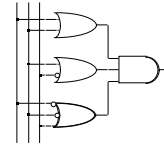
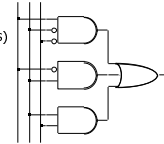


Combinational logic implementation

- Two-level logic
 - implementations of two-level logic
 - NAND/NOR
- Multi-level logic
 - factored forms
 - and-or-invert gates
- Time behavior
 - gate delays
 - hazards
- Regular logic
 - multiplexors
 - decoders
 - PAL/PLAs
 - ROMs

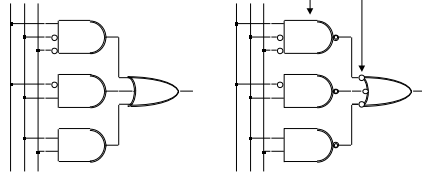
Implementations of two-level logic

- 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



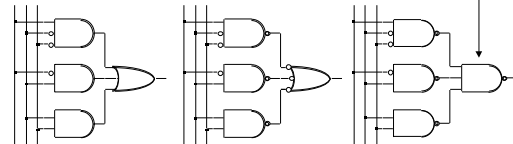
Two-level logic using NAND gates

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



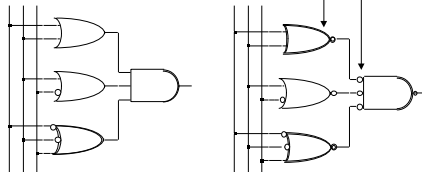
Two-level logic using NAND gates (cont'd)

- 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



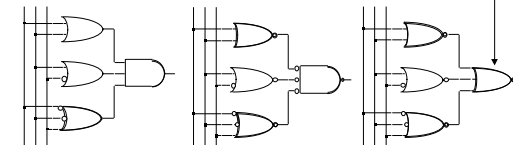
Two-level logic using NOR gates

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



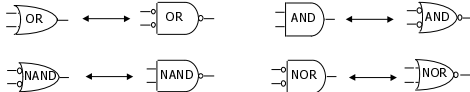
Two-level logic using NOR gates (cont'd)

- 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 and NOR gates

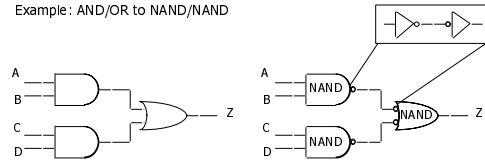
- NAND-NAND and NOR-NOR networks
 - de Morgan's law: $(A + B)' = A' \cdot B'$ $(A \cdot B)' = A' + B'$
 - written differently: $A + B = (A' \cdot B)'$ $(A \cdot B) = (A' + B)'$
- In other words --
 - OR is the same as NAND with complemented inputs
 - AND is the same as NOR with complemented inputs
 -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



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 7

Conversion between forms

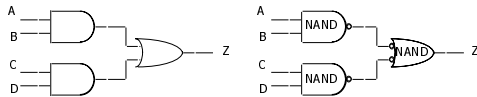
- Convert from networks of ANDs and ORs to networks of NANDs and NORs
 - introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be matched by a corresponding "bubble"
 - conservation of inversions
 - do not alter logic function
- Example: AND/OR to NAND/NAND



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 8

Conversion between forms (cont'd)

- Example: verify equivalence of two forms

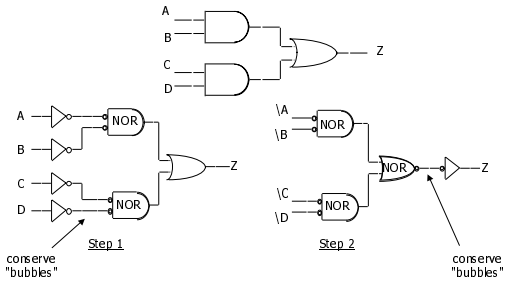


$$\begin{aligned}
 Z &= [(A \cdot B)' \cdot (C \cdot D)'] \\
 &= [(A' + B') \cdot (C' + D)'] \\
 &= [(A' + B)'] + [(C' + D)'] \\
 &= (A \cdot B) + (C \cdot D) \quad \Rightarrow
 \end{aligned}$$

1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 9

Conversion between forms (cont'd)

