Implementation: Two-level Logic

- **Sum-of-products**
  - AND gates to form product terms (minterms)
  - OR gate to form sum

- **Product-of-sums**
  - OR gates to form sum terms (maxterms)
  - AND gates to form product
Two-level Logic using NAND Gates

- Replace minterm AND gates with NAND gates.
- Place compensating inversion at inputs of OR gate.

OR gate with inverted inputs is a NAND gate
- de Morgan's: \( A' + B' = (A \cdot B)' \)
- Two-level NAND-NAND network
  - Inverted inputs are not counted
  - In a typical circuit, inversion is done once and signal distributed
Two-level Logic using NOR Gates

- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate

AND gate with inverted inputs is a NOR gate
- de Morgan’s: \( A' \cdot B' = (A + B)' \)
- Two-level NOR-NOR network
  - Inverted inputs are not counted
  - In a typical circuit, inversion is done once and signal distributed
Two-level Logic using NAND / NOR

- NAND-NAND and NOR-NOR networks
  - de Morgan's law:  
    \[(A + B)' = A' \cdot B'\]
    \[(A \cdot B)' = A' + B'\]
  - written differently:  
    \[A + B = (A' \cdot B')'\]
    \[(A \cdot B) = (A' + B')'\]
- In other words —
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented inputs
  - NAND is the same as OR with complemented inputs
  - NOR is the same as AND with complemented inputs

Conversion Between Forms

- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - Introduce appropriate inversions ("bubbles")
  - Each introduced "bubble" must be matched by a corresponding "bubble"
    - Conservation of inversions
    - Do not alter logic function
- Example: AND/OR to NAND/NAND

[Diagrams showing logic gates and connections]
### Multi-level Logic

- $x = A D F + A E F + B D F + B E F + C D F + C E F + G$
  - Reduced sum-of-products form – already simplified
  - 6 x 3-input AND gates + 1 x 7-input OR gate (may not exist!)
  - 25 wires (19 literals plus 6 internal wires)
- $x = (A + B + C) (D + E) F + G$
  - Factored form – not written as two-level S-o-P
  - 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
  - 10 wires (7 literals plus 3 internal wires)

---

### Conversion: Multi-level to NAND

- $F = A (B + C D) + B C'$
  - Original AND-OR network
  - Introduction and conservation of bubbles
  - Redrawn in terms of conventional NAND gates
**Conversion: Multi-level to NORs**

- \( F = A (B + C D) + B C' \)

**AND-OR-Invert Gates**

- AOI function: three stages of logic—AND, OR, Invert
  - Multiple gates "packaged" as a single circuit block
Conversion to AOI Forms

- General procedure to place in AOI form
  - Compute complement of the function in sum-of-products form
  - By grouping the 0s in the Karnaugh map
- Example: XOR implementation—$A \text{ xor } B = A' B + A B'$
  - AOI form: $F = (A' B' + A B')'$

Examples of using AOI gates

- Example:
  - $F = B \ C' + A \ C' + A \ B$
  - $F' = A' \ B' + A' \ C + B' \ C$
  - Implemented by 2-input 3-stack AOI gate

  - $F = (A + B) (A + C') (B + C')$
  - $F' = (B' + C) (A' + C) (A' + B')$
  - Implemented by 2-input 3-stack OAI gate

- Example: 4-bit equality function
  - $Z = (A0B0+A0B0')(A1B1+A1'B1')(A2B2+A2'B2')(A3B3+A3'B3')$
  - each implemented in a single 2x2 AOI gate
Time Behavior

- Waveforms
  - Visualization of values carried on signal wires over time
  - Useful in explaining sequences of events (changes in value)
- Simulation tools are used to create these waveforms
  - Input to the simulator includes gates and their connections
  - Input stimulus, that is, input signal waveforms
- Some terms
  - Gate delay—time for change at input to cause change at output
    - Min delay—typical/nominal delay—max delay
    - Careful designers design for the worst case
  - Rise time—time for output to transition from low to high voltage
  - Fall time—time for output to transition from high to low voltage
  - Pulse width—time an output stays high or low between changes

Hazards/Glitches

- Hazards/glitches: unwanted switching at the outputs
  - Occur when different paths through circuit have different propagation delays
    - As in pulse shaping circuits we just analyzed
  - Dangerous if logic causes an action while output is unstable
    - May need to guarantee absence of glitches
- Usual solutions
  - 1) Wait until signals are stable (by using a clock): preferable (easiest to design when there is a clock – synchronous design)
  - 2) Design hazard-free circuits: sometimes necessary (clock not used – asynchronous design)
Types of Hazards

