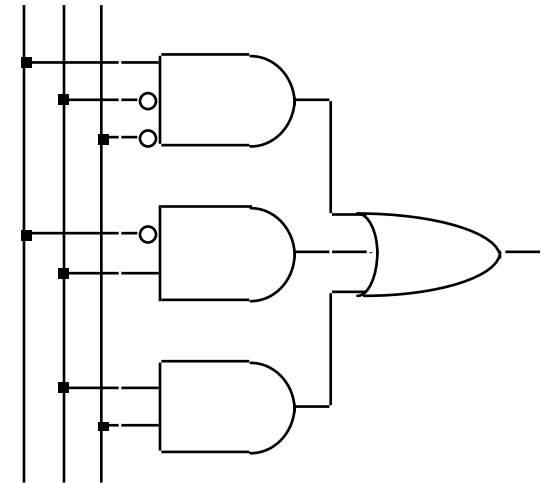# Implementations of two-level logic
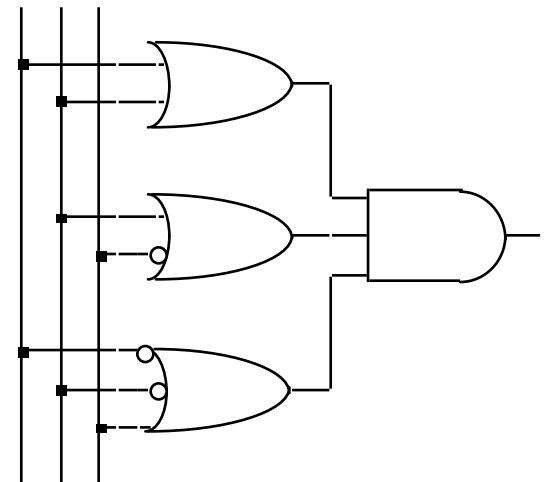
- ## Sum-of-products
  - AND gates to form product terms (minterms)
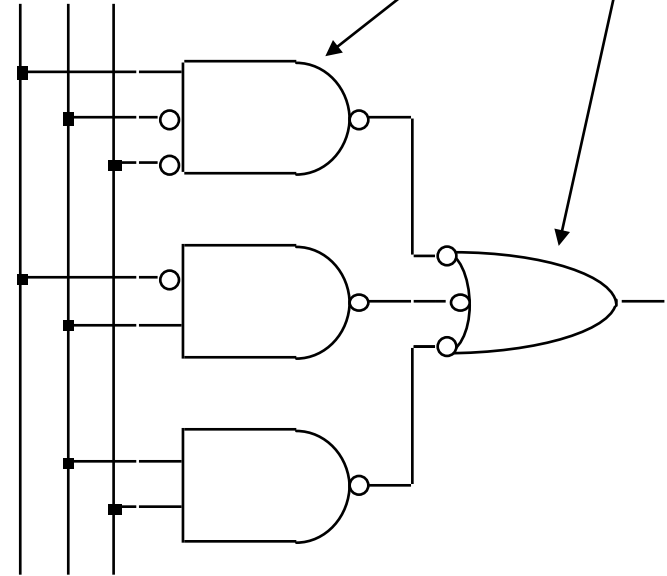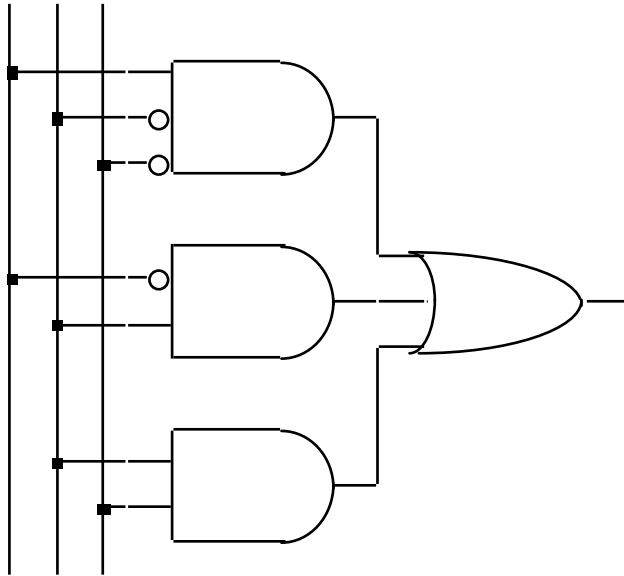 - OR gate to form sum

