EECS150

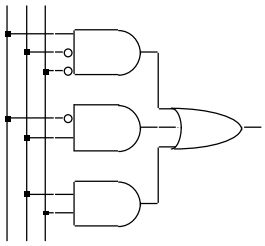
Section 6

Advanced Combinational Logic Issues

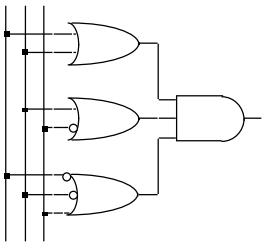
Fall 2001

Implementation: Two-level Logic

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

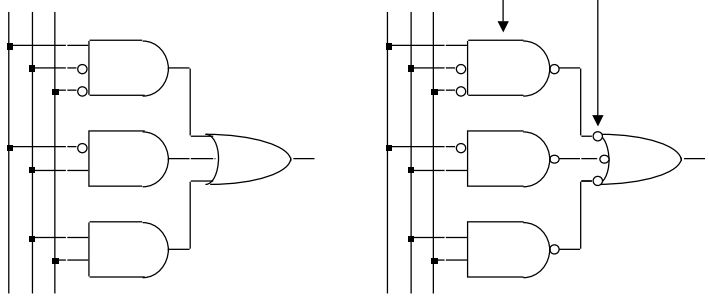


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



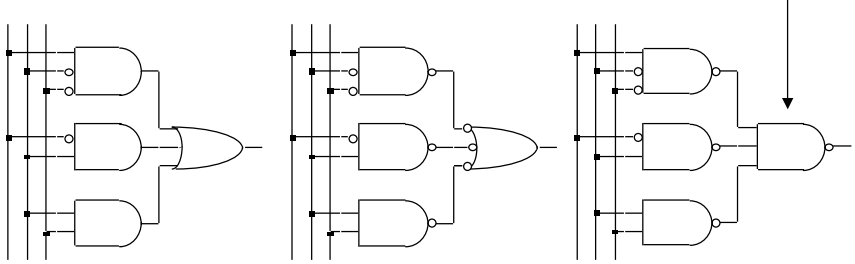
Two-level Logic using NAND Gates

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



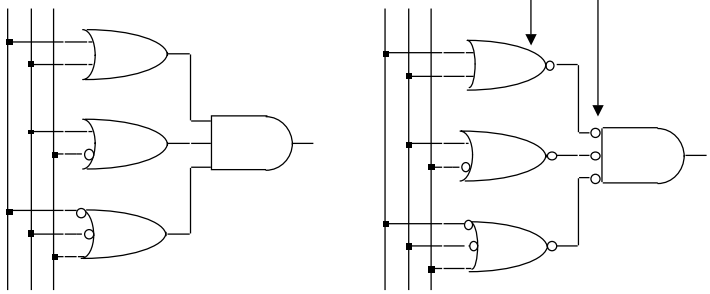
Two-level Logic using NAND Gates

- OR gate with inverted inputs is a NAND gate
  - de Morgan's: $A' + B' = (A \bullet B)'$
- Two-level NAND-NAND network
  - Inverted inputs are not counted
  - In a typical circuit, inversion is done once and signal distributed