- Static 1-hazard
  - Input change causes output to go from 1 to 0 to 1

- Static 0-hazard
  - Input change causes output to go from 0 to 1 to 0

- Dynamic hazards
  - Input change causes a double change from 0 to 1 to 0 to 1 OR from 1 to 0 to 1 to 0

Static Hazards

- Due to a literal and its complement momentarily taking on the same value
  - Thru different paths with different delays and reconverging
  - May cause an output that should have stayed at the same value to momentarily take on the wrong value

- Example:
Dynamic Hazards

- Due to the same versions of a literal taking on opposite values
  - Thru different paths with different delays and reconverging
- May cause an output that was to change value to change 3 times instead of once
- Example:

![Dynamic Hazards Diagram]

Making Connections

- Direct point-to-point connections between gates
  - Wires we've seen so far
- Route one of many inputs to a single output --- *multiplexer*
- Route a single input to one of many outputs --- *demultiplexer*

![Making Connections Diagram]
Mux and Demux

- Switch implementation of multiplexers and demultiplexers
  - Can be composed to make arbitrary size switching networks
  - Used to implement multiple-source/multiple-destination interconnections

![Diagram of Mux and Demux](image)

Multiplexers/Selectors

- Multiplexers/Selectors: general concept
  - $2^n$ data inputs, $n$ control inputs (called "selects"), 1 output
  - Used to connect $2^n$ points to a single point
  - Control signal pattern forms binary index of input connected to output

\[
Z = A' I_0 + A I_1
\]

<table>
<thead>
<tr>
<th>A</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I_0</td>
</tr>
<tr>
<td>1</td>
<td>I_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I_1</th>
<th>I_0</th>
<th>A</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>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>

Two alternative forms for a 2:1 Mux truth table
Multiplexers/Selectors (cont'd)

- 2:1 mux: \( Z = A' I0 + A I1 \)
- 4:1 mux: \( Z = A' B' I0 + A' B I1 + A B' I2 + A B I3 \)
- 8:1 mux: \( Z = A'B'C'I0 + A'B'CI1 + A'BC'I2 + A'BCI3 + AB'C'I4 + AB'CI5 + ABC'I6 + ABCI7 \)
- In general, \( Z = \sum_{k=0}^{2^n-1} (m_k I_k) \)
  - in minterm shorthand form for a \( 2^n:1 \) Mux

Gate Level Implementation

- 2:1 mux

- 4:1 mux
Cascading Multiplexers

- Large multiplexers implemented by cascading smaller ones

Control signals B and C simultaneously choose one of I0, I1, I2, I3 and one of I4, I5, I6, I7.

Control signal A chooses which of the upper or lower mux's output to gate to Z.

Multiplexers as Logic

- 2^n:1 multiplexer implements any function of n variables
  - With the variables used as control inputs and
  - Data inputs tied to 0 or 1
  - In essence, a lookup table

Example:
- \( F(A,B,C) = m0 + m2 + m6 + m7 \)
  - \( = A'B'C' + A'BC' + ABC' + ABC \)
  - \( = A'B'(C') + A'B(C') + AB'(0) + AB(1) \)
Multiplexers as Logic

- \(2^{n-1}:1\) mux can implement any function of \(n\) variables
  - With \(n-1\) variables used as control inputs and
  - Data inputs tied to the last variable or its complement
- Example:
  - \(F(A,B,C) = m_0 + m_2 + m_6 + m_7 = A'B'C' + A'BC' + ABC' + ABC\)
  - \(= A'B'(C) + A'B(C') + AB'(0) + AB(1)\)

Demultiplexers/Decoders

- Decoders/demultiplexers: general concept
  - Single data input, \(n\) control inputs, \(2^n\) outputs
  - Control inputs (called "selects" (S)) represent binary index of output to which the input is connected
  - Data input usually called "enable" (G)

1:2 Decoder:
- \(O_0 = G \cdot S'\)
- \(O_1 = G \cdot S\)

2:4 Decoder:
- \(O_0 = G \cdot S_1' \cdot S_0'\)
- \(O_1 = G \cdot S_1' \cdot S_0\)
- \(O_2 = G \cdot S_1 \cdot S_0'\)
- \(O_3 = G \cdot S_1 \cdot S_0\)

