Assignment: Required Lecture Video

- Why study computer architecture?
- Why is it important?
- Future Computing Architectures

Required Assignment
- **Watch** Prof. Mutlu’s inaugural lecture at ETH and understand it
- [https://www.youtube.com/watch?v=kgiZlSOcGFM](https://www.youtube.com/watch?v=kgiZlSOcGFM)

Optional Assignment – for 1% extra credit
- **Write a 1-page summary** of the lecture and email us
  - What are your key takeaways?
  - What did you learn?
  - What did you like or dislike?
- Submit your summary to [Moodle](https://moodle) – Deadline: April 1
Assignment: Required Readings

- **Last+This week**
  - Combinational Logic
    - P&P Chapter 3 until 3.3  +  H&H Chapter 2

- **This+Next week**
  - Hardware Description Languages and Verilog
    - H&H Chapter 4 until 4.3 and 4.5
  - Sequential Logic
    - P&P Chapter 3.4 until end  +  H&H Chapter 3 in full

- By the end of next week, make sure you are done with
  - **P&P Chapters 1-3**  +  **H&H Chapters 1-4**
Combinational Logic Circuits and Design
What We Will Learn in This Lecture

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational circuits
- Boolean algebra
- How to use Boolean algebra to represent combinational circuits
- Minimizing logic circuits
Now, we know how a MOS transistor works

How do we build logic out of MOS transistors?

We construct basic logic structures out of individual MOS transistors

These logical units are named logic gates
  - They implement simple Boolean functions
Recall: CMOS NOT, NAND, AND Gates

**NOT Gate**

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

**NAND Gate**

<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>

**AND Gate**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</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>
Recall: General CMOS Gate Structure

- The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR
  - The networks may consist of transistors in series or in parallel
  - When transistors are in parallel, the network is **ON** if one of the transistors is **ON**
  - When transistors are in series, the network is **ON** only if all transistors are **ON**

pMOS transistors are used for pull-up
nMOS transistors are used for pull-down
Recall: Digging Deeper: Power Consumption

- **Dynamic Power Consumption**
  - \( C \times V^2 \times f \)
    - \( C \) = capacitance of the circuit (wires and gates)
    - \( V \) = supply voltage
    - \( f \) = charging frequency of the capacitor

- **Static Power Consumption**
  - \( V \times I_{\text{leakage}} \)
    - supply voltage \(*\) leakage current

- **Energy Consumption**
  - **Power \(*\)** Time

- See more in H&H Chapter 1.8
Recall: Common Logic Gates

<table>
<thead>
<tr>
<th>Buffer</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Buffer circuit" /></td>
<td><img src="image" alt="AND circuit" /></td>
<td><img src="image" alt="OR circuit" /></td>
<td><img src="image" alt="XOR circuit" /></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inverter</th>
<th>NAND</th>
<th>NOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Inverter circuit" /></td>
<td><img src="image" alt="NAND circuit" /></td>
<td><img src="image" alt="NOR circuit" /></td>
<td><img src="image" alt="XNOR circuit" /></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>0</td>
<td>1</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>1</td>
</tr>
</tbody>
</table>
Boolean Equations
Recall: Functional Specification

- **Functional specification** of outputs in terms of inputs
- What do we mean by “function”? 
  - Unique **mapping** from input values to output values
  - The **same** input values produce the **same** output value every time
  - **No memory** (does not depend on the history of input values)

**Example (full 1-bit adder – more later):**

\[
S = F(A, B, C_{in}) \\
C_{out} = G(A, B, C_{in})
\]

\[
S = A \oplus B \oplus C_{in} \\
C_{out} = AB + AC_{in} + BC_{in}
\]
Recall: Boolean NOT / AND / OR

\( \overline{A} \) (reads “not A”) is 1 iff A is 0

\[ \begin{array}{c|c}
A & \overline{A} \\
0 & 1 \\
1 & 0 \\
\end{array} \]