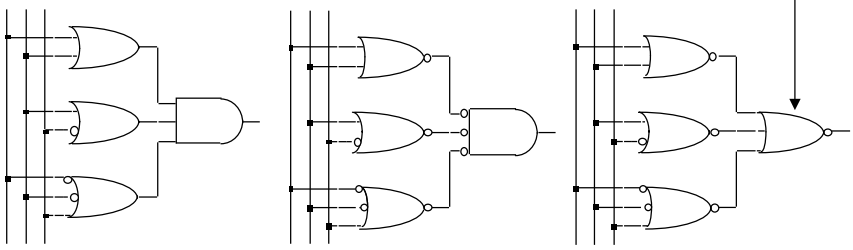
Two-level Logic using NOR Gates

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



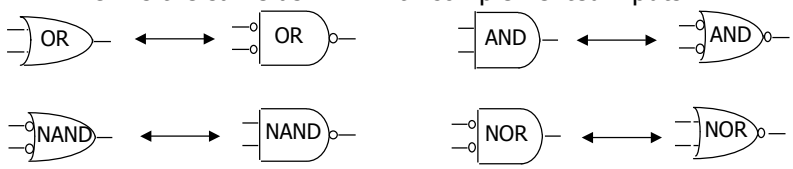
Two-level Logic using NOR Gates

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



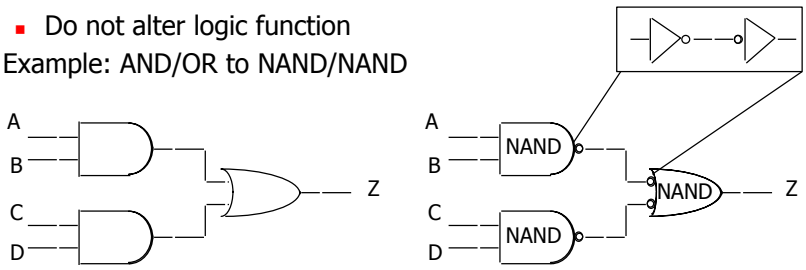
Two-level Logic using NAND / NOR

- NAND-NAND and NOR-NOR networks
  - de Morgan's law: $(A + B)' = A' \bullet B'$
    $(A \bullet B)' = A' + B'$
  - written differently: $A + B = (A' \bullet B')'$
    $(A \bullet B) = (A' + B')'$
- 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



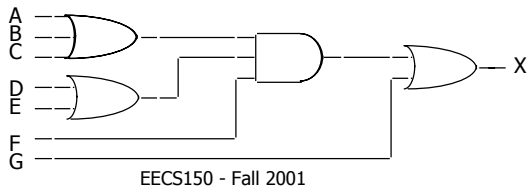
Conversion Between Forms

- 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



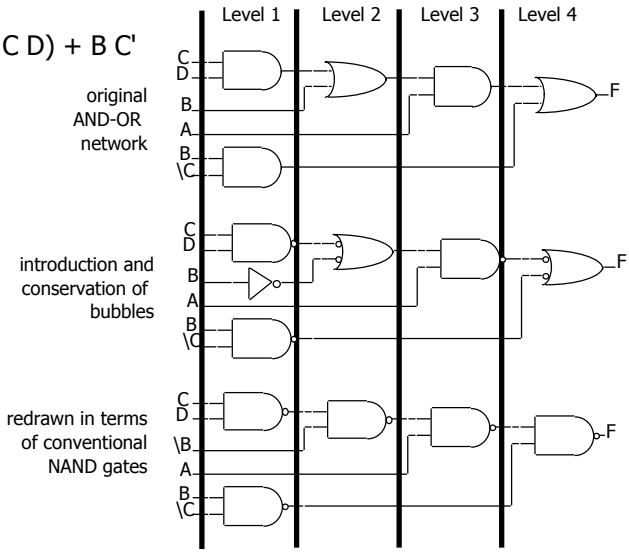
Multi-level Logic

- 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)



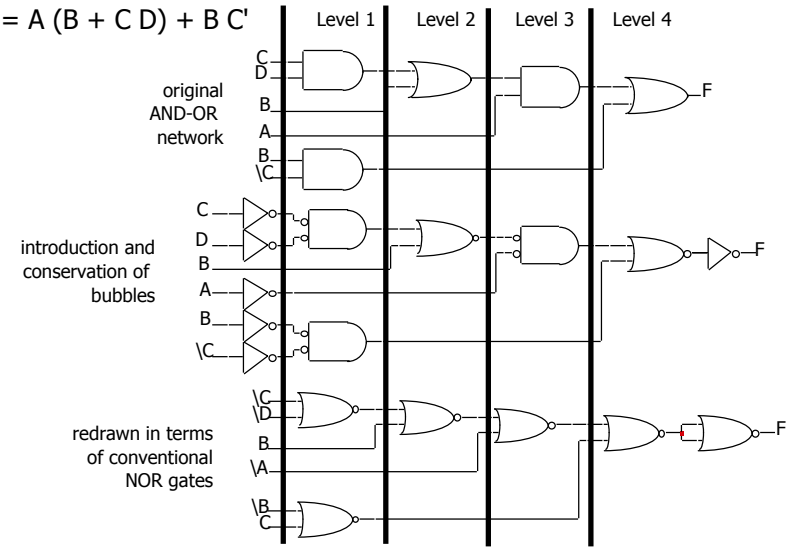
Conversion: Multi-level to NAND

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



Conversion: Multi-level to NORs

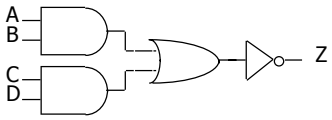
- F = A (B + C D) + B C'



AND-OR-Invert Gates

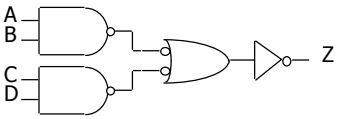
- AOI function: three stages of logic—AND, OR, Invert
  - Multiple gates "packaged" as a single circuit block

logical concept



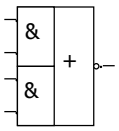
AND OR Invert

possible implementation

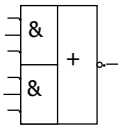


NAND NAND Invert

2x2 AOI gate symbol

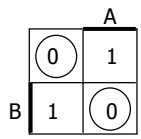


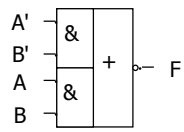
3x2 AOI gate symbol



Conversion to AOI Forms

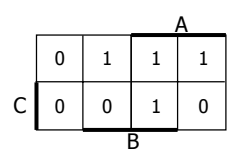
- General procedure to place in AOI form
  - Compute complement of the function in sum-of-products form
  - By grouping the 0s in the Karnaugh map
- Example: XOR implementation—A <u>xor</u> B = A' B + A B'
  - AOI form: F = (A' B' + A B)'



