

# CEDT

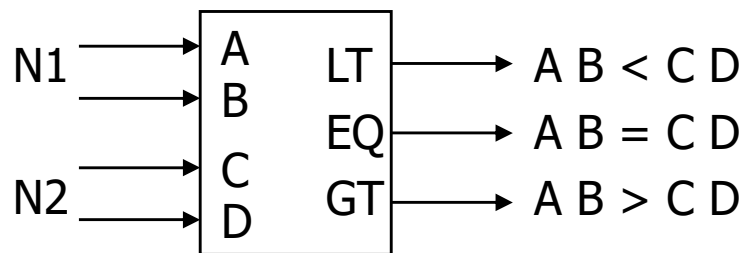
## 2110252 Digital Computer Logic

### Chap03 Working with Combinational Logic

# Working with combinational logic

- Simplification
  - two-level simplification
  - exploiting don't cares
  - algorithm for simplification
- Logic realization
  - two-level logic and canonical forms realized with NANDs and NORs
  - multi-level logic, converting between ANDs and ORs
- Time behavior
- Hardware description languages

# Design example: two-bit comparator



block diagram  
and  
truth table

A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

we'll need a 4-variable Karnaugh map  
for each of the 3 output functions

# Design example: two-bit comparator (cont'd)

A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

CD AB 00 01 11 10

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

K-map for LT

CD \ AB	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

K-map for EQ

CD \ AB	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

K-map for GT

$$LT = A' B' D + A' C + B' C D$$

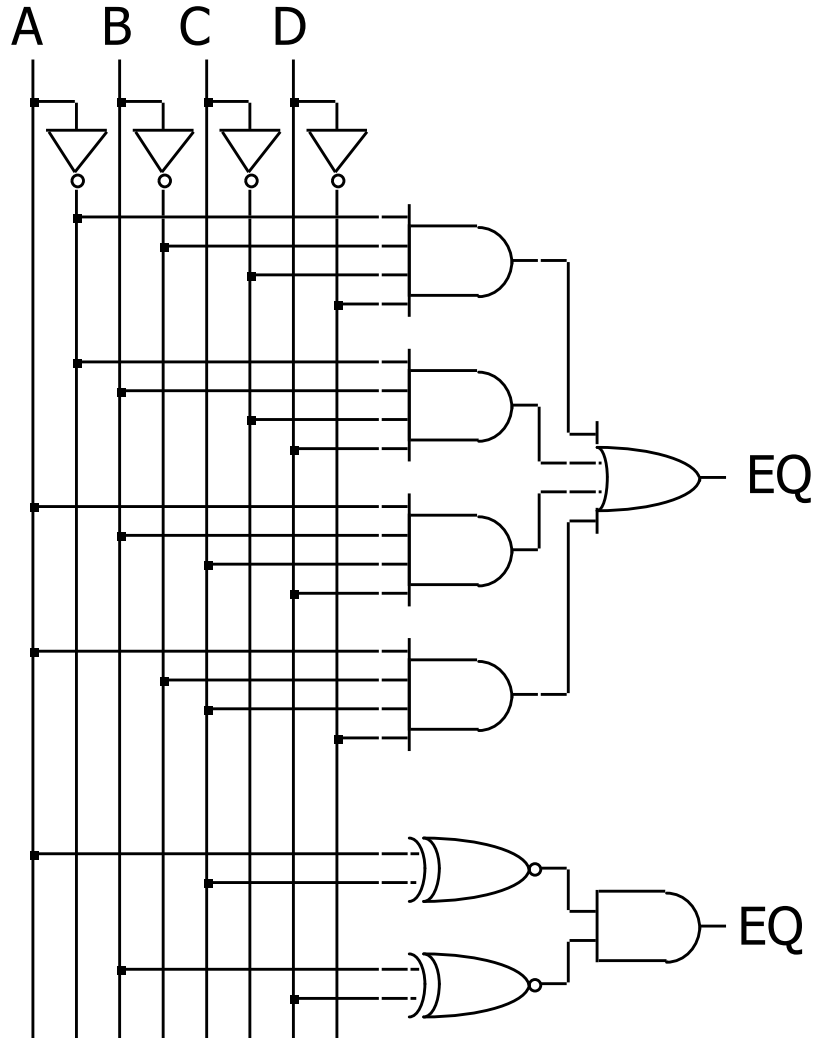
$$EQ = A' B' C' D' + A' B C' D + A B C D + A B' C D' = (A \text{ xnor } C) \bullet (B \text{ xnor } D)$$

$$GT = B C' D' + A C' + A B D'$$

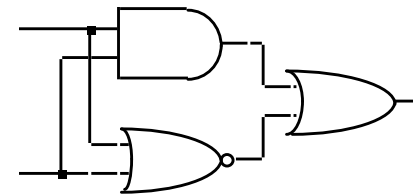
LT and GT are similar (flip A/C and B/D)

# Design example: two-bit comparator (cont'd)

$$EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A \text{ xnor } C) \bullet (B \text{ xnor } D)$$

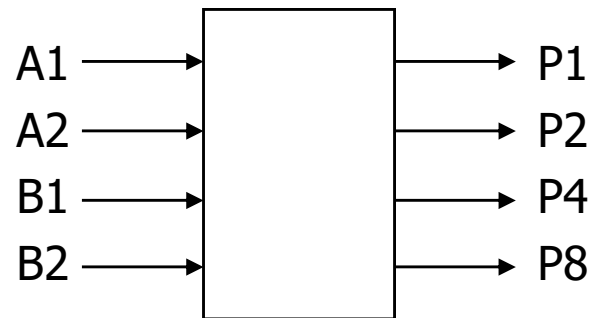


two alternative  
implementations of EQ  
with and without XOR



XNOR is implemented with  
at least 3 simple gates

# Design example: 2x2-bit multiplier



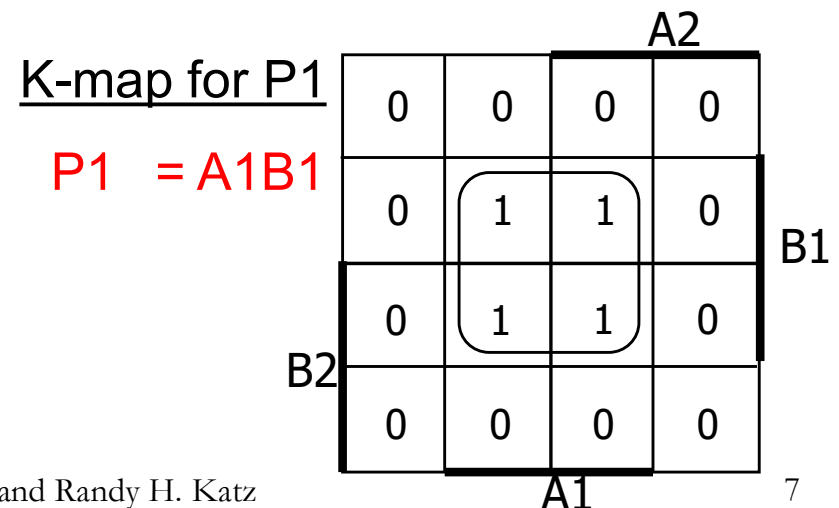
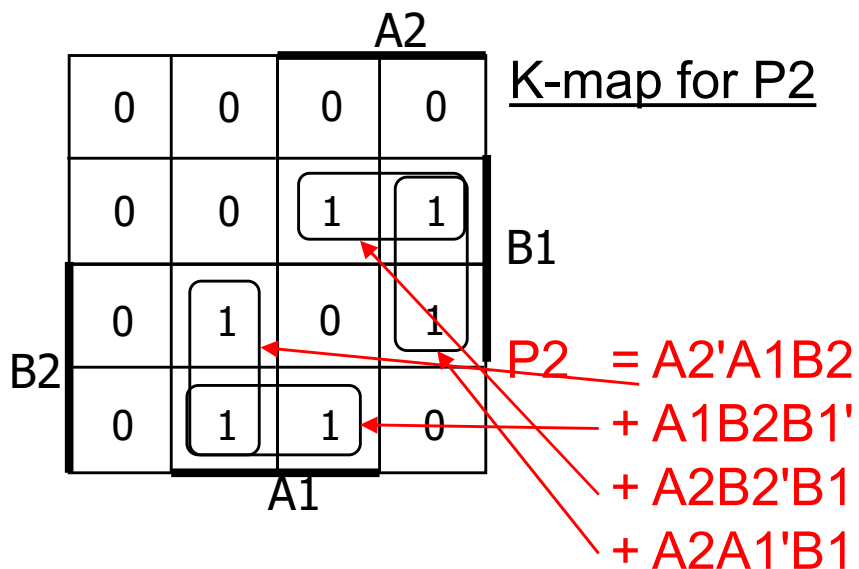
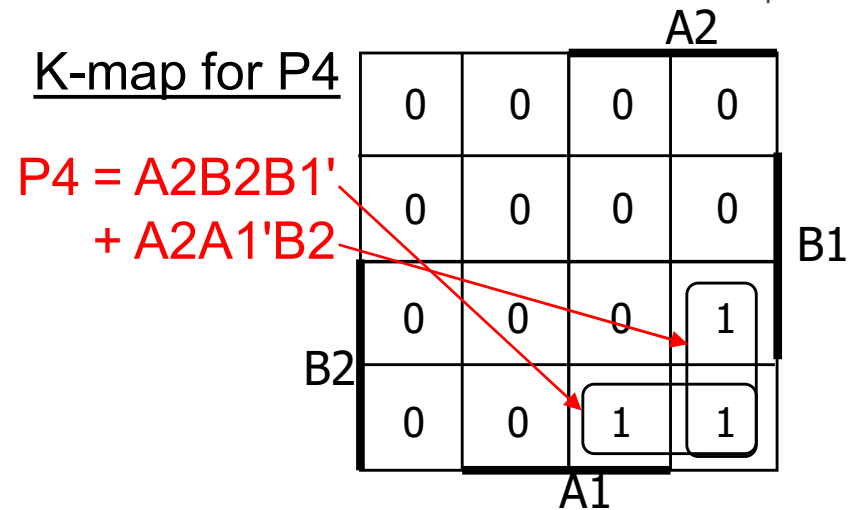
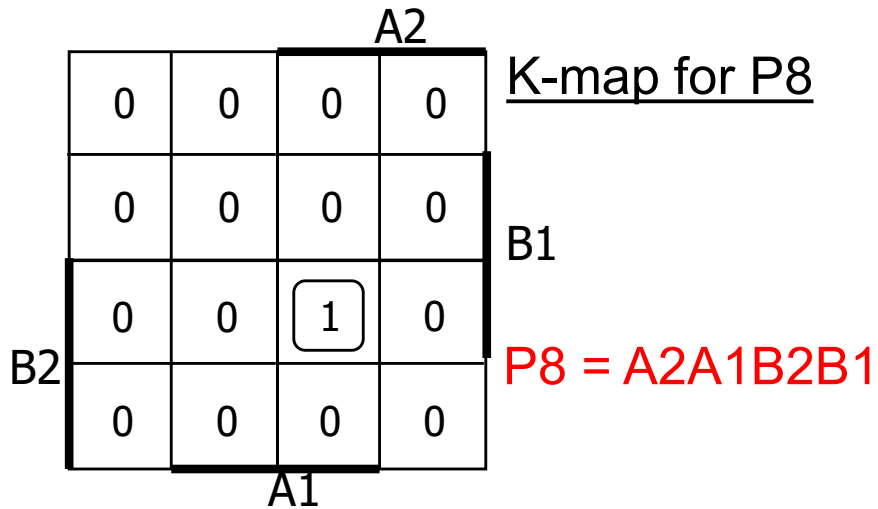
block diagram  
and  
truth table