\( A \cdot B \) (reads “A and B”) is 1 iff A and B are both 1

\[ \begin{array}{c|c|c}
A & B & A \cdot B \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array} \]

\( A + B \) (reads “A or B”) is 1 iff either A or B is 1

\[ \begin{array}{c|c|c}
A & B & A + B \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array} \]
Recall: Boolean Algebra: Big Picture

- An algebra on 1’s and 0’s
  - with AND, OR, NOT operations

- What you start with
  - **Axioms**: basic things about objects and operations you just assume to be true at the start

- What you derive first
  - **Laws and theorems**: allow you to manipulate Boolean expressions
  - ...also allow us to do some simplification on Boolean expressions

- What you derive later
  - More “sophisticated” properties useful for manipulating digital designs represented in the form of Boolean equations

George Boole, “The Mathematical Analysis of Logic,” 1847.
### Recall: Boolean Algebra: Axioms

#### Formal version

1. **B** contains at least two elements, 0 and 1, such that 0 ≠ 1

2. **Closure** \( a, b \in B, \)
   - (i) \( a + b \in B \)
   - (ii) \( a \cdot b \in B \)

3. **Commutative Laws**: \( a, b \in B, \)
   - (i) \( a + b = b + a \)
   - (ii) \( a \cdot b = b \cdot a \)

4. **Identities**: 0, 1 \( \in B \)
   - (i) \( a + 0 = a \)
   - (ii) \( a \cdot 1 = a \)

5. **Distributive Laws**:
   - (i) \( a \cdot (b + c) = a \cdot b + a \cdot c \)
   - (ii) \( a + (b \cdot c) = (a + b) \cdot (a + c) \)

6. **Complement**:
   - (i) \( a + \overline{a} = 1 \)
   - (ii) \( a \cdot \overline{a} = 0 \)

#### English version

Math formality...

Result of AND, OR stays in set you start with

For primitive AND, OR of 2 inputs, order doesn’t matter

There are identity elements for AND, OR, that give you back what you started with

• distributes over +, just like algebra

…but + distributes over •, also (!!)

There is a complement element; AND/ORing with it gives the identity elm.
Recall: Boolean Algebra: Duality

- Observation
  - All the axioms come in “dual” form
  - Anything true for an expression also true for its dual
  - So any derivation you could make that is true, can be flipped into dual form, and it stays true

- Duality — More formally
  - A dual of a Boolean expression is derived by replacing
    - Every AND operation with... an OR operation
    - Every OR operation with... an AND
    - Every constant 1 with... a constant 0
    - Every constant 0 with... a constant 1
    - But don’t change any of the literals or play with the complements!

Example

\[ a \cdot (b + c) = (a \cdot b) + (a \cdot c) \]
\[ \rightarrow a + (b \cdot c) = (a + b) \cdot (a + c) \]
### Recall: Boolean Algebra: Useful Laws

<table>
<thead>
<tr>
<th>Operations with 0 and 1:</th>
<th>Idempotent Law:</th>
<th>Involution Law:</th>
<th>Laws of Complementarity:</th>
<th>Commutative Law:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. X + 0 = X</td>
<td>3. X + X = X</td>
<td>4. (\overline{X}) = X</td>
<td>5. X + \overline{X} = 1</td>
<td>6. X + Y = Y + X</td>
</tr>
<tr>
<td>2. X + 1 = 1</td>
<td>3D. X • X = X</td>
<td></td>
<td>5D. X • \overline{X} = 0</td>
<td>6D. X • Y = Y • X</td>
</tr>
<tr>
<td></td>
<td>1D. X • 1 = X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2D. X • 0 = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Dual**: AND, OR with identities gives you back the original variable or the identity.
- **Idempotent Law**: AND, OR with self = self.
- **Involution Law**: Double complement = no complement.
- **Laws of Complementarity**: AND, OR with complement gives you an identity.
- **Commutative Law**: Just an axiom...
Recall: Useful Laws (continued)

**Associative Laws:**
7. \((X + Y) + Z = X + (Y + Z)\)
   \[= X + Y + Z\]
   7D. \((X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)\)
   \[= X \cdot Y \cdot Z\]

**Distributive Laws:**
8. \(X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)\)
   8D. \(X + (Y \cdot Z) = (X + Y) \cdot (X + Z)\)

**Simplification Theorems:**
9. \(X \cdot Y = Y \cdot X\)
   9D.
10. \(X + Y = Y + X\)
    10D.
11. \(X + (X \cdot Y) = X\)
    11D.