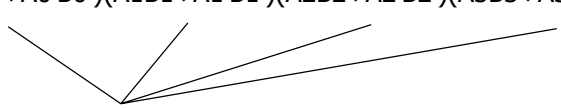


Examples of using AOI gates

- 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+A0'B0')(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

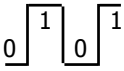
- Hazards/glitches: unwanted switching at the outputs
  - Occur when different paths through circuit have different propagation delays
    - As in pulse shaping circuits we just analyzed
  - Dangerous if logic causes an action while output is unstable
    - May need to guarantee absence of glitches
- 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)

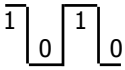
Types of Hazards

- 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

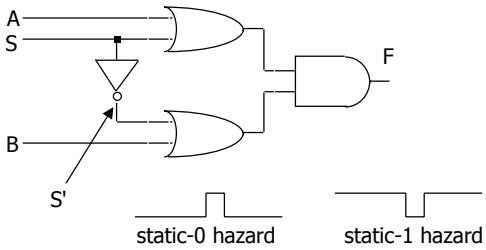
- Dynamic hazards
  - Input change causes a double change from 0 to 1 to 0 to 1 OR from 1 to 0 to 1 to 0

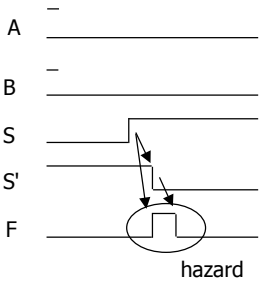




Static Hazards

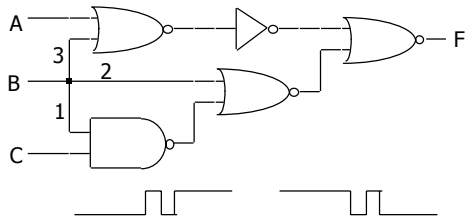
- Due to a literal and its complement momentarily taking on the same value
  - Thru different paths with different delays and reconverging
- May cause an output that should have stayed at the same value to momentarily take on the wrong value
- Example:



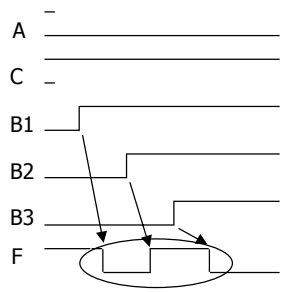


Dynamic Hazards

- 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:



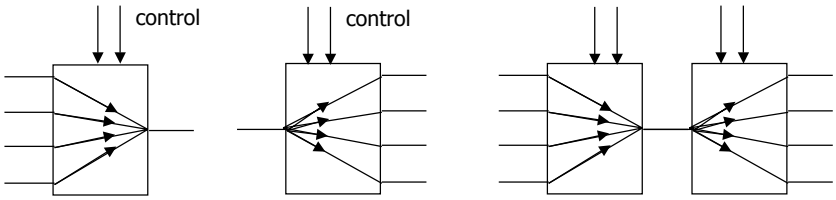
dynamic hazards



hazard

Making Connections

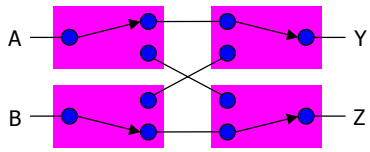
- Direct point-to-point connections between gates
  - Wires we've seen so far
- Route one of many inputs to a single output --- *multiplexer*
- Route a single input to one of many outputs --- *demultiplexer*

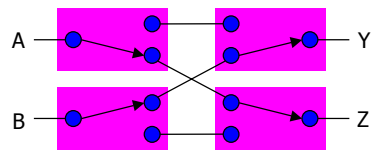


multiplexer demultiplexer 4x4 switch

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





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

$Z = A' I_0 + A I_1$

A	Z
0	I_0
1	I_1

functional form

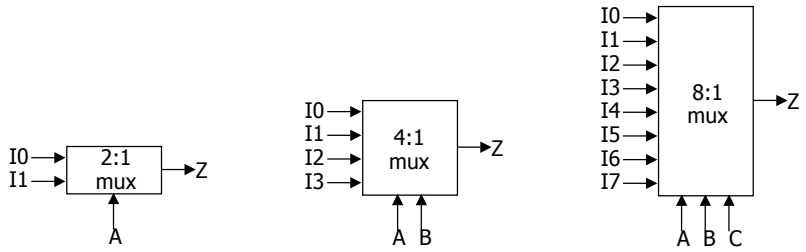
logical form

I_1	I_0	A	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

two alternative forms
for a 2:1 Mux truth table

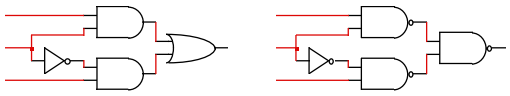
Multiplexers/Selectors (cont'd)