A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

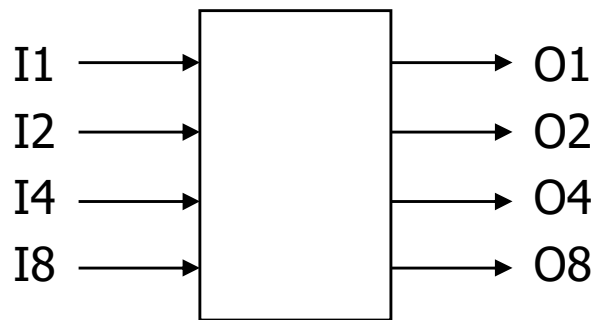
4-variable K-map  
for each of the 4  
output functions

# Design example: 2x2-bit multiplier (cont'd)

A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1



# Design example: BCD increment by 1



block diagram  
and  
truth table

I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

4-variable K-map for each of  
the 4 output functions



# Design example: BCD increment by 1 (cont'd)

				I8	
				O8	
				I1	
				I2	
0	0	X	1		
0	0	X	0		
0	1	X	X		
0	0	X	X		
				I4	

				I8	
				O4	
				I1	
				I2	
0	1	X	0		
0	1	X	0		
1	0	X	X		
0	1	X	X		
				I4	

				I8	
				O2	
				I1	
				I2	
0	0	X	0		
1	1	X	0		
0	0	X	X		
1	1	X	X		
				I4	

I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	1	0	1	1	0
0	1	1	0	1	0	0	1
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	1
1	0	1	1	0	0	0	1
1	0	1	0	1	0	0	1
1	1	0	1	X	X	X	X
1	1	1	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

				I8	
				O1	
				I1	
				I2	
1	1	X	1		
0	0	X	0		
0	0	X	X		
1	1	X	X		
				I4	

# Definition of terms for two-level simplification

- Implicant
  - single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube
- Prime implicant
  - implicant that can't be combined with another to form a larger subcube
- Essential prime implicant
  - prime implicant is essential if it alone covers an element of ON-set
  - will participate in ALL possible covers of the ON-set
  - DC-set used to form prime implicants but not to make implicant essential
- Objective:
  - grow implicant into prime implicants (minimize literals per term)
  - cover the ON-set with as few prime implicants as possible (minimize number of product terms)

# Examples to illustrate terms

		A		
	0	X	1	0
	1	1	1	0
C	1	0	1	1
	0	0	1	1
		B		

6 prime implicants:

$A'B'D$ ,  $BC'$ ,  $AC$ ,  $A'C'D$ ,  $AB$ ,  $B'CD$

essential

minimum cover:  $AC + BC' + A'B'D$

5 prime implicants:

$BD$ ,  $ABC'$ ,  $ACD$ ,  $A'BC$ ,  $A'C'D$

essential

minimum cover: 4 essential implicants

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

# Algorithm for two-level simplification

- Algorithm: minimum sum-of-products expression from a Karnaugh map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements always a power of 2)
  - Repeat Steps 1 and 2 to find all prime implicants
  - Step 3: revisit the 1s in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover
    - 1s covered by essential prime implicant do not need to be revisited
  - Step 4: if there remain 1s not covered by essential prime implicants
    - select the smallest number of prime implicants that cover the remaining 1s

# Algorithm for two-level simplification (example)

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

**2 primes around  $A'BC'D'$**

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

**2 primes around  $ABC'D$**

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

**3 primes around  $AB'C'D'$**

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

**2 essential primes**

		A		
	X	1	0	1
	0	1	1	1
C	0	X	X	0
	0	1	0	1
		B		D

**minimum cover (3 primes)**

# Activity

- List all prime implicants for the following K-map:

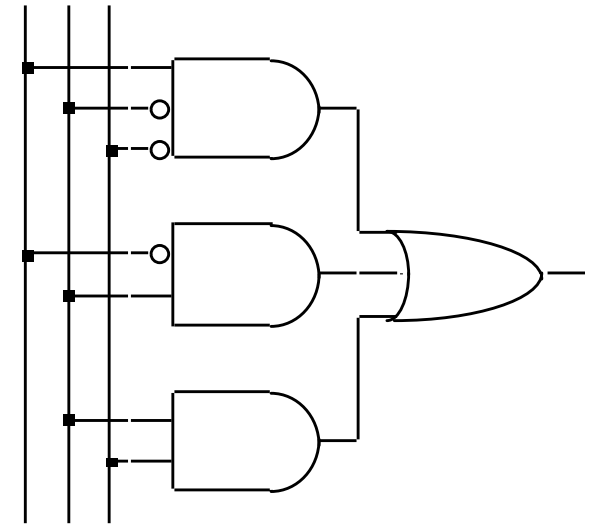
		A		D
C	X	0	X	0
	0	1	X	1
	0	X	X	0
	X	1	1	1
		B		

- Which are essential prime implicants?
- What is the minimum cover?