**Parenthesis order does not matter**

**Axiom**

Actually worth remembering — they show up a lot in real designs…
Boolean Algebra: Proving Things

Proving theorems via axioms of Boolean Algebra:

EX: Prove the theorem:  \( X \cdot Y + X \cdot \bar{Y} = X \)

- Distributive (5)
- Complement (6)
- Identity (4)

EX2: Prove the theorem:  \( X + X \cdot Y = X \)

- Identity (4)
- Distributive (5)
- Identity (2)
- Identity (4)
DeMorgan’s Law: Enabling Transformations

\textit{DeMorgan's Law:}

12. \((X + Y + Z + \cdots) = \overline{X}.\overline{Y}.\overline{Z} \cdots\)

12D. \((X \cdot Y \cdot Z \cdots) = \overline{X} + \overline{Y} + \overline{Z} + \cdots\)

- Think of this as a transformation
  - Let’s say we have:

  \[ F = A + B + C \]

  - Applying DeMorgan’s Law (12), gives us

  \[ F = (A + B + C) = (\overline{A}.\overline{B}.\overline{C}) \]

  At least one of A, B, C is TRUE --> It is not the case that A, B, C are all false
These are conversions between different types of logic functions. They can prove useful if you do not have every type of gate.

\[ A = \overline{(X + Y)} = \overline{XY} \]

NOR is equivalent to AND with inputs complemented.

\[ B = \overline{XY} = \overline{X} + \overline{Y} \]

NAND is equivalent to OR with inputs complemented.
Using Boolean Equations to Represent a Logic Circuit
Sum of Products Form: Key Idea

- Assume we have the truth table of a Boolean Function

- How do we express the function in terms of the inputs in a standard manner?

- Idea: **Sum of Products** form

- Express the truth table as a two-level Boolean expression
  - that contains **all** input variable combinations that result in a 1 output
  - If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
  - $F = \text{OR of all input variable combinations that result in a 1}$
Some Definitions

- **Complement:** variable with a bar over it
  \[\overline{A}, \overline{B}, \overline{C}\]

- **Literal:** variable or its complement
  \[A, \overline{A}, B, \overline{B}, C, \overline{C}\]

- **Implicant:** product (AND) of literals
  \[(A \cdot B \cdot \overline{C}), (\overline{A} \cdot C), (B \cdot \overline{C})\]

- **Minterm:** product (AND) that includes all input variables
  \[(A \cdot B \cdot \overline{C}), (\overline{A} \cdot \overline{B} \cdot C), (\overline{A} \cdot B \cdot \overline{C})\]

- **Maxterm:** sum (OR) that includes all input variables
  \[(A + \overline{B} + \overline{C}), (\overline{A} + B + \overline{C}), (A + B + \overline{C})\]
Two-Level Canonical (Standard) Forms

- **Truth table** is the unique *signature* of a Boolean *function* ...
  - But, it is an expensive representation

- A Boolean function can have many alternative Boolean expressions
  - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)
  - If they all say the same thing, why do we care?
    - Different Boolean expressions lead to different gate realizations

- **Canonical form**: *standard form for a Boolean expression*
  - Provides a unique algebraic signature
Two-Level Canonical Forms

Sum of Products Form (SOP)
Also known as disjunctive normal form or minterm expansion

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>1</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>

\[ F = \overline{A}BC + \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} + ABC \]

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

All Boolean equations can be written in SOP form

Find all the input combinations (minterms) for which the output of the function is TRUE.
SOP Form — Why Does It Work?