- 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

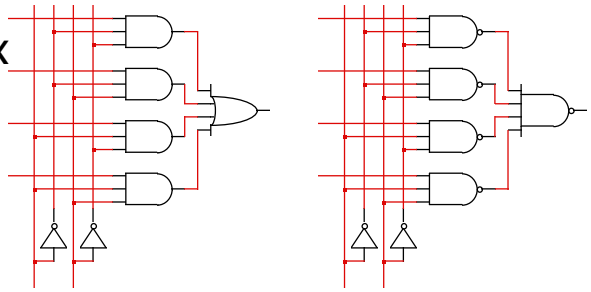


Gate Level Implementation

- 2:1 mux

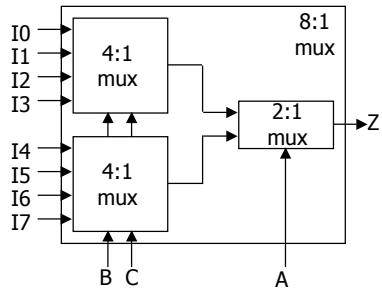


- 4:1 mux



Cascading Multiplexers

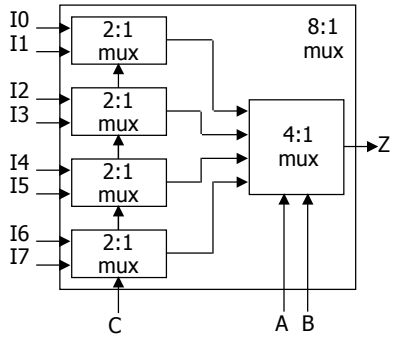
- Large multiplexers implemented by cascading smaller ones



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

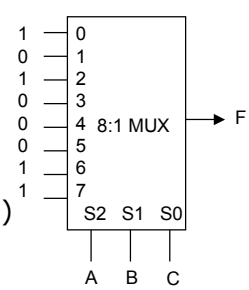
control signal A chooses which of the upper or lower mux's output to gate to Z

alternative implementation



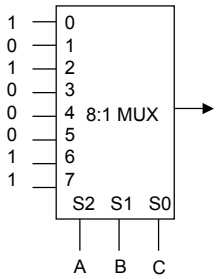
Multiplexers as Logic

- 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)

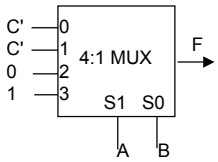


Multiplexers as Logic

- 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) = m0 + m2 + m6 + m7
    = A'B'C' + A'BC' + ABC' + ABC
    = A'B'(C') + A'B(C') + AB'(0) + AB(1)



A	B	C	F	
0	0	0	1	C'
0	0	1	0	
0	1	0	1	C'
0	1	1	0	
1	0	0	0	0
1	0	1	0	
1	1	0	1	1
1	1	1	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)

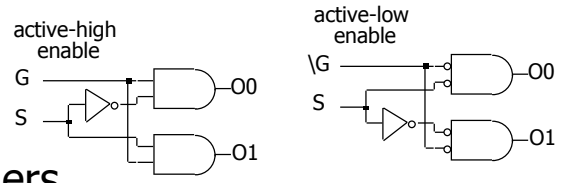
1:2 Decoder:
O0 = G • S′
O1 = G • S

2:4 Decoder:
O0 = G • S1′ • S0′
O1 = G • S1′ • S0
O2 = G • S1 • S0′
O3 = G • S1 • S0

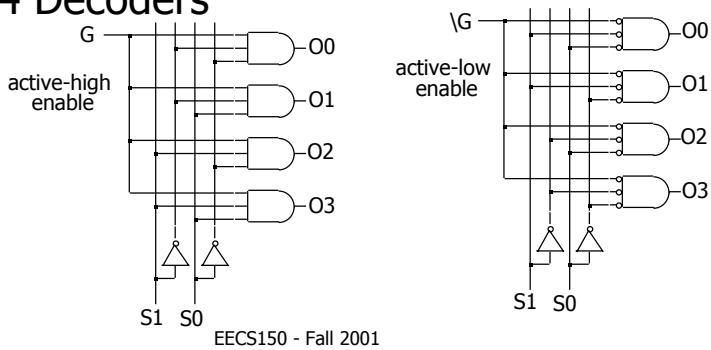
3:8 Decoder:
O0 = G • S2′ • S1′ • S0′
O1 = G • S2′ • S1′ • S0
O2 = G • S2′ • S1 • S0′
O3 = G • S2′ • S1 • S0
O4 = G • S2 • S1′ • S0′
O5 = G • S2 • S1′ • S0
O6 = G • S2 • S1 • S0′
O7 = G • S2 • S1 • S0

