR** X + Y  X \(\lor Y\)  
  \[ \begin{array}{c|c|c|c}
  x & y & z \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
  \end{array} \]
From Boolean expressions to logic gates (cont’d)

- **NAND**
  
  \[
  X \land Y \rightarrow Z
  \]
  
<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</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>

- **NOR**
  
  \[
  X \lor Y \rightarrow Z
  \]
  
<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **XOR**

  \[
  X \oplus Y \rightarrow Z
  \]
  
<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</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>

  \(X \text{ xor } Y = X \cdot Y' + X' \cdot Y\)
  
  \(X\) or \(Y\) but not both
  
  (“inequality”, “difference”)

- **XNOR**

  \[
  X = Y \rightarrow Z
  \]
  
<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

  \(X \text{ xnor } Y = X \cdot Y' + X' \cdot Y\)
  
  \(X\) and \(Y\) are the same
  
  (“equality”, “coincidence”)

More than one way to map expressions to gates

- e.g., \(Z = A' \cdot B' \cdot (C + D) = (A' \cdot (B' \cdot (C + D)))\)

  \[\begin{array}{c}
  T_2 \\
  T_1
  \end{array}\]

  use of 3-input gate
Waveform view of logic functions

- Just a sideways truth table
  - but note how edges don’t line up exactly
  - it takes time for a gate to switch its output!

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Change in Y takes time to "propagate" through gates

---

Choosing different realizations of a function

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- two-level realization (we don’t count NOT gates)
- multi-level realization (gates with fewer inputs)
- XOR gate (easier to draw but costlier to build)
Which realization is best?

- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - fan-ins (# of gate inputs) are limited in some technologies
- Reduce number of gates
  - fewer gates (and the packages they come in) means smaller circuits
    - directly influences manufacturing costs

Which is the best realization? (cont’d)

- Reduce number of levels of gates
  - fewer level of gates implies reduced signal propagation delays
  - minimum delay configuration typically requires more gates
    - wider, less deep circuits
- How do we explore tradeoffs between increased circuit delay and size?
  - automated tools to generate different solutions
  - logic minimization: reduce number of gates and complexity
  - logic optimization: reduction while trading off against delay
Are all realizations equivalent?

- Under the same input stimuli, the three alternative implementations have almost the same waveform behavior
  - delays are different
  - glitches (hazards) may arise – these could be bad, it depends
  - variations due to differences in number of gate levels and structure
- The three implementations are functionally equivalent

Implementing Boolean functions

- Technology independent
  - canonical forms
  - two-level forms
  - multi-level forms
- Technology choices
  - packages of a few gates
  - regular logic
  - two-level programmable logic
  - multi-level programmable logic
Truth table is the unique signature of a Boolean function
The same truth table can have many gate realizations
Canonical forms
- standard forms for a Boolean expression
- provides a unique algebraic signature

Also known as disjunctive normal form
Also known as minterm expansion

F = \text{001} \quad \text{011} \quad \text{101} \quad \text{110} \quad \text{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 term (or minterm)**
  - ANDed product of literals – input combination for which output is true
  - each variable appears exactly once, true or inverted (but not both)

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

\[
\]

\[
= ABC' + C = AB + C
\]

short-hand notation for minterms of 3 variables

---

**Product-of-sums canonical form**

- Also known as conjunctive normal form
- Also known as maxterm expansion

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

\[
F' = (A + B + C') (A + B' + C') (A' + B + C) (A' + B' + C)
\]
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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B+C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A+B+C’</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A+B’+C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B’+C’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A’+B+C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’+B+C’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’+B’+C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A’+B’+C’</td>
</tr>
</tbody>
</table>

- **F in canonical form:**
  - \( F(A, B, C) = \text{IM}(0,2,4) \)
  - \( = M_0 \cdot M_2 \cdot M_4 \)
  - \( = (A + B + C) (A + B’ + C) (A’ + B + C) \)

- **S-o-P, P-o-S, and de Morgan’s theorem**

  - **Sum-of-products**
    - \( F’ = A’B’C’ + A’BC’ + AB’C’ \)
  
  - **Apply de Morgan’s**
    - \( (F’)’ = (A’B’C’ + A’BC’ + AB’C’)’ \)
    - \( F = (A + B + C) (A + B’ + C) (A’ + B + C) \)

  - **Product-of-sums**
    - \( F’ = (A + B + C’) (A + B’ + C’) (A’ + B + C’) (A’ + B’ + C’) \)
  
  - **Apply de Morgan’s**
    - \( (F’)’ = ( (A + B + C’) (A + B’ + C’) (A’ + B + C’) (A’ + B’ + C’))’ \)
    - \( F = A’B’C + A’BC + AB’C + ABC’ + ABC \)
Waveforms for the four alternatives

- Waveforms are essentially identical
  - except for timing hazards (glitches)
  - delays almost identical (modeled as a delay per level, not type of gate or number of inputs to gate)
Mapping between canonical forms

- Minterm to maxterm conversion
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., \( F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4) \)

- Maxterm to minterm conversion
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., \( F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7) \)

- Minterm expansion of \( F \) to minterm expansion of \( F' \)
  - use minterms whose indices do not appear
  - e.g., \( F(A,B,C) = \Sigma m(1,3,5,6,7) \quad F'(A,B,C) = \Sigma m(0,2,4) \)