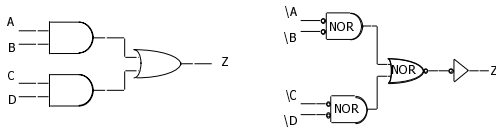
- Example: map AND/OR network to NOR/NOR network



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 10

Conversion between forms (cont'd)

- Example: verify equivalence of two forms

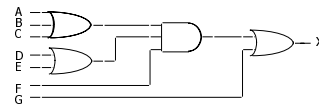


$$\begin{aligned}
 Z &= \{ [(A' + B') + (C' + D)'] \}' \\
 &= \{ (A' + B) + (C' + D) \}' \\
 &= (A \cdot B) + (C \cdot D) \quad \Rightarrow
 \end{aligned}$$

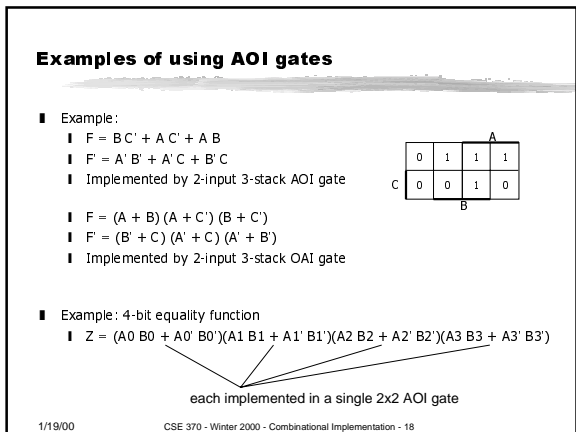
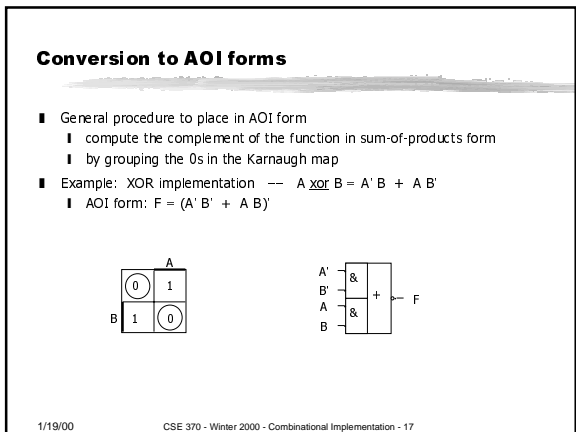
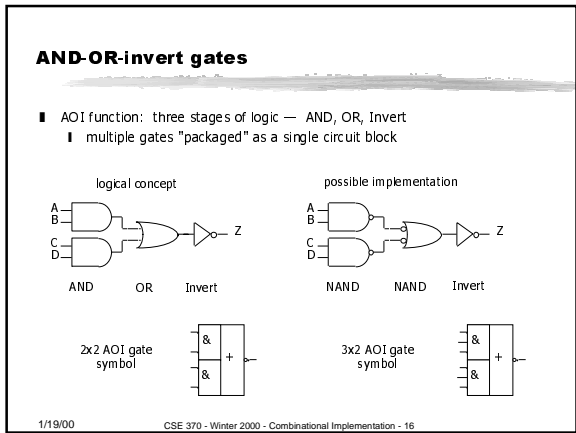
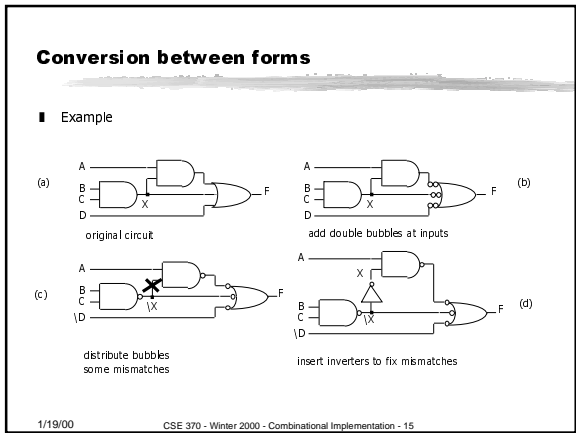
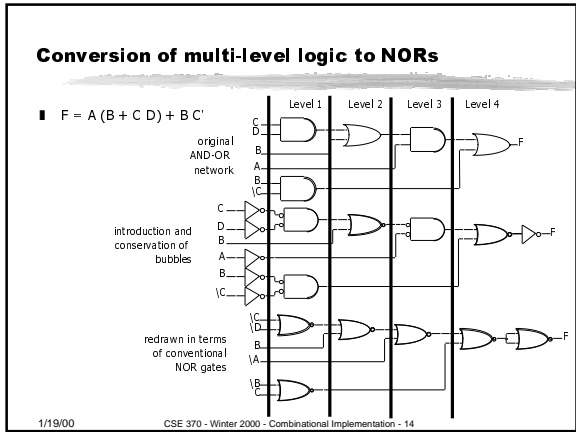
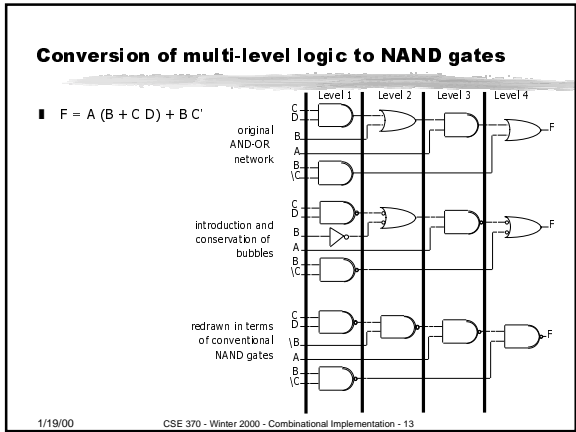
1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 11