- Only the shaded product term — $A\overline{B}C = 1 \cdot \overline{0} \cdot 1$ — will be 1
- No other product terms will “turn on” — they will all be 0
- So if inputs $A \ B \ C$ correspond to a product term in expression,
  - We get $0 + 0 + ... + 1 + ... + 0 + 0 = 1$ for output
- If inputs $A \ B \ C$ do not correspond to any product term in expression
  - We get $0 + 0 + ... + 0 = 0$ for output

$$F = \overline{A}BC + A\overline{B}C + \overline{A}BC + AB\overline{C} + ABC$$
Aside: Notation for SOP

- Standard “shorthand” notation
  - If we agree on the order of the variables in the rows of truth table...
    - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>1</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>

100 = decimal 4 so this is minterm #4, or m4
111 = decimal 7 so this is minterm #7, or m7

f = \[ f = \sum m(3,4,5,6,7) \]

We can write this as a sum of products

Or, we can use a summation notation
Canonical SOP Forms

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ABC = m0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ABC = m1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>ABC = m2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>ABC = 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>

Shorthand Notation for Minterms of 3 Variables

\( F(\text{A}, \text{B}, \text{C}) = \sum m(3,4,5,6,7) \)
\( = m3 + m4 + m5 + m6 + m7 \)

\( F = \)

canonical form ≠ minimal form

2-Level AND/OR Realization

\( F \)
SOP (sum-of-products) leads to two-level logic

Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$
Alternative Canonical Form: POS

A product of sums (POS)

$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)$$

Each sum term represents one of the "zeros" of the function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>1</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>

For the given input, only the shaded sum term will equal 0

$$A + B + C = 0 + \overline{1} + 0$$

Anything ANDed with 0 is 0; Output F will be 0
Consider $A=0$, $B=1$, $C=0$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>1</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>

Input 0 1 0

$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)$

Output 0 1 0 0 1 0 0 1 0

Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is $F = 0$
## Maxterm form:

1. Find truth table rows where F is 0

2. 0 in input col → true literal

3. 1 in input col → complemented literal

4. OR the literals to get a Maxterm

5. AND together all the Maxterms

### Example:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>1</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>

\[ F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C) \]

\[ \overline{F} = A \cdot \overline{B} \cdot C \]
### Canonical POS Forms

#### Product of Sums / Conjunctive Normal Form / Maxterm Expansion

Maxterm shorthand notation for a function of three variables:

<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 ) = M0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( A + B + \overline{C} ) = M1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( A + \overline{B} + C ) = M2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( A + \overline{B} + \overline{C} ) = M3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \overline{A} + B + C ) = M4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( \overline{A} + B + \overline{C} ) = M5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( \overline{A} + \overline{B} + C ) = M6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( \overline{A} + \overline{B} + \overline{C} ) = M7</td>
</tr>
</tbody>
</table>

\[
F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)
\]

\[
\prod M(0, 1, 2)
\]

Note that you form the maxterms around the “zeros” of the function!

This is not the complement of the function!
Useful Conversions

1. **Minterm to Maxterm conversion:**
   rewrite minterm shorthand using maxterm shorthand
   replace minterm indices with the indices not already used
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) = \prod M(0, 1, 2) \)

2. **Maxterm to Minterm conversion:**
   rewrite maxterm shorthand using minterm shorthand
   replace maxterm indices with the indices not already used
   E.g., \( F(A, B, C) = \prod M(0, 1, 2) = \sum m(3, 4, 5, 6, 7) \)

3. **Expansion of \( F \) to expansion of \( \overline{F} \):**
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) \quad \rightarrow \quad \overline{F}(A, B, C) = \sum m(0, 1, 2) \)
   \( = \prod M(0, 1, 2) \quad \rightarrow \quad = \prod M(3, 4, 5, 6, 7) \)

4. **Minterm expansion of \( F \) to Maxterm expansion of \( \overline{F} \):**
   rewrite in Maxterm form, using the same indices as \( F \)
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) \quad \rightarrow \quad \overline{F}(A, B, C) = \prod M(3, 4, 5, 6, 7) \)
   \( = \prod M(0, 1, 2) \quad \rightarrow \quad = \sum m(0, 1, 2) \)
Combinational Building Blocks used in Modern Computers
Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more complex systems.