# Implementations of two-level logic

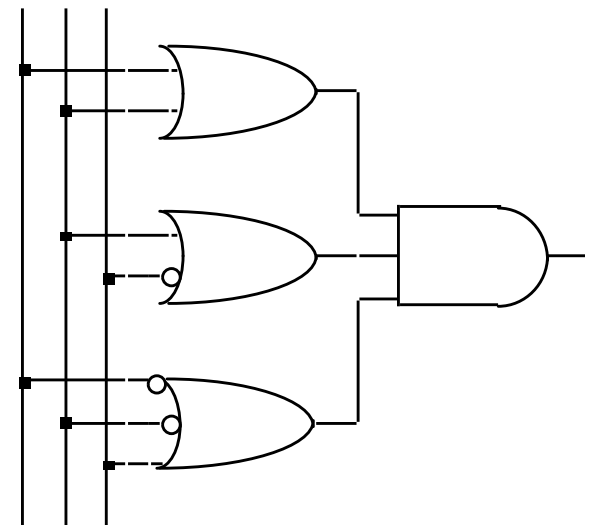
## ■ Sum-of-products

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



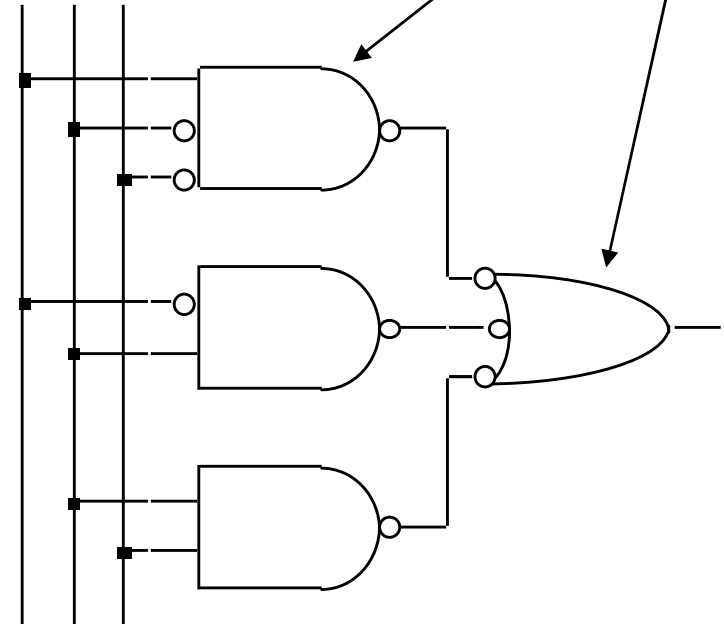
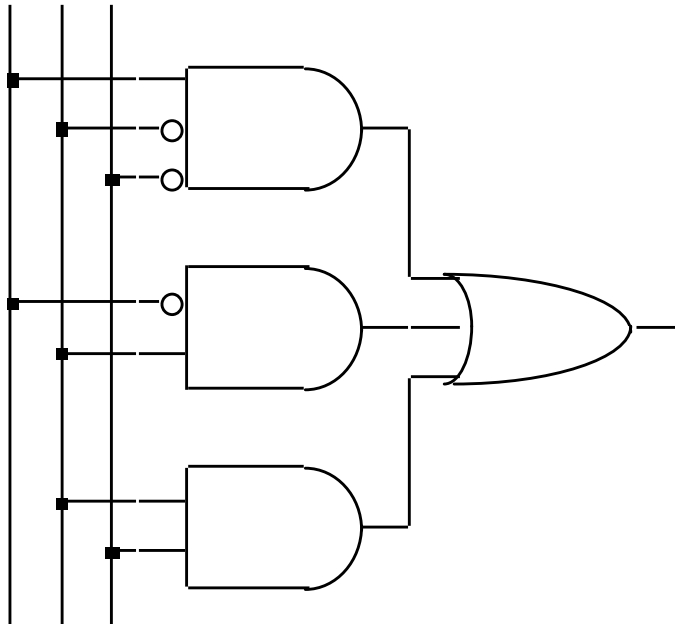
## ■ Product-of-sums

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



# Two-level logic using NAND gates

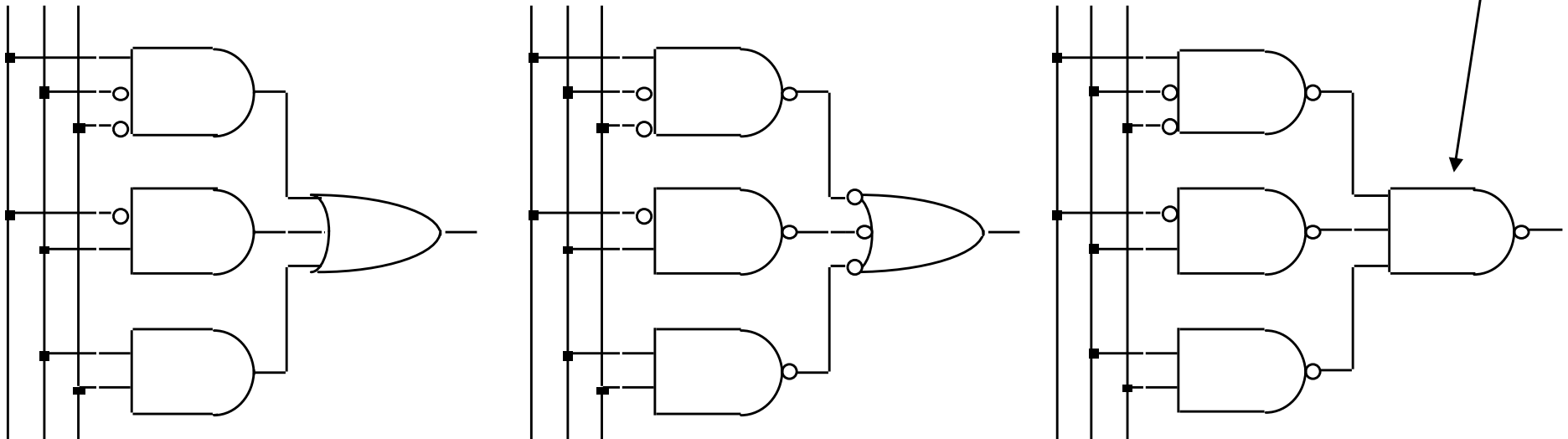
- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate





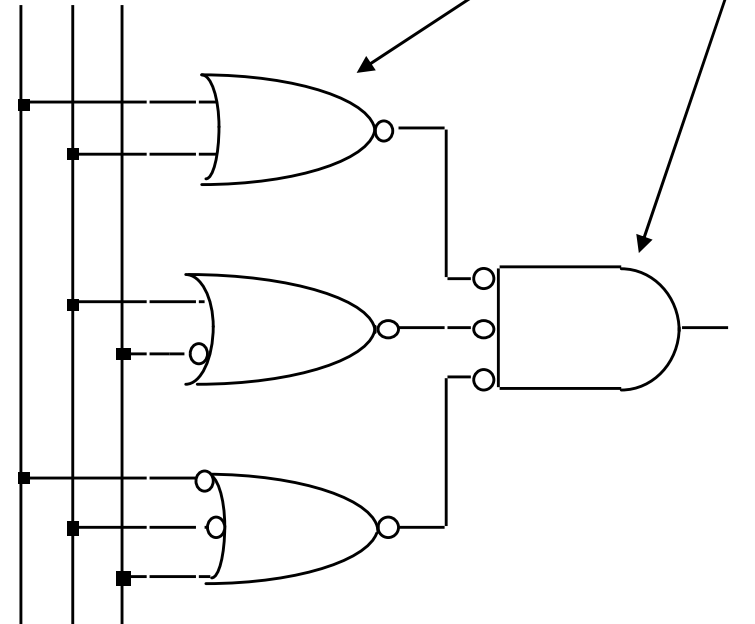
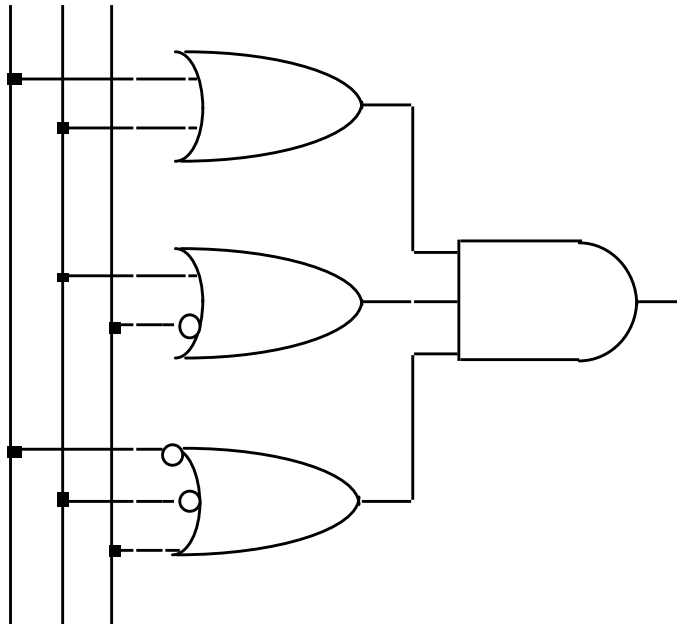
# Two-level logic using NAND gates (cont'd)