Multi-level logic

- $x = ADF + AEF + BDF + BEF + CDF + CEF + G$
 - 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)

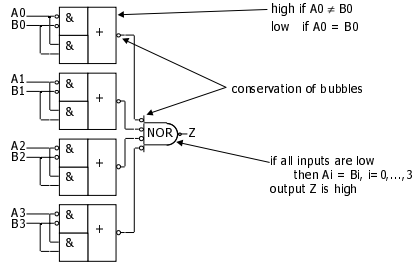


1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 12



Examples of using AOI gates (cont'd)

- Example: AOI implementation of 4-bit equality function



Summary for multi-level logic

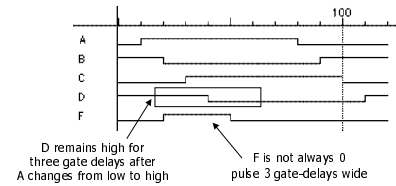
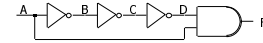
- Advantages
 - circuits may be smaller
 - gates have smaller fan-in
 - circuits may be faster
- Disadvantages
 - more difficult to design
 - tools for optimization are not as good as for two-level
 - analysis is more complex

Time behavior of combinational networks

- 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 that an output stays high or stays low between changes

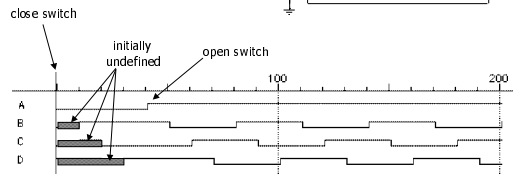
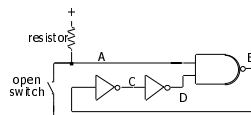
Momentary changes in outputs

- Can be useful — pulse shaping circuits
- Can be a problem — incorrect circuit operation (glitches/hazards)
 - Example: pulse shaping circuit
 - $A' \cdot A = 0$
 - delays matter in function



Oscillatory behavior

- Another pulse shaping circuit
- NOT combinational logic!