- Hides the unnecessary gate-level details to emphasize the function of the building block.

- We now look at:
  - Decoder
  - Multiplexer
  - Full adder
  - PLA (Programmable Logic Array)
Decoder

- “Input pattern detector”
- $n$ inputs and $2^n$ outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_3$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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>
</tr>
</tbody>
</table>
Decoder (I)

- \( n \) inputs and \( 2^n \) outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect

\[\begin{align*}
A = 1, B = 0 & \quad 1 \\
A = 1, B = 1 & \quad 0 \\
A = 0, B = 0 & \quad 0 \\
A = 0, B = 1 & \quad 0
\end{align*}\]
The decoder is useful in determining how to interpret a bit pattern

- It could be the address of a row in DRAM, that the processor intends to read from.

- It could be an instruction in the program and the processor has to decide what action to do! (based on instruction opcode)
Multiplexer (MUX), or Selector

- **Selects** one of the $N$ inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called **select**
- Example: 2-to-1 MUX

<table>
<thead>
<tr>
<th>$S$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$Y$</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>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>0</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>

![MUX Diagram](image)
**Multiplexer (MUX), or Selector (II)**

- **Selects** one of the $N$ inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called **select**
- Example: 2-to-1 MUX
Multiplexer (MUX), or Selector (III)

- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

Your task: Draw the schematic for an 4-input (4:1) MUX
- Gate level: as a combination of basic AND, OR, NOT gates
- Module level: As a combination of 2-input (2:1) MUXes
A 4-to-1 Multiplexer
### Full Adder (I)

- **Binary addition**
  - Similar to decimal addition
  - From right to left
  - One column at a time
  - One sum and one carry bit

- Truth table of binary addition on one column of bits within two n-bit operands

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>carry&lt;sub&gt;i&lt;/sub&gt;</th>
<th>carry&lt;sub&gt;i+1&lt;/sub&gt;</th>
<th>$S_i$</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>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>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>
Full Adder (II)

- **Binary addition**
  - N 1-bit additions
  - **SOP of 1-bit addition**

<table>
<thead>
<tr>
<th></th>
<th>a_i</th>
<th>b_i</th>
<th>carry_i</th>
<th>carry_{i+1}</th>
<th>S_i</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>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>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>
<td>1</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>1</td>
<td>1</td>
</tr>
</tbody>
</table>
4-Bit Adder from Full Adders

- Creating a **4-bit adder** out of 1-bit full adders
  - To add two 4-bit binary numbers A and B

```
<table>
<thead>
<tr>
<th></th>
<th>a_3</th>
<th>a_2</th>
<th>a_1</th>
<th>a_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>b_3</td>
<td>b_2</td>
<td>b_1</td>
<td>b_0</td>
</tr>
<tr>
<td>c_4</td>
<td>c_3</td>
<td>c_2</td>
<td>c_1</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
\text{s_3} & \text{s_2} & \text{s_1} & \text{s_0} \\
1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0
\end{array}
\]
```
Adder Design: Ripple Carry Adder

Figure 5.5 32-bit ripple-carry adder
Adder Design: Carry Lookahead Adder
PLA: Recall: From Logic to Gates

- **SOP (sum-of-products) leads to two-level logic**

- **Example:** \( Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) \)
The Programmable Logic Array (PLA)

- The below logic structure is a very common building block for implementing any collection of logic functions one wishes to.
- An array of AND gates followed by an array of OR gates.
- **How do we determine the number of AND gates?**
  - Remember **SOP**: the number of possible minterms.
  - For an n-input logic function, we need a PLA with $2^n$ n-input AND gates.
- **How do we determine the number of OR gates?** The number of output columns in the truth table.
The Programmable Logic Array (PLA)