- 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



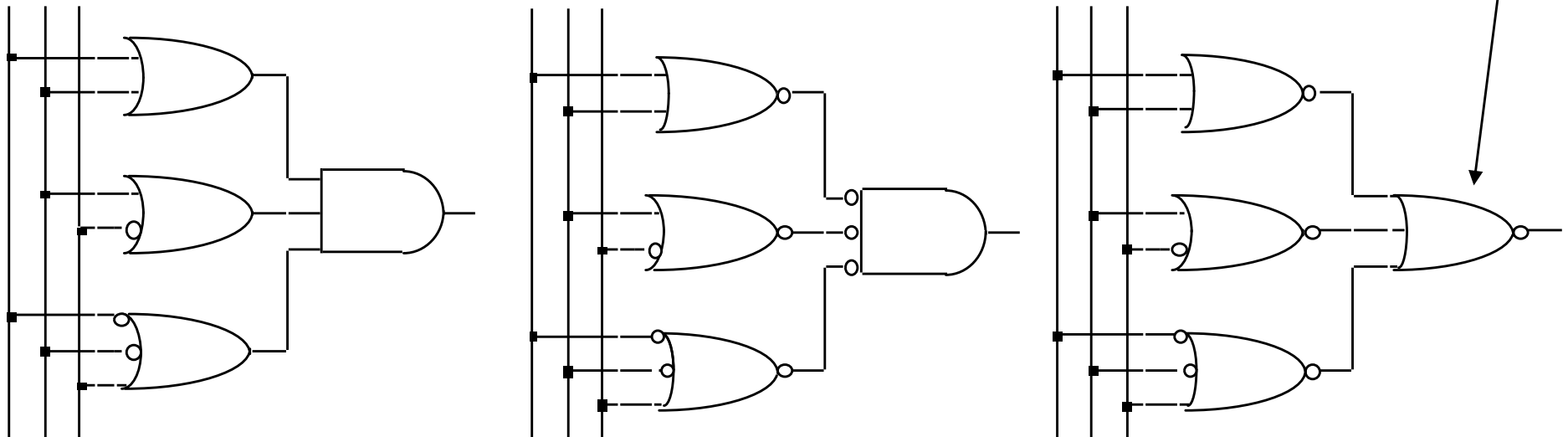
# Two-level logic using NOR gates

- 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



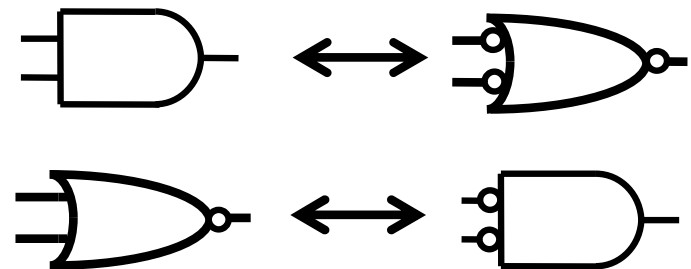
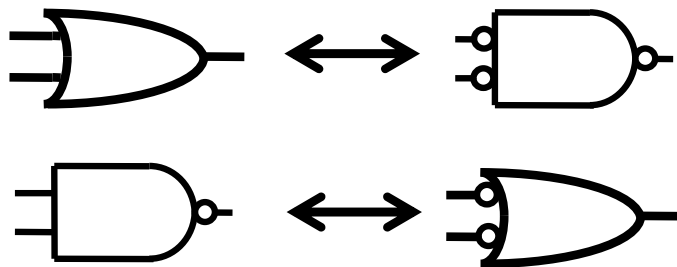
# Two-level logic using NAND and NOR gates

## ■ NAND-NAND and NOR-NOR networks

- de Morgan's law:  $(A + B)' = A' \cdot B'$        $(A \cdot B)' = A' + B'$
- written differently:  $A + B = (A' \cdot B')'$        $(A \cdot B) = (A' + B')'$

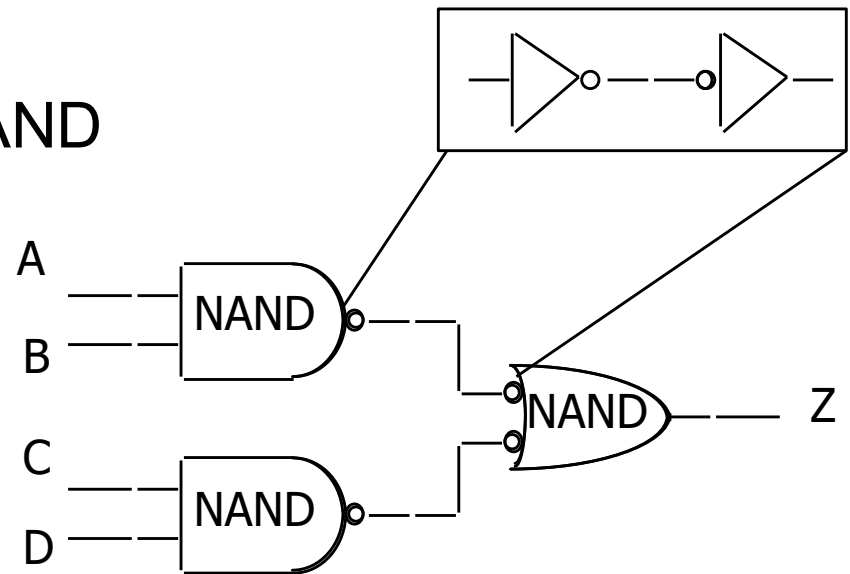
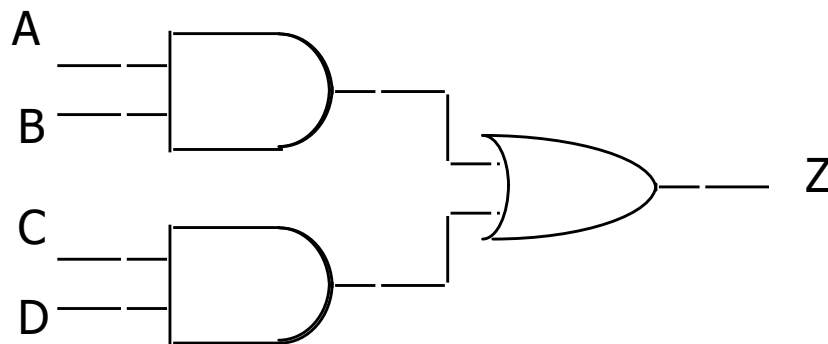
## ■ In other words —

- 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



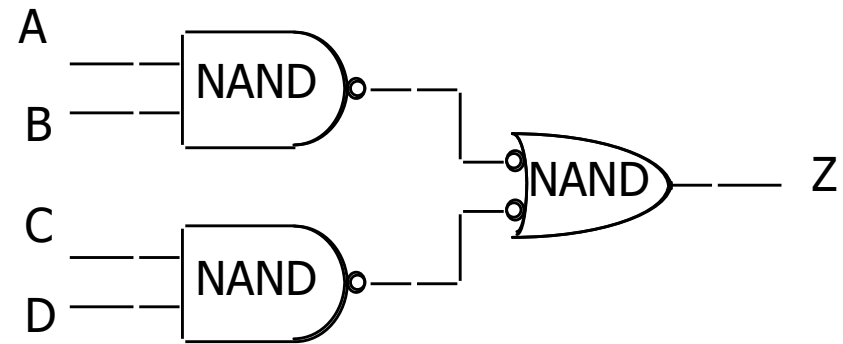
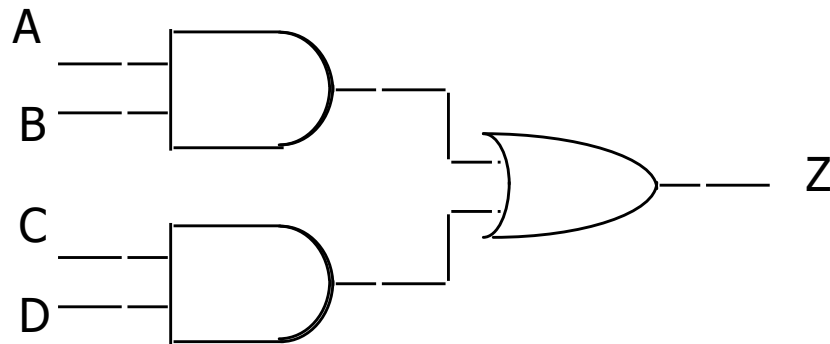
# Conversion between forms

