Combinational Logic Technologies

- Standard gates
  - gate packages
  - cell libraries
- Regular logic
  - multiplexers
  - decoders
- Two-level programmable logic
  - PALs
  - PLAs
  - ROMs

Random logic

- Transistors quickly integrated into logic gates (1960s)
- Catalog of common gates (1970s)
  - Texas Instruments Logic Data Book – the yellow bible
  - all common packages listed and characterized (delays, power)
  - typical packages:
    - in 14-pin IC: 6-inverters, 4 NAND gates, 4 XOR gates
- Today, very few parts are still in use
- However, parts libraries exist for chip design
  - designers reuse already characterized logic gates on chips
  - same reasons as before
  - difference is that the parts don’t exist in physical inventory – created as needed
Random logic

- Too hard to figure out exactly what gates to use
  - map from logic to NAND/NOR networks
  - determine minimum number of packages
    - slight changes to logic function could decrease cost
- Changes to difficult to realize
  - need to rewire parts
  - may need new parts
  - design with spares (few extra inverters and gates on every board)

Regular logic

- Need to make design faster
- Need to make engineering changes easier to make
- Simpler for designers to understand and map to functionality
  - harder to think in terms of specific gates
  - better to think in terms of a large multi-purpose block
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

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
Mux and demux (cont'd)

- Uses of multiplexers/demultiplexers in multi-point connections

![Diagram of MUX and DEMUX with input and output connections]

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>
<th>$I_0$</th>
<th>$I_1$</th>
<th>$A$</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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<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:** \( Z = A'I_0 + A'I_1 \)
- **4:1 mux:** \( Z = A'B'I_0 + A'B'I_1 + AB'I_2 + ABI_3 \)
- **8:1 mux:** \( Z = A'B'C'I_0 + A'B'C'I_1 + A'BC'I_2 + A'BC'I_3 + AB'C'I_4 + AB'C'I_5 + ABC'I_6 + ABC'I_7 \)

- 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 of muxes

- **2:1 mux**

- **4:1 mux**
Cascading multiplexers

- Large multiplexers can be made by cascading smaller ones

Multiplexers as general-purpose logic

- A $2^n$:1 multiplexer can implement any function of $n$ variables
  - with the variables used as control inputs and
  - the data inputs tied to 0 or 1
  - in essence, a lookup table

Example:

- $F(A,B,C) = m_0 + m_2 + m_6 + m_7$
  - $= A'B'C' + A'BC' + ABC' + ABC$
  - $= A'B'C'(1) + A'B'C(0)$
  - $+ A'BC'(1) + A'BC(0)$
  - $+ AB'C'(0) + AB'C(0)$
  - $+ ABC'(1) + ABC(1)$

$$Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + A'B'C'I_4 + AB'C'I_5 + ABC'I_6 + ABCI_7$$
Multiplexers as general-purpose logic (cont’d)

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

- Generalization
  - $n$-1 mux control variables
  - single mux data variable
  - four possible configurations of truth table rows can be expressed as a function of $I_n$
- Example:
  - $G(A,B,C,D)$ can be realized by an 8:1 MUX
  - choose $A,B,C$ as control variables
Activity

- Realize \( F = B'CD' + ABC' \) with a 4:1 multiplexer and a minimum of other gates:

<table>
<thead>
<tr>
<th>ABCD</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>1:2 Decoder:</th>
<th>3:8 Decoder:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 = G * S'</td>
<td>00 = G * S2' * S1' * S0'</td>
</tr>
<tr>
<td>01 = G * S</td>
<td>01 = G * S2' * S1' * S0</td>
</tr>
<tr>
<td></td>
<td>02 = G * S2' * S1 * S0'</td>
</tr>
<tr>
<td>2:4 Decoder:</td>
<td></td>
</tr>
<tr>
<td>00 = G * S1' * S0'</td>
<td>00 = G * S2 * S1' * S0'</td>
</tr>
<tr>
<td>01 = G * S1' * S0</td>
<td>01 = G * S2 * S1 * S0</td>
</tr>
<tr>
<td>02 = G * S1 * S0'</td>
<td>02 = G * S2 * S1 * S0'</td>
</tr>
<tr>
<td>03 = G * S1 * S0</td>
<td>03 = G * S2 * S1 * S0</td>
</tr>
<tr>
<td></td>
<td>04 = G * S2 * S1 * S0</td>
</tr>
<tr>
<td></td>
<td>05 = G * S2 * S1 * S0</td>
</tr>
<tr>
<td></td>
<td>06 = G * S2 * S1 * S0</td>
</tr>
<tr>
<td></td>
<td>07 = G * S2 * S1 * S0</td>
</tr>
</tbody>
</table>
Gate level implementation of demultiplexers

- **1:2 decoders**
  - active-high enable
  - active-low enable

- **2:4 decoders**
  - active-high enable
  - active-low enable

Demultiplexers as general-purpose logic

- A n:2ⁿ decoder can implement any function of n variables
  - with the variables used as control inputs
  - the enable inputs tied to 1 and
  - the appropriate minterms summed to form the function

demuxer generates appropriate minterm based on control signals (it "decodes" control signals)
Demultiplexers as general-purpose logic (cont’d)

- \( F_1 = A'B'C'D + A'B'CD + ABCD \)
- \( F_2 = ABC'D' + ABC \)
- \( F_3 = (A' + B' + C' + D') \)

Cascading decoders

- 5:32 decoder
  - 1x2:4 decoder
  - 4x3:8 decoders
Programmable logic arrays

- Pre-fabricated building block of many AND/OR gates
  - actually NOR or NAND
  - "personalized" by making/breaking connections among the gates
  - programmable array block diagram for sum of products form

![Programmable logic array diagram]