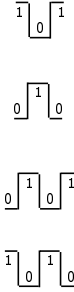


Hazards/glitches

- 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
 - wait until signals are stable (by using a clock)
 - preferable (easiest to design when there is a clock — synchronous design)
 - design hazard-free circuits
 - sometimes necessary (clock not used — asynchronous design)

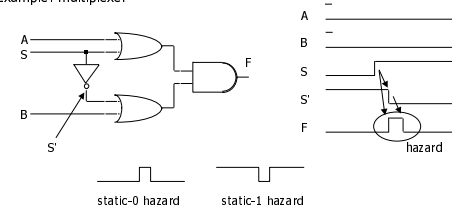
Types of hazards

- 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



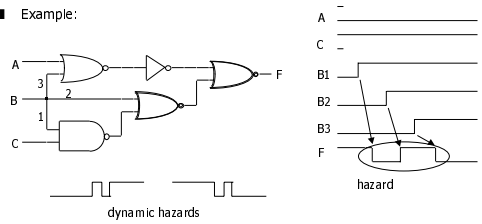
Static hazards

- Due to a literal and its complement momentarily taking on the same value
 - ┆ through 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: multiplexer



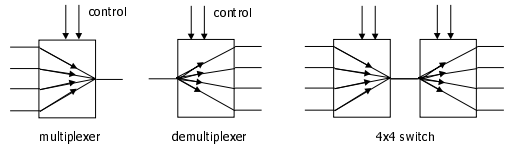
Dynamic hazards

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



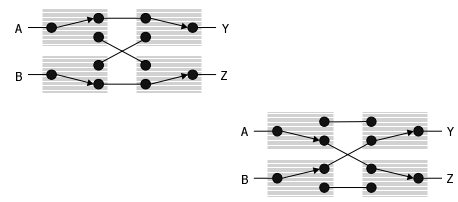
Making connections

- 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



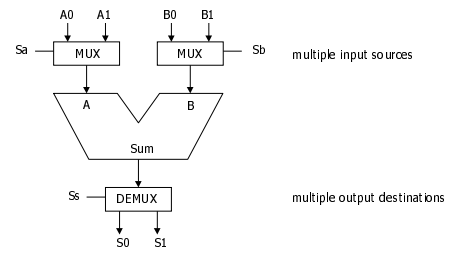
Mux and demux

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



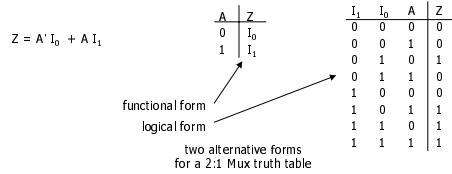
Mux and demux (cont'd)

- Uses of multiplexers/demultiplexers in multi-point connections



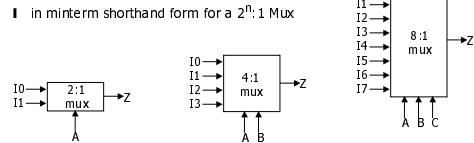
Multiplexers/selectors

- 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



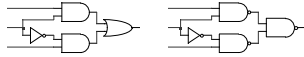
Multiplexers/selectors (cont'd)

- 2:1 mux: $Z = A' I_0 + A I_1$
- 4:1 mux: $Z = A' B' I_0 + A' B I_1 + A B' I_2 + A B I_3$
- 8:1 mux: $Z = A' B' C' I_0 + A' B' C I_1 + A' B C' I_2 + A' B C I_3 + A B' C' I_4 + A B' C I_5 + A B C' I_6 + A B C I_7$
- In general, $Z = \sum_{k=0}^{2^n-1} (m_k I_k)$

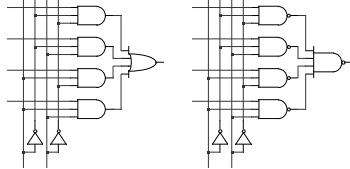


Gate level implementation of muxes

- 2:1 mux

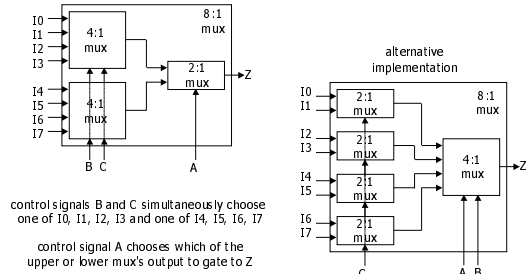


- 4:1 mux



Cascading multiplexers

- Large multiplexers can be implemented by cascading smaller ones

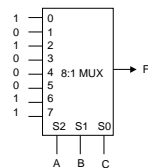


Multiplexers as general-purpose logic

- 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:

$$\begin{aligned}
 F(A,B,C) &= m_0 + m_2 + m_6 + m_7 \\
 &= A'B'C' + A'BC' + ABC' + ABC \\
 &= A'B'(C') + A'B(C') + AB'(0) + AB(1)
 \end{aligned}$$