- 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



# Conversion between forms (cont'd)

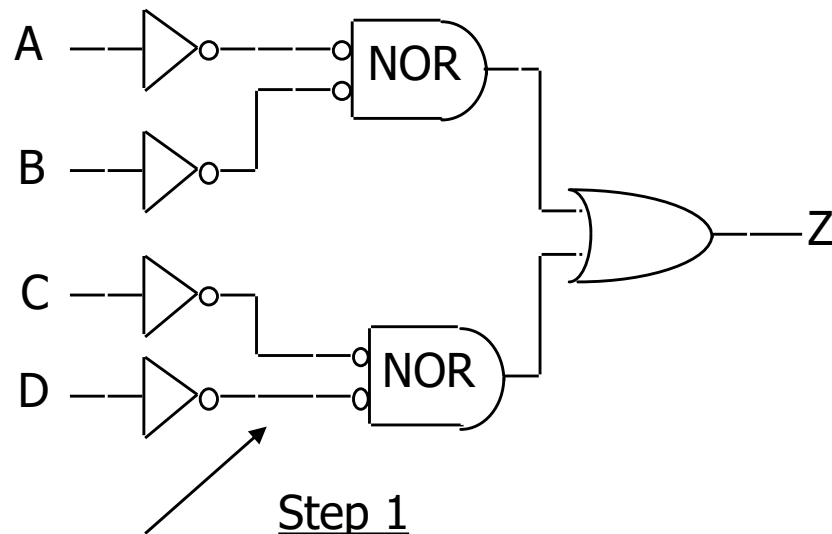
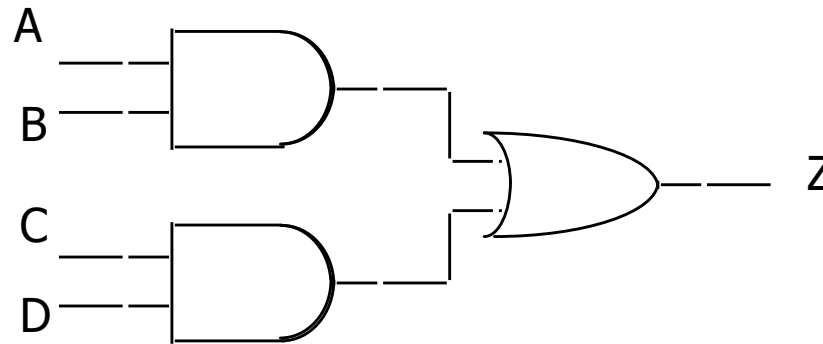
- Example: verify equivalence of two forms



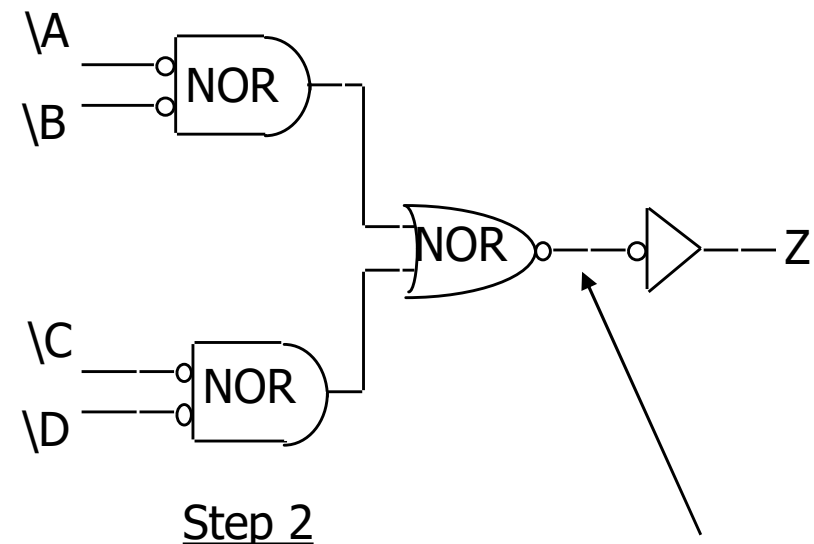
$$\begin{aligned} Z &= [ (A \cdot B)' \cdot (C \cdot D)' ]' \\ &= [ (A' + B') \cdot (C' + D') ]' \\ &= [ (A' + B')' + (C' + D')' ] \\ &= (A \cdot B) + (C \cdot D) \quad \checkmark \end{aligned}$$

# Conversion between forms (cont'd)

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



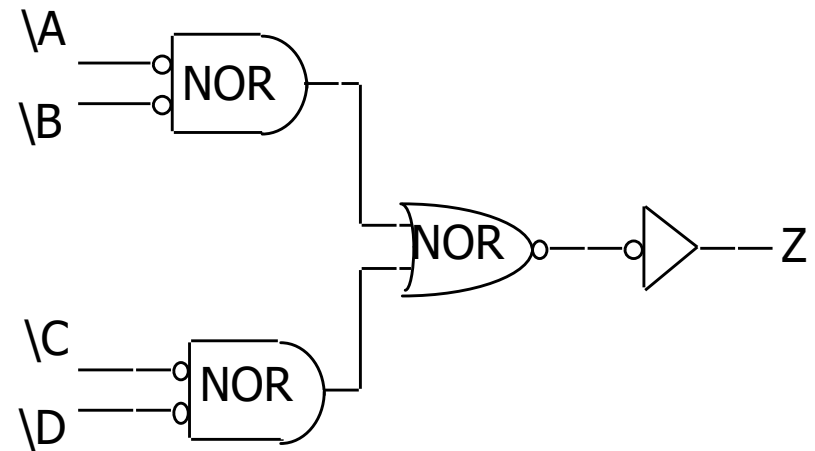
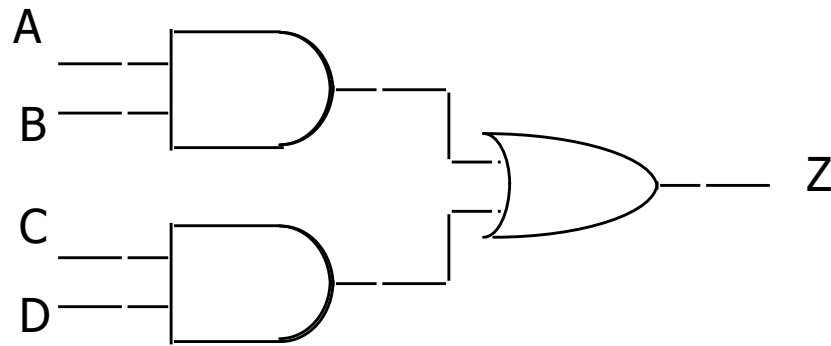
conserve  
"bubbles"



conserve  
"bubbles"

# Conversion between forms (cont'd)

- Example: verify equivalence of two forms

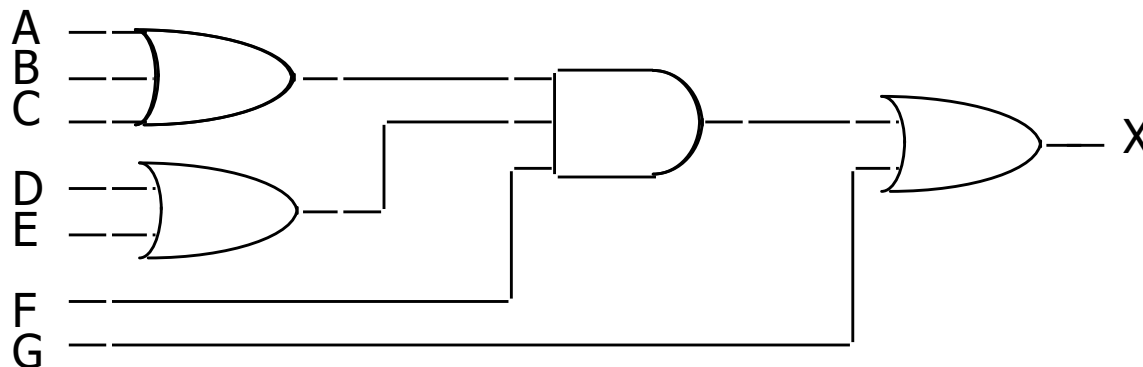


$$\begin{aligned} Z &= \{ [ (A' + B')' + (C' + D')' ]' \}' \\ &= \{ (A' + B') \cdot (C' + D') \}' \\ &= (A' + B')' + (C' + D')' \\ &= (A \cdot B) + (C \cdot D) \quad \checkmark \end{aligned}$$