- **How do we implement a logic function?**
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
  - This is a simple programmable logic

- **Programming a PLA:** we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function

- Have you seen any other type of programmable logic?
  - Yes! An FPGA...
  - An FPGA uses more advanced structures, as we saw in Lecture 3
PLA Example (I)

Read H&H Chapter 5.6.1

Diagram: AND Array connected to OR Array via Implicants.
PLA Example Function (II)

Read H&H Chapter 5.6.1
PLA Example Function (III)

Read H&H Chapter 5.6.1
Implementing a Full Adder Using a PLA

Truth table of a full adder

<table>
<thead>
<tr>
<th>(a_i)</th>
<th>(b_i)</th>
<th>(\text{carry}_i)</th>
<th>(\text{carry}_{i+1})</th>
<th>(S_i)</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>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>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>

This input should not be connected to any outputs

We do not need this output
Logical (Functional) Completeness

- **Any logic function** we wish to implement could be accomplished with a PLA
  - PLA consists of only AND gates, OR gates, and inverters
  - We just have to program connections based on SOP of the intended logic function

- The set of gates \{AND, OR, NOT\} is **logically complete** because we can build a circuit to carry out the specification of **any truth table** we wish, without using any other kind of gate

- NAND is also logically complete. So is NOR.
  - **Your task:** Prove this.
More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

- H&H Chapter 5
  - Will be required reading soon.

- You will benefit greatly by reading the “combinational” parts of Chapter 5 soon.
  - Sections 5.1 and 5.2
Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

![Tri-state Buffer diagram]

<table>
<thead>
<tr>
<th>$E$</th>
<th>$A$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</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>

**Figure 2.40** Tri-state buffer

- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire
Example: Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
  - At any time only the CPU or the memory can place a value on the wire, both not both
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time
Example Design with Tri-State Buffers

CPU

Memory

GateCPU

GateMem

Shared Bus
Another Example

- Processor
- Video
- Ethernet
- Memory
Multiplexer Using Tri-State Buffers

\[ Y = D_0 \overline{S} + D_1 S \]

**Figure 2.56** Multiplexer using tristate buffers
We did not cover the remaining slides. They are for your preparation for the next lecture.
Aside: Logic Using Multiplexers

- Multiplexers can be used as lookup tables to perform logic functions

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

\[ Y = AB \]

**Figure 2.59** 4:1 multiplexer implementation of two-input AND function
Aside: Logic Using Multiplexers (II)

- Multiplexers can be used as lookup tables to perform logic functions

\[ Y = A \oplus B \]
Aside: Logic Using Multiplexers (III)

- Multiplexers can be used as lookup tables to perform logic functions

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

\[ Y = AB + BC + \overline{ABC} \]
Decoders can be combined with OR gates to build logic functions.

**Figure 2.65** Logic function using decoder
Logic Simplification: Karnaugh Maps (K-Maps)
Recall: Full Adder in SOP Form Logic

\[
\begin{array}{c|c|c|c}
 a_i & b_i & carry_i & \text{carry}_{i+1} \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
S_i
\]
Goal: Simplified Full Adder

\[ S = A \oplus B \oplus C_{\text{in}} \]
\[ C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}} \]

<table>
<thead>
<tr>
<th>( C_{\text{in}} )</th>
<th>( A )</th>
<th>( B )</th>
<th>( C_{\text{out}} )</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>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>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>
Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

\[ F = \neg A(A + B) + (B + AA)(A + \neg B) \]

- Can we reduce a given Boolean expression to an equivalent expression with fewer terms?

\[ F = A + B \]

- The goal of logic simplification:
  - Reduce the number of gates/inputs
  - Reduce implementation cost

A basis for what the automated design tools are doing today
Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

**Key Tool: The Uniting Theorem** — \( F = A\overline{B} + AB \)

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

\[ F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A \]