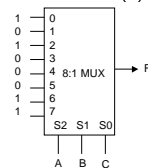


Multiplexers as general-purpose logic (cont'd)

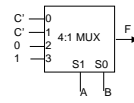
- A $2^{n-1}:1$ multiplexer can implement any function of n variables
 - with $n-1$ variables used as control inputs and
 - the data inputs tied to the last variable or its complement

- Example:

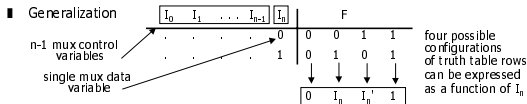
$$\begin{aligned}
 F(A,B,C) &= m_0 + m_2 + m_6 + m_7 \\
 &= A'B'C' + A'BC' + ABC' + ABC \\
 &= A'B'(C') + A'B(C') + AB'(0) + AB(1)
 \end{aligned}$$



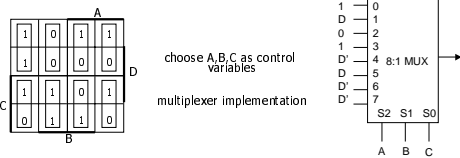
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Multiplexers as general-purpose logic (cont'd)



Example: $F(A,B,C,D)$ can be implemented by an 8:1 MUX



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:

$$\begin{aligned} O0 &= G \cdot S' \\ O1 &= G \cdot S \end{aligned}$$

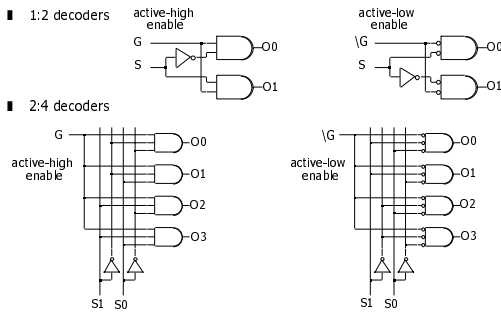
3:8 Decoder:

$$\begin{aligned} O0 &= G \cdot S2' \cdot S1' \cdot S0' \\ O1 &= G \cdot S2' \cdot S1' \cdot S0 \\ O2 &= G \cdot S2' \cdot S1 \cdot S0' \\ O3 &= G \cdot S2' \cdot S1 \cdot S0 \\ O4 &= G \cdot S2 \cdot S1' \cdot S0' \\ O5 &= G \cdot S2 \cdot S1' \cdot S0 \\ O6 &= G \cdot S2 \cdot S1 \cdot S0' \\ O7 &= G \cdot S2 \cdot S1 \cdot S0 \end{aligned}$$

2:4 Decoder:

$$\begin{aligned} O0 &= G \cdot S1' \cdot S0' \\ O1 &= G \cdot S1' \cdot S0 \\ O2 &= G \cdot S1 \cdot S0' \\ O3 &= G \cdot S1 \cdot S0 \end{aligned}$$

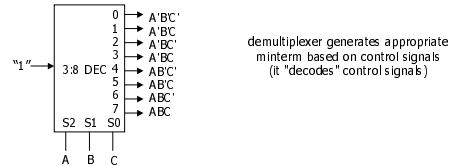
Gate level implementation of demultiplexers



Demultiplexers as general-purpose logic

A $n:2^n$ 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

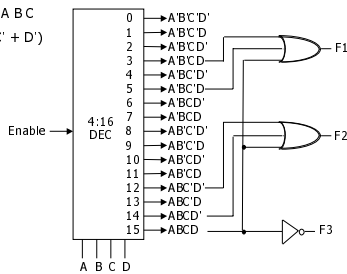


Demultiplexers as general-purpose logic (cont'd)

