Combinational Logic Design

- **Given:** description of circuit behavior
  - Word problem, or truth table
- **Goal:** efficient circuit implementation
  - Usually most important: fewest gates and wires
  - Secondarily: reduce number of levels (propagation delay)
- **Kinds of techniques**
  - Up to 6 inputs: pencil and paper approaches
  - Large but structured: split into repeated pieces
  - Large and unstructured: computer algorithm
**DNF / SOP**

- An input or its negation is called a **literal**
  - E.g.: \( a, \neg b \)
- An AND of literals is a **product term** or **cube**
  - E.g.: \( (a \land c), (a \land \neg b), (\neg a \land \neg b \land \neg c), c \)
- An OR of product terms is a **sum of products** (SOP), or in **disjunctive normal form** (DNF)
  - E.g.: \( (a \land b) \lor (a \land c) \)
  - (Dual: **product of sums** (POS), or **conjunctive normal form** (CNF))

---

**Inefficiency of Straight DNF**

- Consider another example:

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Result: \( (\neg a \land b) \lor (a \land b) \)

- By algebra, can simplify back to "b"
  - Factor, \( (\neg a \land a) = 1 \), \( 1 \land b = b \)
- Can we recognize these patterns earlier?
Karnaugh Map Idea

- Write truth table entries in an array
- Product terms represented by certain rectangles
- Visually, find small number of rectangles to cover 1 bits
  - OK to cover more than once, combine with OR
  - Fewer rectangles = smaller circuit

2-variable "Karnaugh Map"

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-variable “Karnaugh Map” example

Result:
\[ \overline{a} \mid b \]

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

Extending to 3 and 4 Variables

- Put two variables on a side
  - Weird order: 00 01 11 10
  - “Gray Code”: change only one bit at a time
- Rectangles can enclose 1, 2, 4, or 8 entries
  - Bigger is better
- Rectangles can wrap around the edges
  - 00 is adjacent to 10
4-variable Karnaugh Map Example

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ ab = (a \& \!b) | (a \& d) | (\!a \& b \& \!c \& \!d) \]

Extending to 5 and 6 Variables

- 2D is no longer enough
  - No way to order 3 variables to capture 12 adjacencies
- Approach: stacking
  - Make 2 (for 5 inputs) or 4 (for 6 inputs) 4-input Karnaugh maps
  - Corresponding entries are "on top of" each other
  - Rectangles become 3D
  - Usually still drawn as 2D
  - With 6, more possibilities for wrapping too
5-variable Karnaugh Map Example

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

Karnaugh Map Tips: Overlap is Good

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Karnaugh Map Tips: No 3s

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**cd =**

### Karnaugh Map Tips: Wrap Around

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**cd =**

!a & !c
Don’t Cares

- Some results don’t matter
  - Domain of function is a subset of all n-bit strings
  - Unused bit patterns in encodings
  - Bits sometimes ignored by other circuits
- "Don’t care" value could be 0 or 1
  - Usually denoted by X
- Don’t-cares allow designs to be simpler
  - Choose the value that allows a simpler circuit
- In early CPUs, led to undocumented instructions
  - Example: x86 ASL vs. SHL
  - On modern CPUs, more error checking

Karnaugh Map Tips: Don’t Cares

\[
\begin{array}{c|cccc}
ab & 00 & 10 & 11 & 01 \\
\hline
00 & x & 0 & x & 1 \\
10 & 0 & x & x & x \\
11 & x & x & x & x \\
01 & 1 & x & x & x \\
\end{array}
\]
Dual (POS) Karnaugh Maps

- Pretend 0s are 1s
- And vice-versa
- Negate final result

Case Study of a Simple Logic Design: Seven Segment Display

- Chip to drive digital display

<table>
<thead>
<tr>
<th>B3</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Case Study (cont.)

- Implement L4:

  Boolean function for L4
**ASIC vs. Programmable Logic Device**

- PLDs programmable logic devices
  - Simple PLD (SPLD)
    - Programmable logic array (PLA)
    - Programmable array logic (PAL)
  - Complex PLD (CPLD)
    - Field-programmable gate arrays (FPGA)
- ASICs
  - Application specific integrated circuits

---

**PALs and PLAs**

Pre-fabricated building block of many AND/OR gates (or NOR, NAND)
"Personalized" by making or breaking connections among the gates

*Programmable Array Block Diagram for Sum of Products Form*

Difference between PLA and PAL:
- **PLA:** Both AND plane and OR plane are programmable.
- **PAL:** Only AND plane is programmable, while OR plane is fixed.
Example of PALs and PLAs

All possible connections are available before programming

Unwanted connections are "blown"

Alternative Representations

Short-hand notation so we don’t have to draw all the wires!

Notation for implementing

\[ F_0 = A B + A' B' \]
\[ F_1 = C D' + C' D \]
Automated Methods

- Karnaugh maps don't scale well beyond 6 inputs
- Good job for a computer!
- Quine-McCluskey algorithm
  - Tabular analog to Karnaugh maps
  - Optimal, but suffers from exponential blowup
- Heuristic methods like "espresso"
  - First, greedily achieve coverage
  - Then, opportunistically improve
  - No optimality guarantee, but good scalability
- Now a standard part of CAD systems
  - Like compilers for software