3:8 Decoder:
- \(O_0 = G \cdot S_2' \cdot S_1' \cdot S_0'\)
- \(O_1 = G \cdot S_2' \cdot S_1' \cdot S_0\)
- \(O_2 = G \cdot S_2' \cdot S_1 \cdot S_0'\)
- \(O_3 = G \cdot S_2' \cdot S_1 \cdot S_0\)
- \(O_4 = G \cdot S_2 \cdot S_1' \cdot S_0'\)
- \(O_5 = G \cdot S_2 \cdot S_1' \cdot S_0\)
- \(O_6 = G \cdot S_2 \cdot S_1 \cdot S_0'\)
- \(O_7 = G \cdot S_2 \cdot S_1 \cdot S_0\)
Implementation of Demultiplexers

1:2 Decoders

2:4 Decoders

Demultiplexers as Logic

- n:2^n decoder implements any function of n variables
  - With the variables used as control inputs
  - Enable inputs tied to 1 and
  - Appropriate minterms summed to form the function
Demultiplexers as Logic

- $F_1 = \overline{A} B C' D + \overline{A} B' C D + A B C D$
- $F_2 = A B C' D' + A B C$
- $F_3 = (A' + B' + C' + D')$

Programmable Logic Arrays

- Pre-fabricated building block of many AND/OR gates
  - Actually NOR or NAND
  - "Personalized" by making or breaking connections among gates
  - Programmable array block diagram for sum of products form
Enabling Concept

- Shared product terms among outputs

example:

\[ F_0 = A + B' \quad C' \]
\[ F_1 = A \quad C' + A \quad B \]
\[ F_2 = B' \quad C' + A \quad B \]
\[ F_3 = B' \quad C + A \]

personality matrix

<table>
<thead>
<tr>
<th>product term</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>AB</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B'C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>AC'</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>B'C'</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>–</td>
</tr>
</tbody>
</table>

input side:

1 = uncomplemented in term
0 = complemented in term
– = does not participate

output side:

1 = term connected to output
0 = no connection to output

Before Programming

- All possible connections available before "programming"
- In reality, all AND and OR gates are NANDs
After Programming

- Unwanted connections are "blown"
  - Fuse (normally connected, break unwanted ones)
  - Anti-fuse (normally disconnected, make wanted connections)

Alternate Representation

- Short-hand notation--don't have to draw all the wires
  - X Signifies a connection is present and perpendicular signal is an input to gate

notation for implementing

\[ F_0 = A \cdot B + A' \cdot B' \]
\[ F_1 = C \cdot D' + C' \cdot D \]
Example

- Multiple functions of A, B, C
  - \(F_1 = A \cdot B \cdot C\)
  - \(F_2 = A + B + C\)
  - \(F_3 = A' \cdot B' \cdot C'\)
  - \(F_4 = A' + B' + C'\)
  - \(F_5 = A \text{xor} B \text{xor} C\)
  - \(F_6 = A \text{xnor} B \text{xnor} C\)

PALs and PLAs

- Programmable logic array (PLA)
  - What we've seen so far
  - Unconstrained fully-general AND and OR arrays
- Programmable array logic (PAL)
  - Constrained topology of the OR array
  - Faster and smaller OR plane
### PALs and PLAs: Design Example

#### BCD to Gray code converter

**minimized functions:**

- \( W = A + B \overline{D} + B \overline{C} \)
- \( X = B \overline{C} \)
- \( Y = B + C \)
- \( Z = A'B'C'D + B \overline{C} D + A \overline{D}' + B' \overline{C} D' \)

#### Code converter: programmed PLA

**minimized functions:**

- \( W = A + B \overline{D} + B \overline{C} \)
- \( X = B \overline{C} \)
- \( Y = B + C \)
- \( Z = A'B'CD + B \overline{C} D + A \overline{D}' + B' \overline{C} D' \)

not a particularly good candidate for PAL/PLA implementation since no terms are shared among outputs

however, much more compact and regular implementation when compared with discrete AND and OR gates
PALs and PLAs: Design Example

- Code converter: programmed PAL

4 product terms per each OR gate

PALs and PLAs: Design Example

- Code converter: NAND gate implementation
  - Loss of regularity, harder to understand
  - Harder to make changes
Read-only Memories