$F1 = A'BC'D + A'B'CD + ABCD$

$F2 = ABC'D' + ABCD$

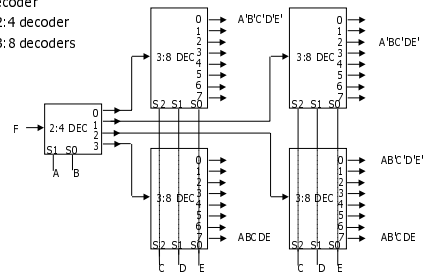
$F3 = (A' + B' + C' + D')$



Cascading decoders

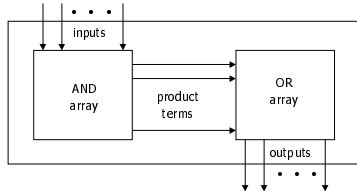
5:32 decoder

- 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 or breaking connections among the gates
 - programmable array block diagram for sum of products form



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 43

Enabling concept

- Shared product terms among outputs

example:

$$\begin{aligned} F_0 &= A + B'C' \\ F_1 &= A'C' + AB \\ F_2 &= B'C' + AB \\ F_3 &= B'C' + A \end{aligned}$$

personality matrix

product term	inputs			outputs			
	A	B	C	F ₀	F ₁	F ₂	F ₃
AB	1	1	-	0	1	1	0
B'C	-	0	1	0	0	0	1
AC'	1	-	0	0	1	0	0
B'C'	-	0	0	1	0	1	0
A	1	-	-	1	0	0	1

input side:
 1 = uncomplemented in term
 0 = complemented in term
 - = does not participate

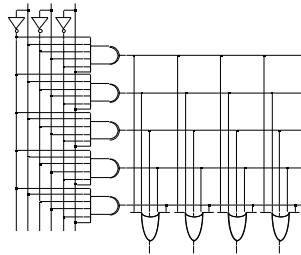
output side:
 1 = term connected to output
 0 = no connection to output

reuse of terms

1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 44

Before programming

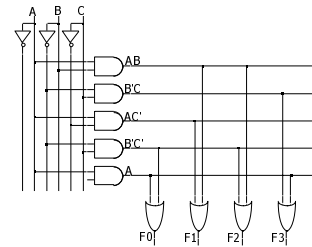
- All possible connections are available before "programming"
 - in reality, all AND and OR gates are NANDs



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 45

After programming

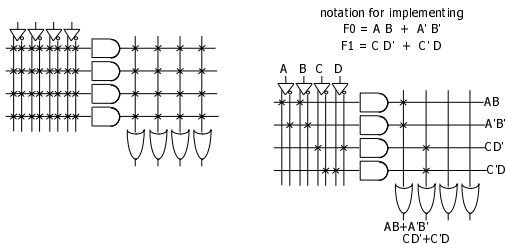
- Unwanted connections are "blown"
 - fuse (normally connected, break unwanted ones)
 - anti-fuse (normally disconnected, make wanted connections)



1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 46

Alternate representation for high fan-in structures

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



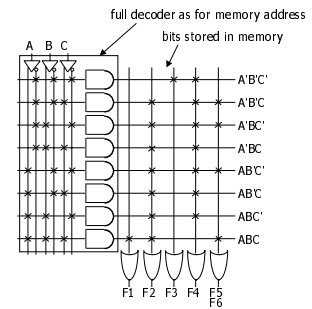
1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 47

Programmable logic array example

- Multiple functions of A, B, C

- F₁ = A B C
- F₂ = A + B + C
- F₃ = A' B' C'
- F₄ = A' + B' + C'
- F₅ = A xor B xor C
- F₆ = A xnor B xnor C

A	B	C	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆
0	0	0	0	1	1	0	0	0
0	0	1	0	1	0	1	1	1
0	1	0	1	0	1	1	1	1
0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1
1	0	1	0	1	0	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	1	0	0	1	1

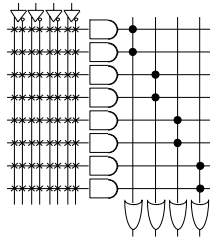


1/19/00 CSE 370 - Winter 2000 - Combinational Implementation - 48

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

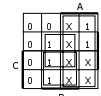
a given column of the OR array has access to only a subset of the possible product terms