**Essence of Simplification:**
Find two element subsets of the ON-set where only one variable changes its value. This single varying variable can be eliminated!

\[ G = \overline{A\overline{B}} + AB = (\overline{A} + A)\overline{B} = \overline{B} \]

- A's value changes within the ON-set rows
  - \( \rightarrow A \) is eliminated, \( B \) remains
- B's value stays the same within the ON-set rows
  - \( \rightarrow B \) is eliminated, \( A \) remains
Complex Cases

One example

\[ Cout = \overline{ABC} + \overline{ABC} + ABC \]

Problem

- Easy to see how to apply Uniting Theorem...
- Hard to know if you applied it in all the right places...
- ...especially in a function of many more variables

Question

- Is there an easier way to find potential simplifications?
- i.e., potential applications of Uniting Theorem...?

Answer

- Need an intrinsically geometric representation for Boolean f( )
- Something we can draw, see...
Karnaugh Map

- Karnaugh Map (K-map) method
  - K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
  - Physical adjacency $\leftrightarrow$ Logical adjacency

**Numbering Scheme:** 00, 01, 11, 10 is called a “Gray Code” — only a single bit (variable) changes from one code word and the next code word.

2-variable K-map

3-variable K-map

4-variable K-map
Karnaugh Map Methods

K-map adjacencies go “around the edges”
Wrap around from first to last column
Wrap around from top row to bottom row
K-map Cover - 4 Input Variables

Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

\[
F(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)
\]

\[
F = A + \overline{B}\overline{D} + B\overline{C}D
\]
Logic Minimization Using K-Maps

- Very simple guideline:
  - Circle all the rectangular blocks of 1’s in the map, using the fewest possible number of circles
    - Each circle should be as large as possible
  - Read off the implicants that were circled

- More formally:
  - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
  - Each circle on the K-map represents an implicant
  - The largest possible circles are prime implicants
K-map Rules

- **What can be legally combined (circled) in the K-map?**
  - Rectangular groups of size $2^k$ for any integer $k$
  - Each cell has the same value (1, for now)
  - All values must be adjacent
    - Wrap-around edge is okay

- **How does a group become a term in an expression?**
  - Determine which literals are constant, and which vary across group
  - Eliminate varying literals, then AND the constant literals
    - constant 1 → use $x$, constant 0 → use $\overline{x}$

- **What is a good solution?**
  - Biggest groupings → eliminate more variables (literals) in each term
  - Fewest groupings → fewer terms (gates) all together
  - OR together all AND terms you create from individual groups
K-map Example: Two-bit Comparator

Design Approach:
Write a 4-Variable K-map for each of the 3 output functions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>0</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>1</td>
</tr>
<tr>
<td>1</td>
<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>1</td>
<td>1</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>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>1</td>
<td>0</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>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
K-map Example: Two-bit Comparator (2)

K-map for F1

\[ A'B'C'D' + A'BC'D + ABCD + AB'CD' \]

F1 =
K-map Example: Two-bit Comparator (3)

K-map for $F_2$

$$F_2 = \quad \text{? (Exercise for you)}$$
K-maps with “Don’t Care”

- Don’t Care really means *I don’t care what my circuit outputs if this appears as input*
- You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better *simplify* the circuit

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I can pick 00, 01, 10, 11 independently of below
I can pick 00, 01, 10, 11 independently of above
Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
  - Encode decimal digits 0 - 9 with bit patterns $0000_2$ — $1001_2$
  - When **incremented**, the decimal sequence is 0, 1, ..., 8, 9, 0, 1

![Truth Table](image)

These input patterns should never be encountered in practice (hey -- it’s a BCD number!)
So, associated output values are “Don’t Cares”
K-map for BCD Increment Function

\[ Z \text{ (without don’t cares) } = \]

\[ Z \text{ (with don’t cares) } = \]
K-map Summary

- **Karnaugh maps** as a formal systematic approach for logic simplification

- 2-, 3-, 4-variable K-maps

- K-maps with “Don’t Care” outputs

- H&H Section 2.7