- Two dimensional array of 1s and 0s
  - Entry (row) is called a "word"
  - Width of row = word-size
  - Index is called an "address"
  - Address is input
  - Selected word is output

![Decoder Diagram]

- Block diagram of a decoder
- Bit lines (normally pulled to 1 through resistor – selectively connected to 0 by word line controlled switches)
- Word lines (only one is active – decoder is just right for this)

ROMs and Combinational Logic

- Combinational logic implementation (two-level canonical form) using a ROM

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F0</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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ROM block diagram

- 8 words x 4 bits/word
- A B C address
- F0F1F2F3 outputs

Truth table
ROM Structure

- Similar to a PLA structure but with a fully decoded AND array
  - Completely flexible OR array (unlike PAL)

![Diagram of ROM structure]

ROM vs. PLA

- ROM approach advantageous when
  - Design time is short (no need to minimize output functions)
  - Most input combinations are needed (e.g., code converters)
  - Little sharing of product terms among output functions

- ROM problems
  - Size doubles for each additional input
  - Can't exploit don't cares

- PLA approach advantageous when
  - Design tools are available for multi-output minimization
  - There are relatively few unique minterm combinations
  - Many minterms are shared among the output functions

- PAL problems
  - Constrained fan-ins on OR plane
Advantages / Disadvantages

- ROM – full AND plane, general OR plane
  - Cheap (high-volume component)
  - Can implement any function of n inputs
  - Medium speed
- PAL – programmable AND plane, fixed OR plane
  - Intermediate cost
  - Can implement functions limited by number of terms
  - High speed (only one programmable plane that is much smaller than ROM's decoder)
- PLA – programmable AND and OR planes
  - Most expensive (most complex in design, need more sophisticated tools)
  - Can implement any function up to a product term limit
  - Slow (two programmable planes)

Structures for Multi-level Logic

- Difficult to devise a regular structure for arbitrary connections between a large set of different types of gates
  - Efficiency/speed concerns for such a structure
  - Xilinx field programmable gate arrays (FPGAs) are just such programmable multi-level structures
    - Programmable multiplexers for wiring
    - Lookup tables for logic functions (programming fills in the table)
    - Multi-purpose cells (utilization is the big issue)
- Use multiple levels of PALs/PLAs/ROMs
  - Output intermediate result
  - Make it an input to be used in further logic
Combinational Design Case Studies

- General design procedure
- Examples
  - Calendar subsystem
  - BCD to 7-segment display controller
  - Process line controller
  - Logical function unit
- Arithmetic
  - Integer representations
  - Addition/subtraction
  - Arithmetic/logic units

General Design Procedure

- Understand the Problem
  - What is the circuit supposed to do?
  - Write down inputs (data, control) and outputs
  - Draw block diagram or other picture
- Formulate the Problem using a Suitable Design Representation
  - Truth table or waveform diagram are typical
  - May require encoding of symbolic inputs and outputs
- Choose Implementation Target
  - ROM, PAL, PLA
  - Mux, decoder and OR-gate
  - Discrete gates
- Follow Implementation Procedure
  - K-maps for two-level, multi-level
  - Design tools and hardware description language (e.g., Verilog)
Calendar Subsystem

- Determine number of days in a month (to control watch display)
  - Used in controlling the display of a wrist-watch LCD screen
- Inputs: month, leap year flag
- Outputs: number of days
- Use software implementation to help understand the problem

```c
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1)
                then return (29)
                else return (28);
        case 3: return (31);
        case 4: return (30);
        case 5: return (31);
        case 6: return (30);
        case 7: return (31);
        case 8: return (31);
        case 9: return (30);
        case 10: return (31);
        case 11: return (30);
        case 12: return (31);
        default: return (0);
    }
}
```

Formalize the Problem

- Encoding:
  - Binary number for month: 4 bits
  - 4 wires for 28, 29, 30, and 31
    - one-hot – only one true at any time
- Block diagram:

<table>
<thead>
<tr>
<th>month</th>
<th>leap</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>111-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Perform Mapping

- Discrete gates
  - $28 = m_8' m_4' m_2 m_1' \text{ leap}$
  - $29 = m_8' m_4' m_2 m_1' \text{ leap}$
  - $30 = m_8' m_4 m_1' + m_8 m_1$
  - $31 = m_8' m_1 + m_8 m_1'$

- Can translate to S-o-P or P-o-S

<table>
<thead>
<tr>
<th>month</th>
<th>leap</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>111-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