# 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 (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)



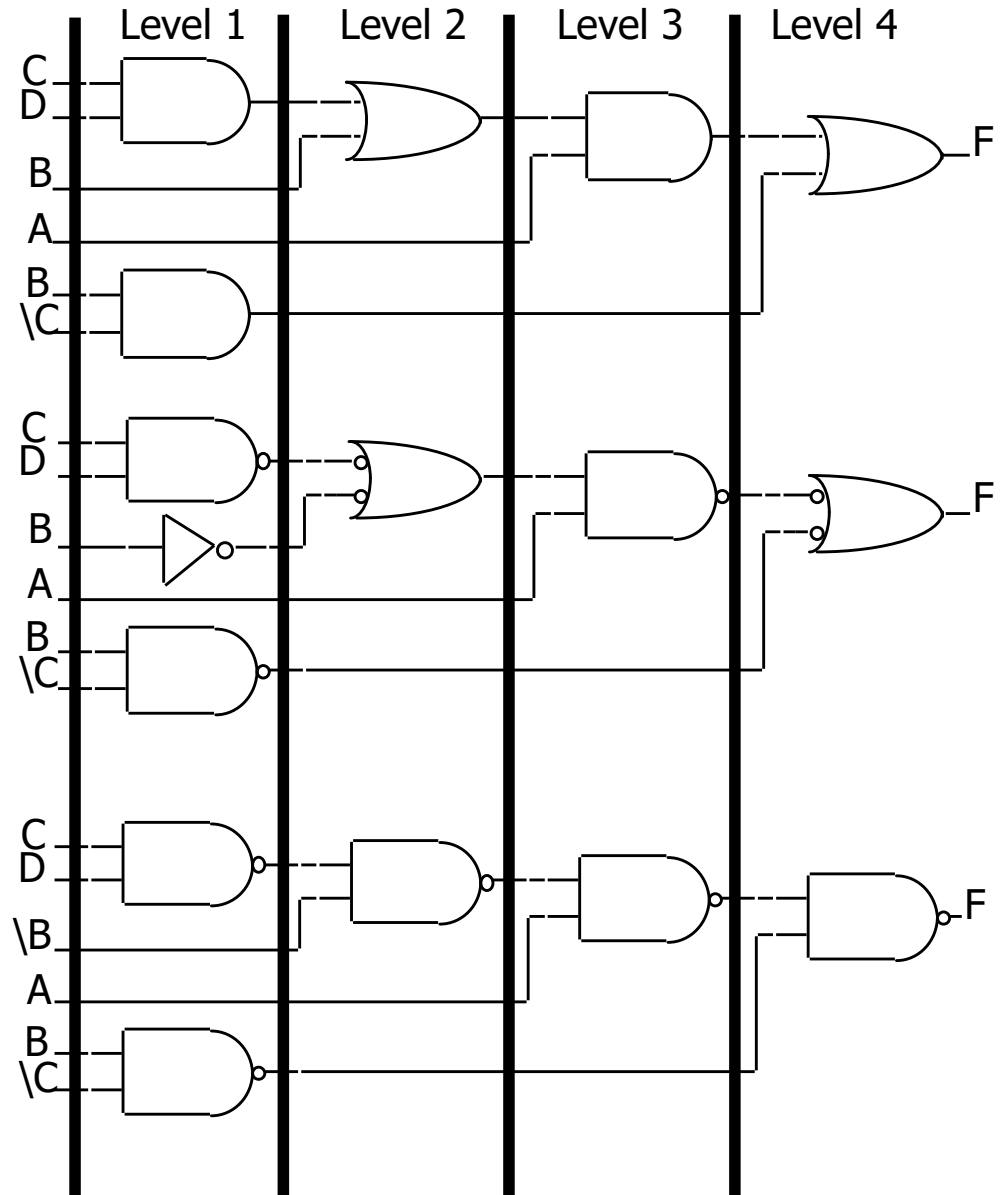
# Conversion of multi-level logic to NAND gates

■  $F = A (B + C D) + B C'$

original  
AND-OR  
network

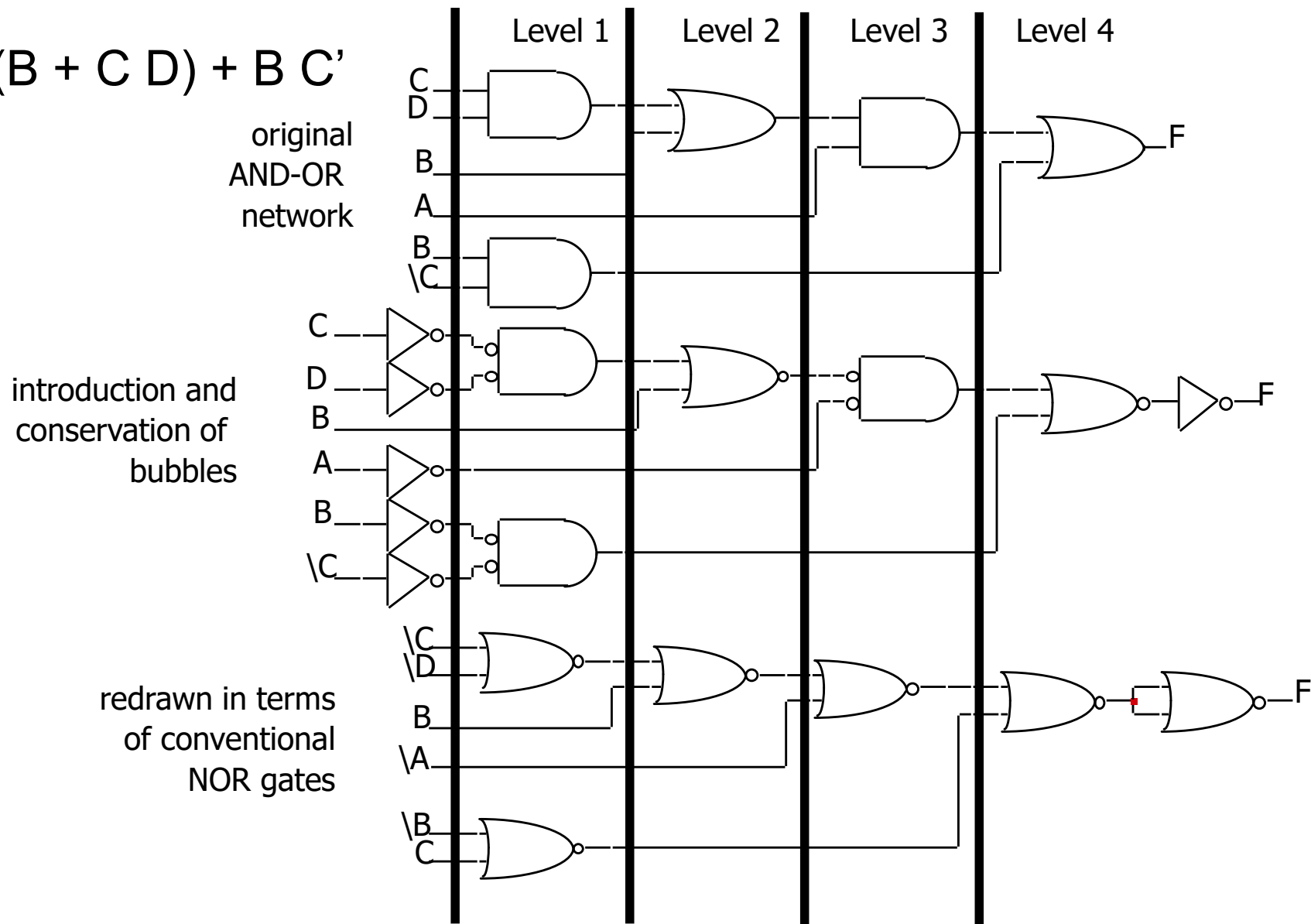
introduction and  
conservation of  
bubbles

redrawn in terms  
of conventional  
NAND gates



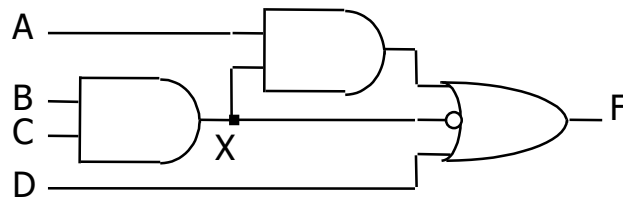
# Conversion of multi-level logic to NORs

