CSE140: Components and Design Techniques for Digital Systems

Tajana Simunic Rosing
Review: Mux/Demux Example

- $F(A, B, C) = \Pi M(0, 2, 4)$
Hazards

- **Glitch** – unwanted pulse on the output
- Circuit with a potential for a glitch has a **hazard**
- Three types:
  - Static-0: output should be 0 but has a 1 glitch
  - Static-1: output should be 1 but has a 0 glitch
  - Dynamic: transition 0->1 or 1->0 with a glitch
Eliminating Hazards Example

- \( F(A, B, C, D) = \sum m(1, 3, 5, 7, 8, 9, 12, 13) \)
- Test two single bit input transitions:
  - 1100 -> 1101
  - 1100 -> 0101
CSE140: Components and Design Techniques for Digital Systems

Two and Multilevel logic implementation

Tajana Simunic Rosing
Conversion to NAND/NOR gates

- Use De Morgan’s:
  - $A’ + B’ = (A \cdot B)’$
  - $A’ \cdot B’ = (A+B)’$

- Two-level NAND-NAND example:
Conversion between forms

• Example: map AND/OR network to NOR/NOR network
• Use De Morgan’s:
  – \( A' + B' = (A \cdot B)' \)
  – \( A' \cdot B' = (A+B)' \)
Multiple-Output Circuits

- Many circuits have more than one output
- Can give each a separate circuit, or can share gates
- Ex: $F = ab + c'$, $G = ab + bc$

Option 1: Separate circuits
Option 2: Shared gates
Multi-level logic

- \( x = A \overline{D} F + A \overline{E} F + B \overline{D} F + B \overline{E} F + C \overline{D} F + C \overline{E} F + G \)
  - reduced sum-of-products form – already simplified
  - 6 x 3-input AND gates + 1 x 7-input OR gate (that may not even 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)
Multiple-Output Example: BCD to 7-Segment Converter

\[ a = w'x'y'z' + w'x'yz + w'x'yz + w'xy'z + w'xyz' + w'xyz + wx'y'z' + wx'y'z \]

\[ b = w'x'y'z' + w'x'y'z + w'x'yz' + w'x'yz + w'xy'z' + w'xyz + wx'y'z' + wx'y'z \]
CSE140: Components and Design Techniques for Digital Systems

Regular logic implementation

Tajana Simunic Rosing
AND-OR-invert gates

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

```
2x2 AOI gate symbol

3x2 AOI gate symbol

Sources: TSR, Katz, Boriello & Vahid
```
Conversion to AOI forms

- General procedure to place in AOI form
  - compute the complement of the function in sum-of-products form
  - by grouping the 0s in the Karnaugh map
- Example: XOR implementation
  - \( A \text{ xor } B = A' B + A B' \)
Programmable logic arrays

• Pre-fabricated building block of many AND/OR gates
  – "personalized" by making/breaking connections among the gates
  – programmable array block diagram for sum of products form

![Diagram of programmable logic arrays]

AND array

OR array

inputs

outputs

product terms
Before & after programming

- Two different technologies:
  - fuse (normally connected, break unwanted ones)
  - anti-fuse (normally disconnected, make wanted connections)
Programmable logic array example

- Multiple functions of A, B, C
  - $F_1 = A \ b \ c$
  - $F_2 = A + B + C$
  - $F_3 = \overline{A} \ \overline{B} \ \overline{C}$
  - $F_4 = \overline{A} + \overline{B} + \overline{C}$
  - $F_5 = A \ xor \ B \ xor \ C$
  - $F_6 = A \ xnor \ B \ xnor \ C$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
<th>$F_5$</th>
<th>$F_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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>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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>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>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
CSE140: Components and Design Techniques for Digital Systems

Tajana Simunic Rosing
PALs and PLAs

- Programmable logic array (PLA)
  - unconstrained fully-general AND and OR arrays
- Programmable array logic (PAL)
  - constrained topology of the OR array
  - faster and smaller OR plane

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

Map the following functions to the PLA below:

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

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

word lines

word[i] = 0011
word[j] = 1010

bit lines (normally 1; set to 0 by a switch)

2^n - 1

decoder

internal organization

Sources: TSR, Katz, Boriello & Vahid
ROM structure

• Similar to a PLA structure but with a fully decoded AND array
  – completely flexible OR array (unlike PAL)
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)
CSE140: Components and Design Techniques for Digital Systems

Adders

Tajana Simunic Rosing
Ripple-carry adder critical delay path

late arriving signal
two gate delays to compute Cout

Sources: TSR, Katz, Boriello & Vahid
Carry-lookahead adders

- Adder with propagate (P) and generate (G) outputs:
- Evaluate Sum and Ci+1
  - Sum = $A_i \oplus B_i \oplus C_i$
  - $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$
    \[= A_i B_i + C_i (A_i \oplus B_i)\]
    \[= G_i + C_i P_i\]

Increasingly complex logic for carries
Carry-lookahead implementation (cont’d)

- Carry-lookahead logic generates individual carries
  - sums computed much more quickly in parallel
  - however, cost of carry logic increases with more stages

\[ 0 \]

\[ (A_0, B_0) \rightarrow (S_0, C_1) \]

\[ (A_1, B_1) \rightarrow (S_1, C_2) \]

\[ (A_2, B_2) \rightarrow (S_2, C_3) \]

\[ (A_3, B_3) \rightarrow (S_3, C_{out}) \]
Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits

\[
G = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0
\]

\[
P = P_3 P_2 P_1 P_0
\]

\[
C_1 = G_0 + P_0 C_0
\]

\[
C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0
\]
Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed
Adders - review

- Carry-ripple
- Carry lookahead
- Carry select

(a) 4-bit Adder

\[ \begin{align*}
&\text{Pi} @ 1 \text{ gate delay} \\
&\text{Si} @ 2 \text{ gate delays} \\
&\text{Gi} @ 1 \text{ gate delay} \\
\end{align*} \]

\[ \begin{align*}
&\text{Pi} @ 1 \text{ gate delay} \\
&\text{Si} @ 2 \text{ gate delays} \\
&\text{Gi} @ 1 \text{ gate delay} \\
\end{align*} \]
Adder/subtractor
Multiplier – Array Style

- Multiplier design – array of AND gates
Logical function unit

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

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Equality check</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A OR B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A−B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A+B</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid
CSE140: Components and Design Techniques for Digital Systems

Review

Tajana Simunic Rosing
NAND gates implementation

\[ A = (X + ((\overline{Y})(Z)))(Y + \overline{Z}) \]
Kmap, PIs, EPIs, min SOP/POS, Hazards

\[ F(A, B, C, D) = \sum m(2, 4, 7, 9, 10, 11, 14) + \sum d(3, 6, 12) \]
Mux & Demux

\[ F(A, B, C, D) = \sum m(2, 4, 7, 9, 10, 11, 14) + \sum d(3, 6, 12) \]
PLA
Design problem – distance between numbers

- Design a circuit that gives the absolute distance between the two numbers (e.g. x=3 y=1 d=2)

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_0$</th>
<th>$y_1$</th>
<th>$y_0$</th>
<th>$d_1$</th>
<th>$d_0$</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</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>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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary

• What we’ve covered so far (Chap 1-5, App. A& B)
  – Number representations
  – Switches and CMOS transistor gates
  – Boolean algebra
  – SOP and POS
  – Logic minimization using K-maps
  – Two and multi-level implementation
  – Hazards
  – AOI, PAL, PLA, ROM implementation
  – Mux and Demux
  – Adders, Multipliers and ALUs

• What comes next:
  – Sequential circuits