Implementation of Demultiplexers

- 1:2 Decoders

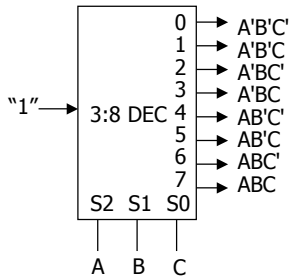


- 2:4 Decoders



Demultiplexers as Logic

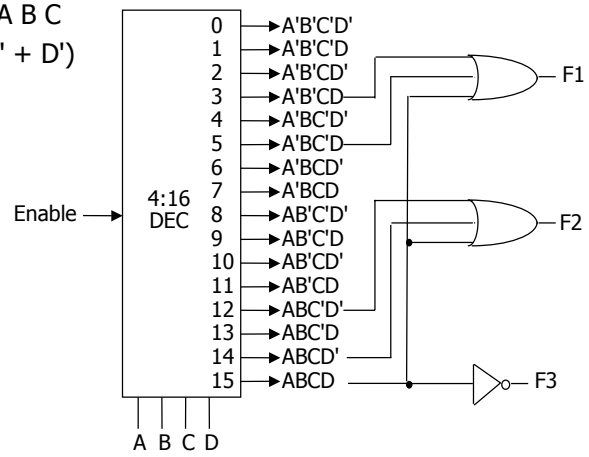
- n:2^n decoder implements any function of n variables
  - With the variables used as control inputs
  - Enable inputs tied to 1 and
  - Appropriate minterms summed to form the function



demultiplexer generates appropriate minterm based on control signals (it "decodes" control signals)

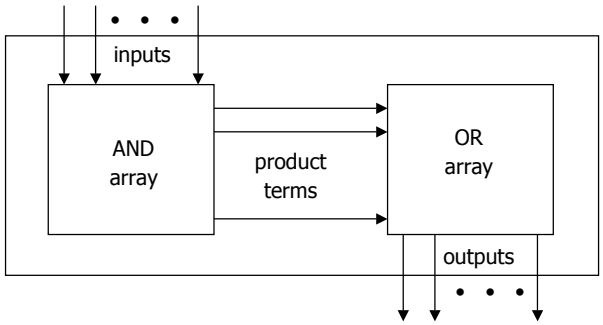
Demultiplexers as Logic

- F1 = A' B C' D + A' B' C D + A B C D
- F2 = A B C' D′ + A B C
- F3 = (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:

F0 = A + B' C'
F1 = A C' + A B
F2 = B' C' + A B
F3 = B' C + A

personality matrix

product term	inputs A	B	C	outputs F0	F1	F2	F3
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

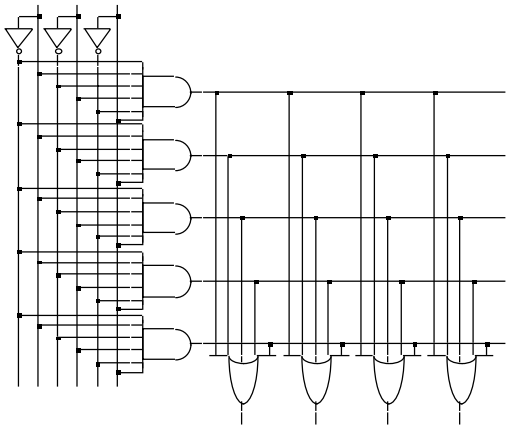
output side:

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

reuse of terms

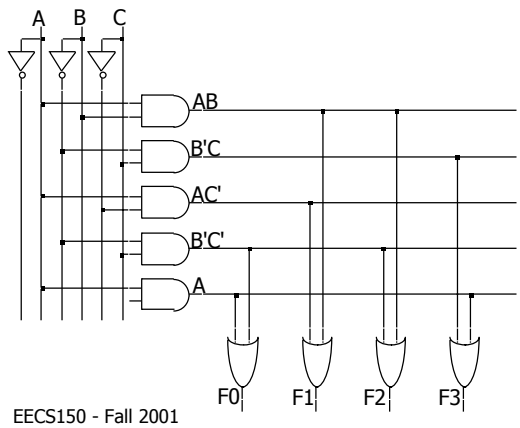
Before Programming

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



After Programming

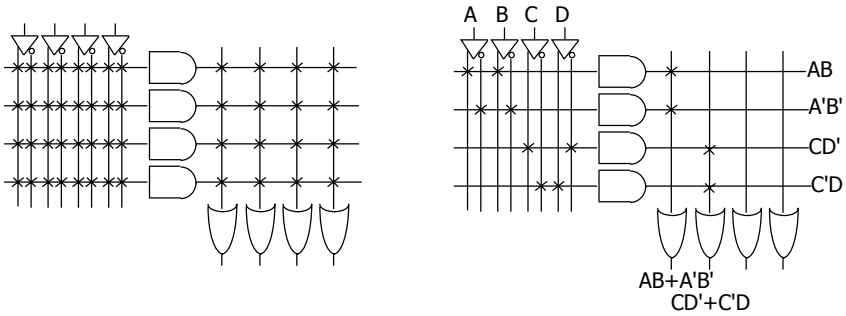
- Unwanted connections are "blown"
  - Fuse (normally connected, break unwanted ones)
  - Anti-fuse (normally disconnected, make wanted connections)



Alternate Representation

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

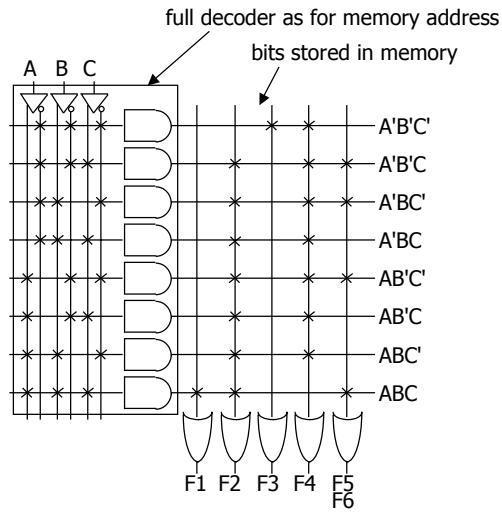
notation for implementing
F0 = A B + A' B'
F1 = C D' + C' D



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

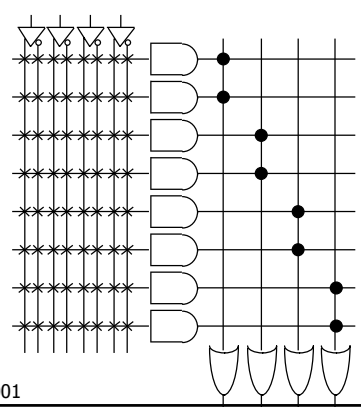
A	B	C	F1	F2	F3	F4	F5	F6
0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1	1



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

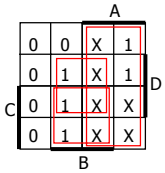
a given column of the OR array has access to only a subset of the possible product terms



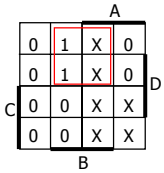
PALs and PLAs: Design Example

- BCD to Gray code converter

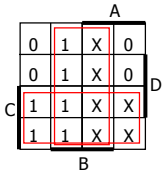
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	–	–	–	–	–
1	1	–	–	–	–	–	–



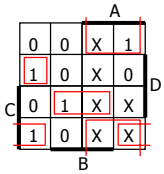
K-map for W



K-map for X



K-map for Y



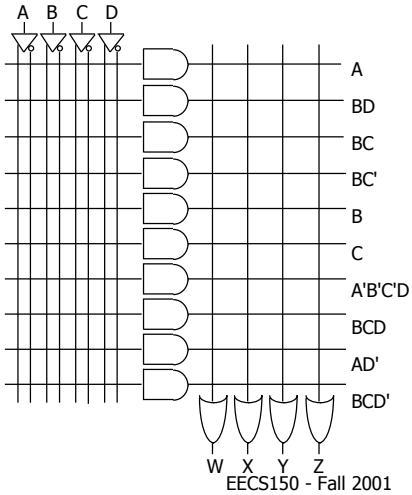
K-map for Z

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

- Code converter: programmed PLA



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

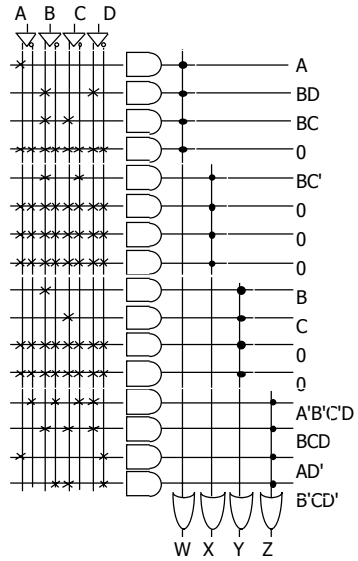
not a particularly good candidate for PAL/PLA implementation since no terms are shared among outputs