■  $F = A(B + CD) + BC'$

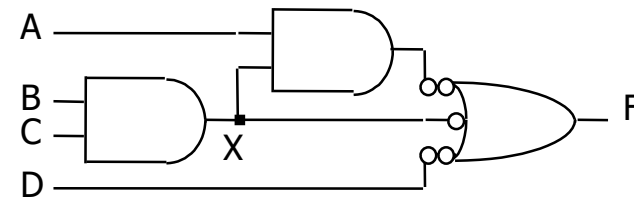


# Conversion between forms

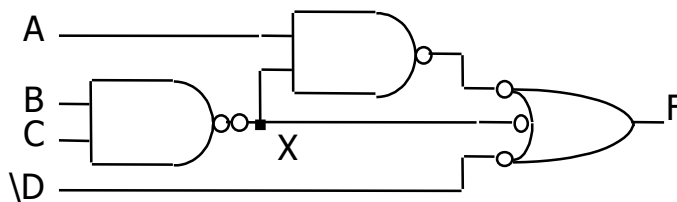
## ■ Example



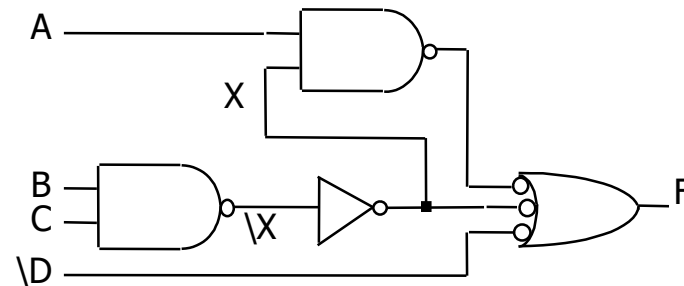
original circuit



add double bubbles to  
invert all inputs of OR gate



add double bubbles to  
invert output of AND gate

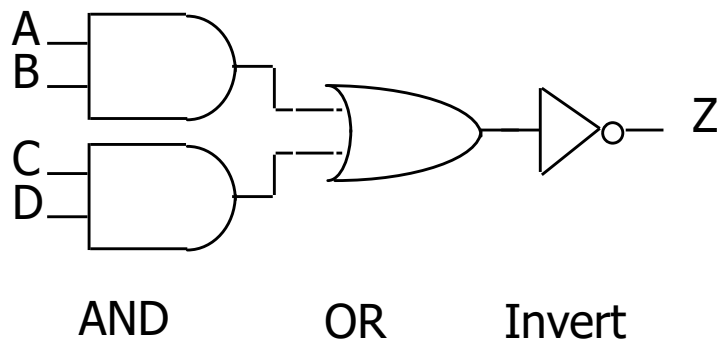


insert inverters to eliminate  
double bubbles on a wire

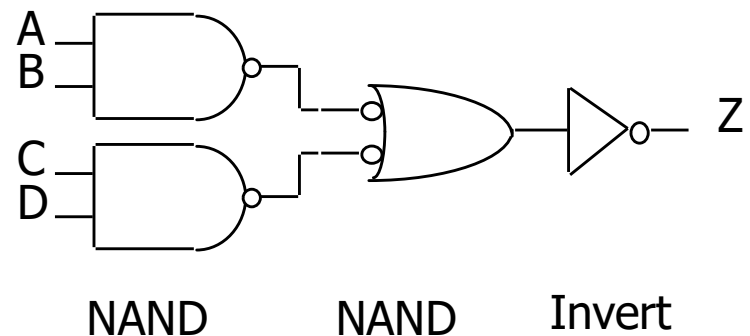
# AND-OR-invert gates

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

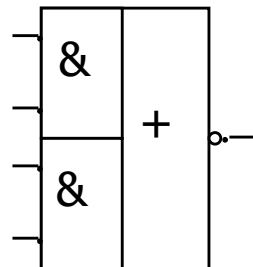
logical concept



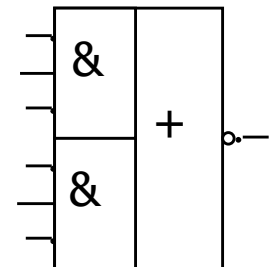
possible implementation



2x2 AOI gate symbol

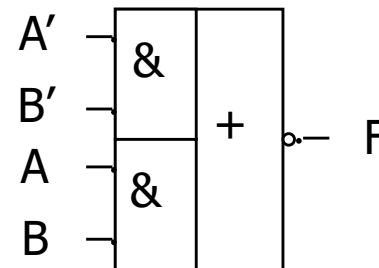


3x2 AOI gate symbol



# 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
  - $A \text{ xor } B = A' B + A B'$
  - AOI form:
    - $F = (A' B' + A B)'$



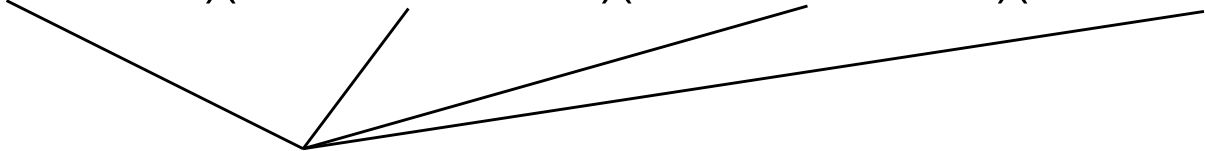
# Examples of using AOI gates

## ■ Example:

- $F = AB + AC' + BC'$
- $F = (A'B' + A'C + B'C)'$
- Implemented by 2-input 3-stack AOI gate
  
- $F = (A + B)(A + C')(B + C')$
- $F = [(A' + B')(A' + C)(B' + C)]'$
- Implemented by 2-input 3-stack OAI gate

## ■ Example: 4-bit equality function

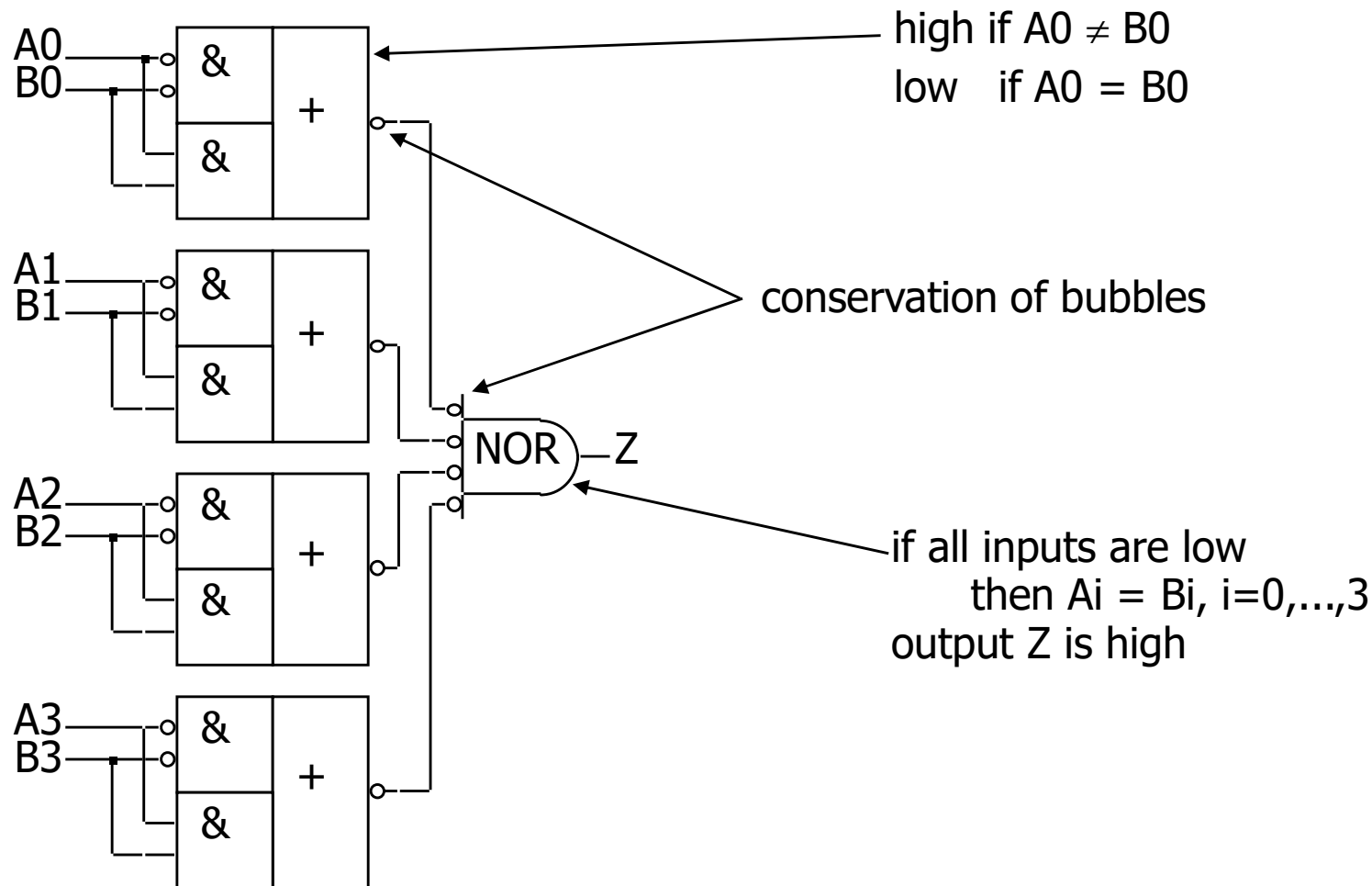
- $Z = (A_0 B_0 + A_0' B_0')(A_1 B_1 + A_1' B_1')(A_2 B_2 + A_2' B_2')(A_3 B_3 + A_3' B_3')$



each implemented in a single 2x2 AOI gate

# 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

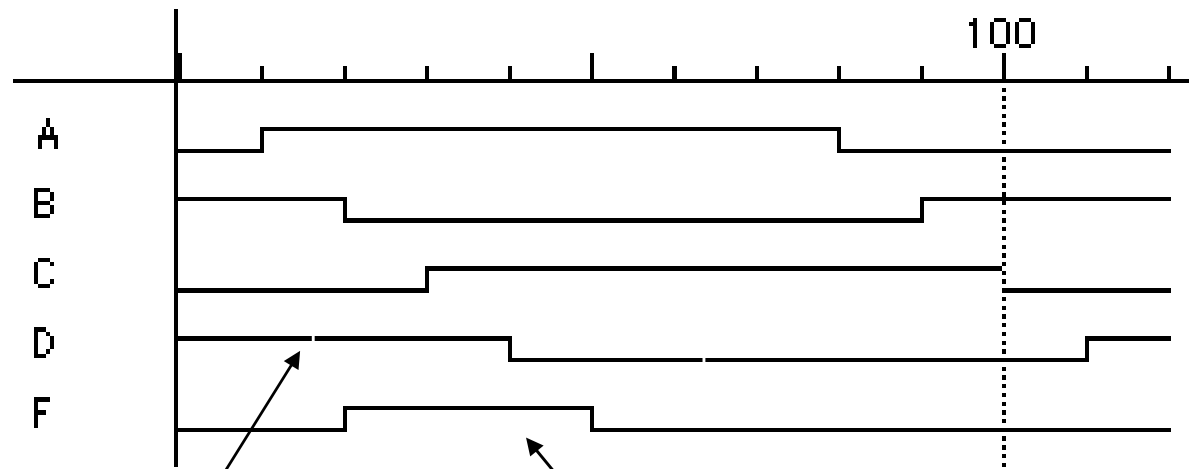
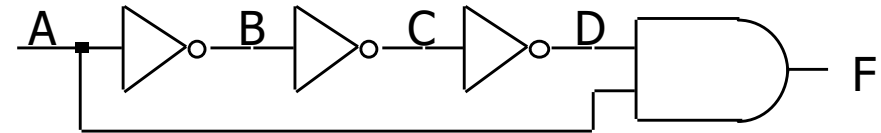
# Time behavior of combinational networks