Enabling concept

- Shared product terms among outputs

example:

\[
\begin{align*}
F_0 &= A + B' C' \\
F_1 &= A C' + A B \\
F_2 &= B' C' + A B \\
F_3 &= B' C + A
\end{align*}
\]

personality matrix

<table>
<thead>
<tr>
<th>product term</th>
<th>inputs</th>
<th>outputs</th>
<th>F₀</th>
<th>F₁</th>
<th>F₂</th>
<th>F₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>A'B'C'</td>
<td>1 0 0</td>
<td>1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td>1 0 -</td>
<td>1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A'B'C</td>
<td>- 1 0</td>
<td>- 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>- 1 0</td>
<td>- 1 0 0</td>
<td></td>
<td></td>
<td></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

reuse of terms
Before programming

- All possible connections are 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 for high fan-in structures

- Short-hand notation so we don't have to draw all the wires
  - $\times$ signifies a connection is present and perpendicular signal is an input to gate

Programmable logic array example

- Multiple functions of A, B, C
  - $F_1 = A \cdot B \cdot C$
  - $F_2 = A + B + C$
  - $F_3 = A' \cdot B' \cdot C'$
  - $F_4 = A' + B' + C'$
  - $F_5 = A \oplus B \oplus C$
  - $F_6 = A \oplus B \oplus C$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</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>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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>1</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>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>1</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>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Full decoder as for memory address

<table>
<thead>
<tr>
<th>Full decoder as for memory address</th>
<th>bits stored in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A'B'C'$</td>
<td>$A'B'$</td>
</tr>
<tr>
<td>$A'B'C$</td>
<td>$A'BC'$</td>
</tr>
<tr>
<td>$A'BC$</td>
<td>$A'BC$</td>
</tr>
<tr>
<td>$AB'C$</td>
<td>$AB'C$</td>
</tr>
<tr>
<td>$ABC'$</td>
<td>$ABC'$</td>
</tr>
<tr>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
</tbody>
</table>

F6
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
  - innovation by Monolithic Memories
  - faster and smaller OR plane

A given column of the OR array has access to only a subset of the possible product terms.

PALs and PLAs: design example

- BCD to Gray code converter

<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>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<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>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<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>0</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Minimized functions:

- \( W = A + BD + BC \)
- \( X = B'C \)
- \( Y = B + C \)
- \( Z = A'B'CD + BCD + AD' + B'CD' \)
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

minimized functions:
\[ W = A + BD + BC \]
\[ X = B'C' \]
\[ Y = B + C \]
\[ Z = A'B'C'D + BCD + AD' + B'CD' \]
PALs and PLAs: design example (cont’d)

- Code converter: NAND gate implementation
  - loss or regularity, harder to understand
  - harder to make changes

PALs and PLAs: another design example

- Magnitude comparator

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>EQ</th>
<th>NE</th>
<th>LT</th>
<th>GT</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>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</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>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<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>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>1</td>
<td>0</td>
<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>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>1</td>
<td>0</td>
<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>0</td>
<td>0</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>0</td>
<td>0</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>0</td>
<td>0</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>0</td>
<td>0</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>0</td>
<td>0</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

minimized functions:
EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD'  NE = AC' + A'C + B'D + BD'  LT = A'C + A'B'D + B'CD  GT = AC' + ABC + BCD'
Activity

Map the following functions to the PLA below:

- \( W = AB + A'C' + BC' \)
- \( X = ABC + AB' + A'B \)
- \( Y = ABC' + BC + B'C' \)

Activity (cont’d)

9 terms won’t fit in a 7 term PLA

- can apply consensus theorem to \( W \) to simplify to:
  \( W = AB + A'C' \)

8 terms wont’ fit in a 7 term PLA

- observe that \( AB = ABC + ABC' \)
- can rewrite \( W \) to reuse terms:
  \( W = ABC + ABC' + A'C' \)

Now it fits

- \( W = ABC + ABC' + A'C' \)
- \( X = ABC + AB' + A'B \)
- \( Y = ABC' + BC + B'C' \)

This is called technology mapping

- manipulating logic functions so that they can use available resources
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](image)

Bit lines (normally pulled to 1 through resistor – selectively connected to 0 by word line controlled switches)

Block diagram

Autumn 2003  CSE370 - IV - Combinational Logic Technologies

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>$F_0$</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

F0 = $A' \cdot B \cdot C$ + $A \cdot B' \cdot C'$ + $A \cdot B \cdot C$
F1 = $A' \cdot B' \cdot C$ + $A' \cdot B \cdot C'$ + $A \cdot B \cdot C$
F2 = $A' \cdot B' \cdot C'$ + $A' \cdot B \cdot C$ + $A \cdot B' \cdot C$
F3 = $A' \cdot B \cdot C$ + $A \cdot B' \cdot C$ + $A \cdot B \cdot C$

8 words x 4 bits/word

Autumn 2003  CSE370 - IV - Combinational Logic Technologies
ROM structure

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

```
  n address lines
    • •   •
  inputs

  decoder

  2^n word lines

  memory array
  (2^n words by m bits)

  • •   •
  outputs

  m data lines
```

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
## Regular logic structures for two-level logic

- **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)

## Regular logic 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
  - in 467 you'll learn about field programmable gate arrays (FPGAs) that 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 logic technology summary

- Random logic
  - Single gates or in groups
  - conversion to NAND-NAND and NOR-NOR networks
  - transition from simple gates to more complex gate building blocks
  - reduced gate count, fan-ins, potentially faster
  - more levels, harder to design

- Time response in combinational networks
  - gate delays and timing waveforms
  - hazards/glitches (what they are and why they happen)

- Regular logic
  - multiplexers/decoders
  - ROMs
  - PLAs/PALs
  - advantages/disadvantages of each