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

PALs and PLAs: Design Example

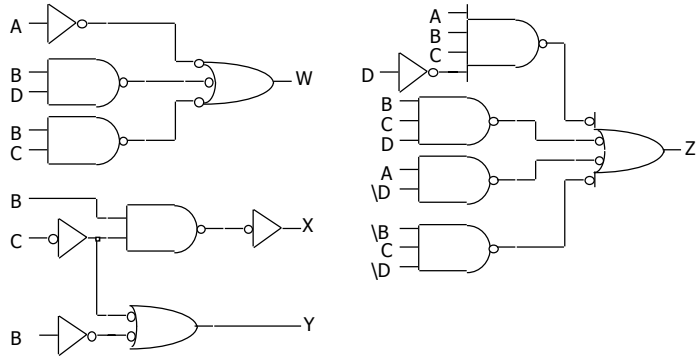
- Code converter: programmed PAL

4 product terms per each OR gate



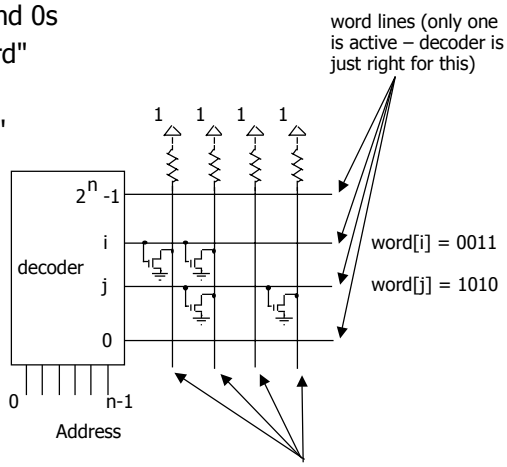
PALs and PLAs: Design Example

- 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



internal organization

ROMs and Combinational Logic

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

F0 = A' B' C + A B' C' + A B' C

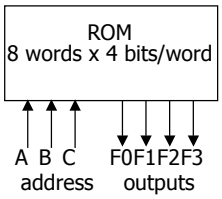
F1 = A' B' C + A' B C' + A B C

F2 = A' B' C' + A' B' C + A B' C'

F3 = A' B C + A B' C' + A B C'

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

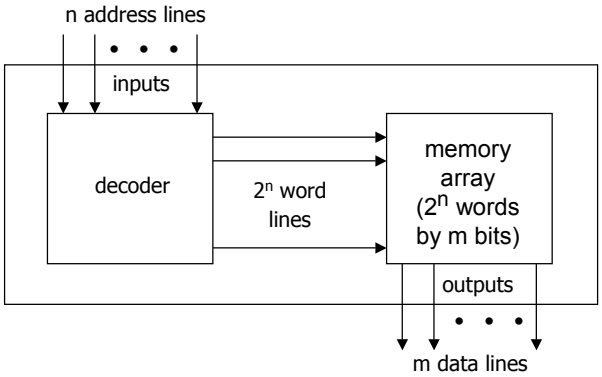
truth table



block diagram

ROM Structure

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



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)