- 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

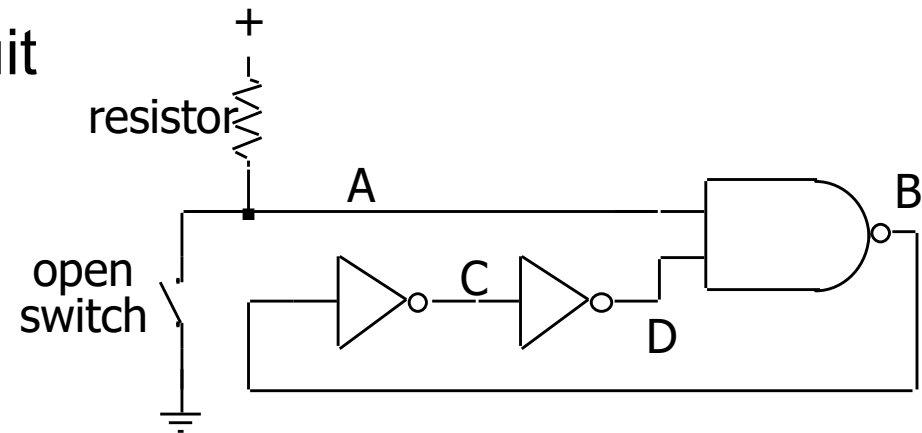


D remains high for three gate delays after A changes from low to high

F is not always 0 pulse 3 gate-delays wide

# Oscillatory behavior

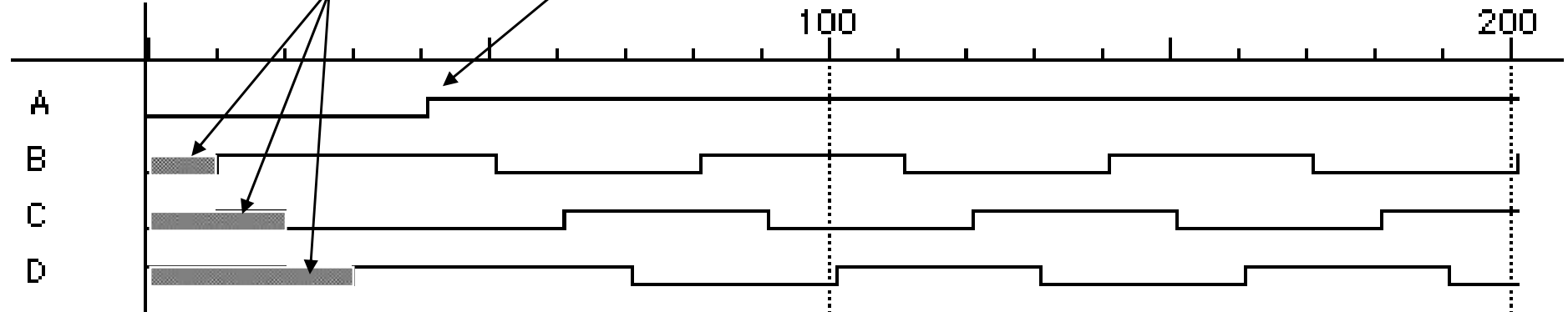
## ■ Another pulse shaping circuit



close switch

initially  
undefined

open switch



# Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

# HDLs

- Abel (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
  - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to Pascal and C
  - delays is only interaction with simulator
  - fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - similar to Ada (emphasis on re-use and maintainability)
  - simulation semantics visible
  - very general but verbose
  - IEEE standard

# Verilog

- Supports structural and behavioral descriptions
- Structural
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- Behavioral
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We'll mostly be using behavioral Verilog in Aldec ActiveHDL
  - rely on schematic when we want structural descriptions

# Structural model

```
module xor_gate (out, a, b);  
    input      a, b;  
    output     out;  
    wire       abar, bbar, t1, t2;  
  
    inverter invA (abar, a);  
    inverter invB (bbar, b);  
    and_gate and1 (t1, a, bbar);  
    and_gate and2 (t2, b, abar);  
    or_gate  or1 (out, t1, t2);  
  
endmodule
```



# Simple behavioral model

- Continuous assignment

```
module xor_gate (out, a, b);  
    input        a, b;  
    output       out;  
    reg          out;  
  
    assign #6 out = a ^ b;  
  
endmodule
```

simulation register -  
keeps track of  
value of signal

**delay from input change  
to output change**

# Simple behavioral model

- always block

```
module xor_gate (out, a, b);  
    input        a, b;  
    output       out;  
    reg          out;  
  
    always @(a or b) begin  
        #6 out = a ^ b;  
    end  
  
endmodule
```

specifies when block is executed  
ie. triggered by which signals

# Driving a simulation through a “testbench”

```
module testbench (x, y);  
    output          x, y;  
    reg [1:0]       cnt;
```

2-bit vector

```
    initial begin  
        cnt = 0;  
        repeat (4) begin  
            #10 cnt = cnt + 1;  
            $display ("@ time=%d, x=%b, y=%b, cnt=%b",  
                    $time, x, y, cnt); end  
            #10 $finish;  
        end
```

initial block executed  
only once at start  
of simulation

print to a console

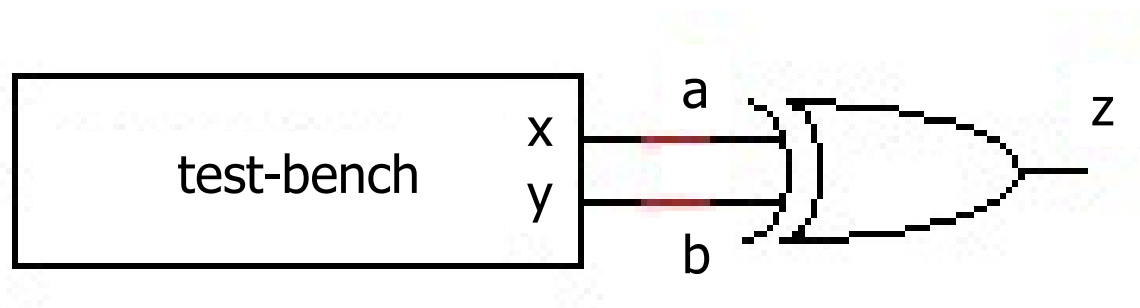
```
        assign x = cnt[1];  
        assign y = cnt[0];
```

directive to stop  
simulation

```
    endmodule
```

# Complete simulation

- Instantiate stimulus component and device to test in a schematic



# Comparator example

```
module Compare1 (Equal, Alarger, Blarger, A, B);  
    input      A, B;  
    output     Equal, Alarger, Blarger;  
  
    assign #5 Equal = (A & B) | (~A & ~B);  
    assign #3 Alarger = (A & ~B);  
    assign #3 Blarger = (~A & B);  
endmodule
```

# More complex behavioral model

```
module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
    input      n0, n1, n2, n3, n4, n5, n6, n7, self;
    output     out;
    reg        out;
    reg [7:0] neighbors;
    reg [3:0] count;
    reg [3:0] i;

    assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

    always @(neighbors or self) begin
        count = 0;
        for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
        out = (count == 3);
        out = out | ((self == 1) & (count == 2));
    end

endmodule
```

# Hardware description languages vs. programming languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL in Aldec ActiveHDL)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)

# Hardware description languages and combinational logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate's output is a function of its inputs at all times (doesn't need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks



# Working with combinational logic summary

- Design problems
  - filling in truth tables
  - incompletely specified functions
  - simplifying two-level logic
- Realizing two-level logic
  - NAND and NOR networks
  - networks of Boolean functions and their time behavior
- Time behavior
- Hardware description languages
- Later
  - combinational logic technologies
  - more design case studies