BCD to 7-segment display

- Understanding the problem
  - Input is a 4 bit bcd digit (A, B, C, D)
  - Output is the control signals for the display (7 outputs C0 – C6)

- Block diagram
Formalize the problem

- Truth table
  - Show don't cares
- Choose implementation target
  - If ROM, we are done
  - Don't cares imply PAL/PLA may be attractive
- Follow implementation procedure
  - Minimization using K-maps

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</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>
<td>1</td>
<td>1</td>
<td>1</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>1</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>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Implementation as Minimized S-o-P

- 15 unique product terms when minimized individually

C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C' D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C
**Implementation as Minimized S-o-P**

- Can do better
  - 9 unique product terms (instead of 15)
  - Share terms among outputs
  - Each output not necessarily in minimized form

![ PLA implementation ]

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

<table>
<thead>
<tr>
<th>C0 = A + B D + C + B' D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 = C' D' + C D + B'</td>
</tr>
<tr>
<td>C2 = B + C' + D</td>
</tr>
<tr>
<td>C3 = B' D' + C D' + B C'</td>
</tr>
<tr>
<td>C4 = B' D' + C D'</td>
</tr>
<tr>
<td>C5 = A + C' D' + B D' + B C</td>
</tr>
<tr>
<td>C6 = A + C D' + B C' + B C</td>
</tr>
</tbody>
</table>

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

| C0 = B C' D + C D + B' D' + B C D' + A |
| C1 = B' D' + C' D' + C D + B' D'   |
| C2 = B' D + B C' D + C D' + C D + B C D' |
| C3 = B' C' D + B' D' + B' D' + B C D' |
| C4 = B' D' + B C D'               |
| C5 = B C' D + C D' + A + B C D'  |
| C6 = B' C + B C' + B C D' + A   |
**PAL Implementation**

- Limit of 4 Product Terms per Output
  - Decomposition of functions with larger number of terms
  - Do not share terms in PAL anyway (although there are some with some shared terms)
    
    \[
    C_2 = B + C' + D
    \]
    
    \[
    C_2 = B' D + B C' D + C' D' + C D + B C D'
    \]
    
    \[
    C_2 = B' D + B C' D + C' D' + W
    \]
    
    \[
    W = C D + B C D'
    \]
    
  - Decompose into multi-level logic (hopefully with CAD support)
    - Find common sub-expressions among functions
      - \( C_0 = C_3 + A' B' X' + A D Y \)
      - \( C_1 = Y + A' C_5' + C' D' C_6 \)
      - \( C_2 = C_5 + A' B' D + A' C D \)
      - \( C_3 = C_4 + B D C_5 + A' B' X' \)
      - \( C_4 = D' Y + A' C D' \)
      - \( C_5 = C' C_4 + A Y + A' B X \)
      - \( C_6 = A C_4 + C C_5 + C_4' C_5 + A' B' C \)

**Production Line Control**

- Rods of varying length (+/-10%) travel on conveyor belt
  - Mechanical arm pushes rods within spec (+/-5%) to one side
  - Second arm pushes rods too long to other side
  - Rods that are too short stay on belt
  - 3 light barriers (light source + photocell) as sensors
  - Design combinational logic to activate the arms

- Understanding the problem
  - Inputs are three sensors
  - Outputs are two arm control signals
  - Assume sensor reads "1" when tripped, "0" otherwise
  - Call sensors A, B, C
Sketch of Problem

- Position of Sensors
  - A to B distance = specification − 5%
  - A to C distance = specification + 5%

Formalize the problem

- Truth Table
  - Show don't cares

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>do nothing</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>too short</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>don't care</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>in spec</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>too long</td>
</tr>
</tbody>
</table>

logic implementation now straightforward
just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

"too long" = A B C
(all three sensors tripped)
Logical Function Unit

- Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>always 1</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>A + B</td>
<td>logical OR</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>0</td>
<td>(A • B)'</td>
<td>logical NAND</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>A xor B</td>
<td>logical xor</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>0</td>
<td>A xnor B</td>
<td>logical xnor</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>A • B</td>
<td>logical AND</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>0</td>
<td>(A + B)'</td>
<td>logical NOR</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>always 0</td>
</tr>
</tbody>
</table>

Formalize the Problem

- choose implementation technology
- 5-variable K-map to discrete gates
- multiplexer implementation