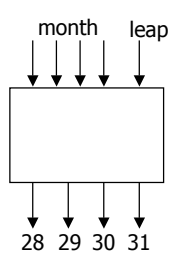
Calendar Subsystem

- 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

```
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:



month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

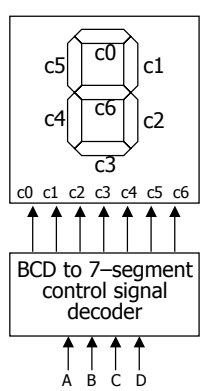
Perform Mapping

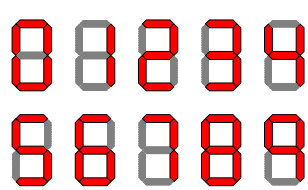
- Discrete gates
  - 28 = m8′ m4′ m2 m1′ leap′
  - 29 = m8′ m4′ m2 m1′ leap
  - 30 = m8′ m4 m1′ + m8 m1
  - 31 = m8′ m1 + m8 m1′
- Can translate to S-o-P or P-o-S

month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

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





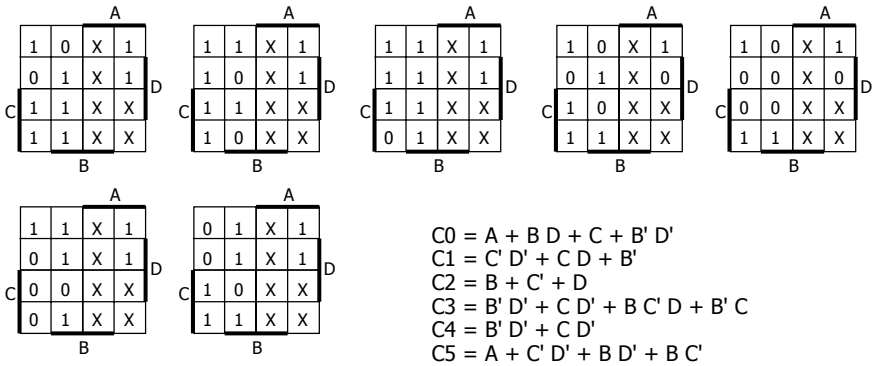
Formalize the problem

- 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

A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	–	–	–	–	–	–	–	–
1	1	–	–	–	–	–	–	–	–	–

Implementation as Minimized S-o-P

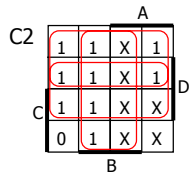
- 15 unique product terms when minimized individually

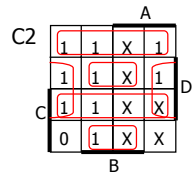


C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

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

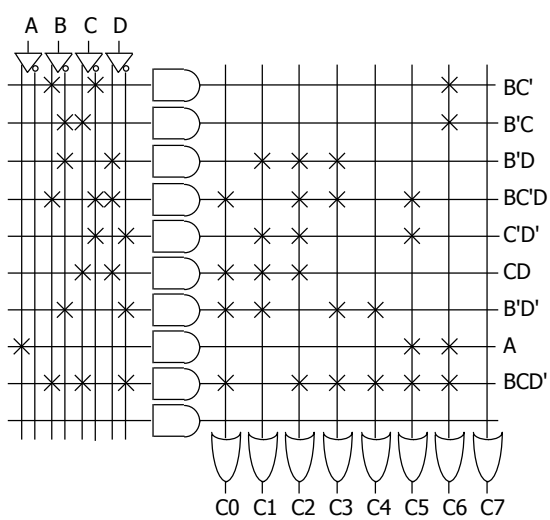




C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

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

PLA implementation



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)

    C2 = B + C' + D

    C2 = B' D + B C' D + C' D' + C D + B C D'

    C2 = B' D + B C' D + C' D' + W
    W = C D + B C D'

    need another input and another output

  - Decompose into multi-level logic (hopefully with CAD support)
    - Find common sub-expressions among functions

C0 = C3 + A' B X' + A D Y
C1 = Y + A' C5' + C' D' C6
C2 = C5 + A' B' D + A' C D
C3 = C4 + B D C5 + A' B' X'
C4 = D' Y + A' C D'
C5 = C' C4 + A Y + A' B X
C6 = A C4 + C C5 + C4' C5 + A' B' C

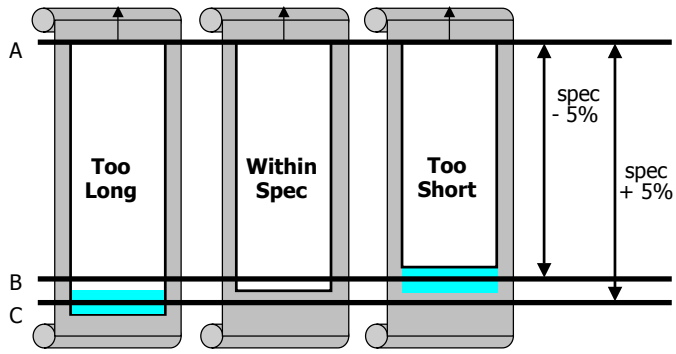
X = C' + D'
Y = B' C'

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

A	B	C	Function
0	0	0	do nothing
0	0	1	do nothing
0	1	0	do nothing
0	1	1	do nothing
1	0	0	too short
1	0	1	don't care
1	1	0	in spec
1	1	1	too long

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

C0	C1	C2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

3 control inputs: C0, C1, C2
2 data inputs: A, B
1 output: F

Formalize the Problem

C0	C1	C2	A	B	F
0	0	0	0	0	
0	1	0	0	0	
0	0	0	0	1	
0	1	0	0	1	
0	0	0	1	0	
0	1	0	1	0	
0	0	0	1	1	
0	1	0	1	1	
0	0	1	0	0	
0	1	1	0	0	
0	0	1	0	1	
0	1	1	0	1	
0	0	1	1	0	
0	1	1	1	0	
0	0	1	1	1	
0	1	1	1	1	
0	0	0	0	0	
0	1	0	0	0	
0	0	0	0	1	
0	1	0	0	1	
0	0	0	1	0	
0	1	0	1	0	
0	0	0	1	1	
0	1	0	1	1	
0	0	1	0	0	
0	1	1	0	0	
0	0	1	0	1	
0	1	1	0	1	
0	0	1	1	0	
0	1	1	1	0	
0	0	1	1	1	
0	1	1	1	1	
1	0	0	0	0	
1	1	0	0	0	

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