- Maxterm expansion of \( F \) to maxterm expansion of \( F' \)
  - use maxterms whose indices do not appear
  - e.g., \( F(A,B,C) = \Pi M(0,2,4) \quad F'(A,B,C) = \Pi M(1,3,5,6,7) \)

Incompletely specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode the decimal digits 0 – 9 in the bit patterns 0000 – 1001

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>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>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>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</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>0</td>
<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
</tbody>
</table>

- off-set of \( W \)
- on-set of \( W \)
- don’t care (DC) set of \( W \)
- these inputs patterns should never be encountered in practice – "don’t care" about associated output values, can be exploited in minimization
Don’t cares and canonical forms

- so far, only represented on-set
- also represent don’t-care-set
- need two of the three sets (on-set, off-set, dc-set)

Canonical representations of the BCD increment by 1 function:

- \[ Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15} \]
- \[ Z = \Sigma [ m(0,2,4,6,8) + d(10,11,12,13,14,15) ] \]
- \[ Z = M_1 \cdot M_3 \cdot M_5 \cdot M_7 \cdot M_9 \cdot D_{10} \cdot D_{11} \cdot D_{12} \cdot D_{13} \cdot D_{14} \cdot D_{15} \]
- \[ Z = \Pi [ M(1,3,5,7,9) \cdot D(10,11,12,13,14,15) ] \]

Finding a minimal sum of products or product of sums realization

- exploit don’t care information in the process

Algebraic simplification

- not an algorithmic/systematic procedure
- how do you know when the minimum realization has been found?

Computer-aided design tools

- precise solutions require very long computation times, especially for functions with many inputs (> 10)
- heuristic methods employed – “educated guesses” to reduce amount of computation and yield good if not best solutions

Hand methods still relevant

- to understand automatic tools and their strengths and weaknesses
- ability to check results (on small examples)
The uniting theorem

- Key tool to simplification: \( A (B' + B) = A \)
- Essence of simplification of two-level logic
  - find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

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

A has a different value in the two rows
- A is eliminated

B has the same value in both on-set rows
- B remains

Boolean cubes

- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"
Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

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

Two faces of size 0 (nodes) combine into a face of size 1 (line)

A varies within face, B does not

ON-set = solid nodes
OFF-set = empty nodes
DC-set = x'd nodes

Three variable example

- Binary full-adder carry-out logic

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

The on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

\[ \text{Cout} = \text{BCin} + \text{AB} + \text{ACin} \]
Higher dimensional cubes

- Sub-cubes of higher dimension than 2

\[ F(A,B,C) = \Sigma m(4,5,6,7) \]

on-set forms a square

i.e., a cube of dimension 2

represents an expression in one variable

i.e., 3 dimensions – 2 dimensions

A is asserted (true) and unchanged

B and C vary

This subcube represents the

literal A

m-dimensional cubes in a n-dimensional
Boolean space

- In a 3-cube (three variables):
  - a 0-cube, i.e., a single node, yields a term in 3 literals
  - a 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  - a 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  - a 3-cube, i.e., a cube of eight nodes, yields a constant term “1”

- In general,
  - an m-subcube within an n-cube (m < n) yields a term
    with n – m literals
Karnaugh maps

- Flat map of Boolean cube
  - wrap-around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - on-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

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

Karnaugh maps (cont’d)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells

13 = 1101 = ABC'D
Adjacencies in Karnaugh maps

- Wrap from first to last column
- Wrap top row to bottom row

Karnaugh map examples

- \( F = \)
- \( \text{Cout} = \)
- \( f(A, B, C) = \Sigma m(0, 4, 5, 7) \)

obtain the complement of the function by covering 0s with subcubes
More Karnaugh map examples

G(A,B,C) = A

F(A,B,C) = \Sigma m(0,4,5,7) = AC + B'C'

F \text{ simply replace 1’s with 0’s and vice versa}
F'(A,B,C) = \Sigma m(1,2,3,6) = B'C' + A'C

Karnaugh map: 4-variable example

- \( F(A,B,C,D) = \Sigma m(0,2,3,5,6,7,8,10,11,14,15) \)

\[ F = C' + A'BD + B'D' \]

find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)
Karnaugh maps: don’t cares

- \( f(A,B,C,D) = \Sigma m(1,3,5,7,9) + d(6,12,13) \)
  - without don’t cares
    - \( f = A'D + B'C'D \)

![Karnaugh map diagram](image)

Karnaugh maps: don’t cares (cont’d)

- \( f(A,B,C,D) = \Sigma m(1,3,5,7,9) + d(6,12,13) \)
  - \( f = A'D + B'C'D \) without don’t cares
  - \( f = A'D + C'D \) with don’t cares

by using don’t care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

don’t cares can be treated as
1s or 0s depending on which is more
advantageous
Activity

- Minimize the function $F = \sum \{0, 2, 7, 8, 14, 15\} + d\{3, 6, 9, 12, 13\}$

Combinational logic summary

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Axioms and theorems of Boolean algebra
  - proofs by re-writing and perfect induction
- Gate logic
  - networks of Boolean functions and their time behavior
- Canonical forms
  - two-level and incompletely specified functions
- Simplification
  - a start at understanding two-level simplification
- Later
  - automation of simplification
  - multi-level logic
  - time behavior
  - hardware description languages
  - design case studies