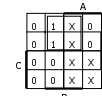
PALs and PLAs: design example

- BCD to Gray code converter

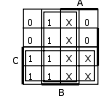
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	-	-	-	-	-
1	1	-	-	-	-	-	-



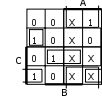
K-map for W



K-map for X



K-map for Y

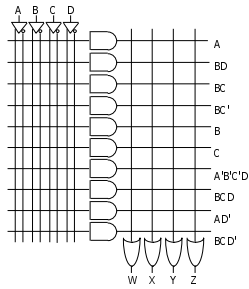


K-map for Z

minimized functions:
 $W = A + B D + B C$
 $X = B C'$
 $Y = B + C$
 $Z = A'B'C'D + B C D + A D' + B' C D'$

PALs and PLAs: design example (cont'd)

- Code converter: programmed PAL



minimized functions:
 $W = A + B D + B C$
 $X = B C'$
 $Y = B + C$
 $Z = A'B'C'D + B C D + A D' + B' C D'$

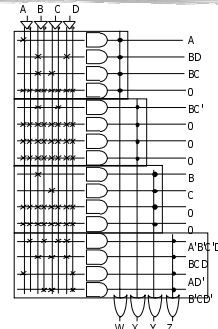
not a particularly good candidate for PAL/PLA implementation since no terms are shared among outputs

however, much more compact and regular implementation when compared with discrete AND and OR gates

PALs and PLAs: design example (cont'd)

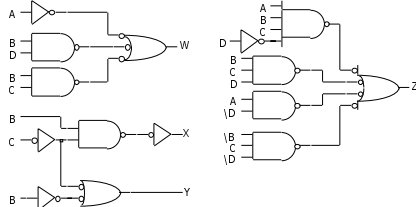
- Code converter: programmed PAL

4 product terms per each OR gate



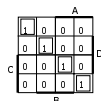
PALs and PLAs: design example (cont'd)

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

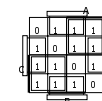


PALs and PLAs: another design example

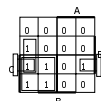
- Magnitude comparator



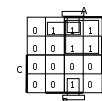
K-map for EQ



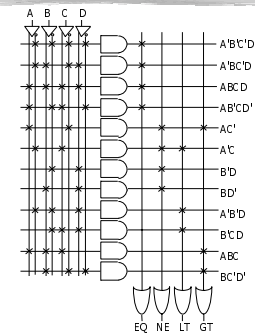
K-map for NE



K-map for LT

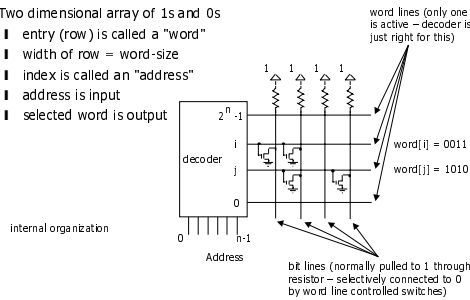


K-map for GT



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



1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 55

ROMs and combinational logic

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

$$F_0 = A'B'C + A'B'C' + A'BC$$

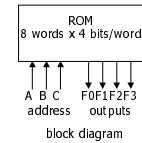
$$F_1 = A'B'C + A'BC' + ABC$$

$$F_2 = A'B'C' + A'BC' + AB'C'$$

$$F_3 = A'BC + AB'C' + ABC'$$

A	B	C	F0	F1	F2	F3
0	0	0	0	1	0	
0	0	1	1	1	0	
0	1	0	0	1	0	
0	1	1	0	0	0	1
1	0	0	1	0	1	1
1	0	1	0	0	0	0
1	1	0	0	0	0	1
1	1	1	0	1	0	0

truth table



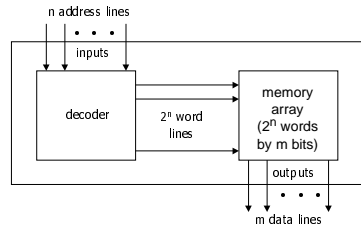
block diagram

1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 56

ROM structure

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



1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 57

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

1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 58

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)

1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 59

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

1/19/00

CSE 370 - Winter 2000 - Combinational Implementation - 60

Combinational logic implementation summary

- Multi-level logic
 - 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