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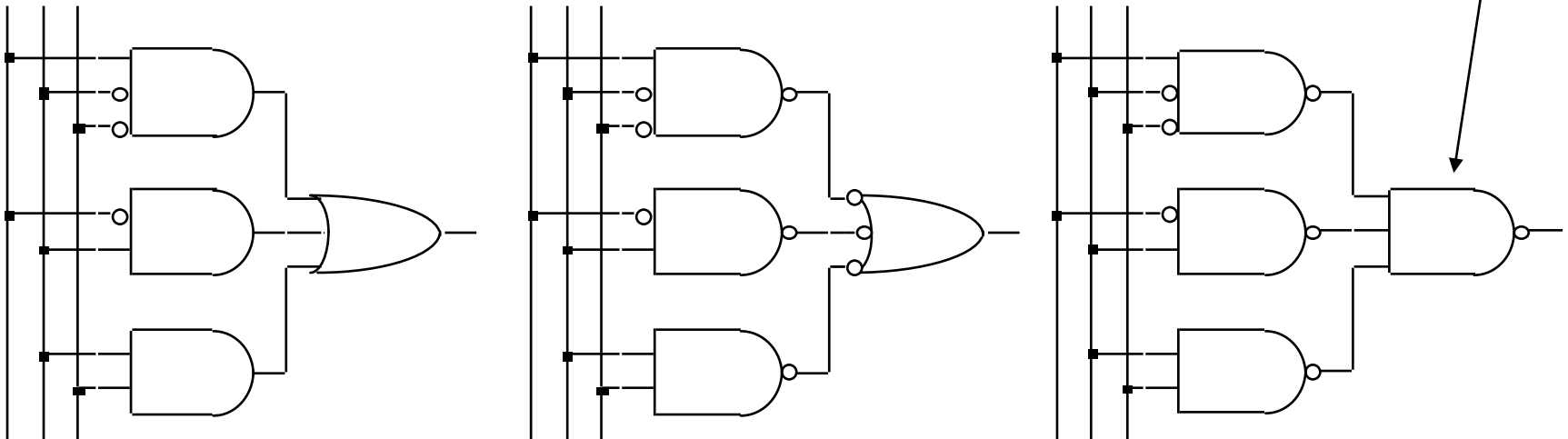 - AND gates to form product

Two-level logic using NAND gates

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

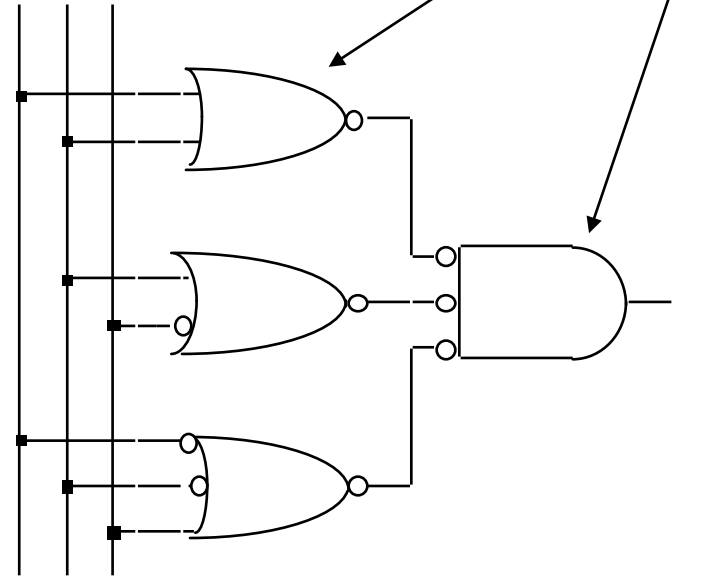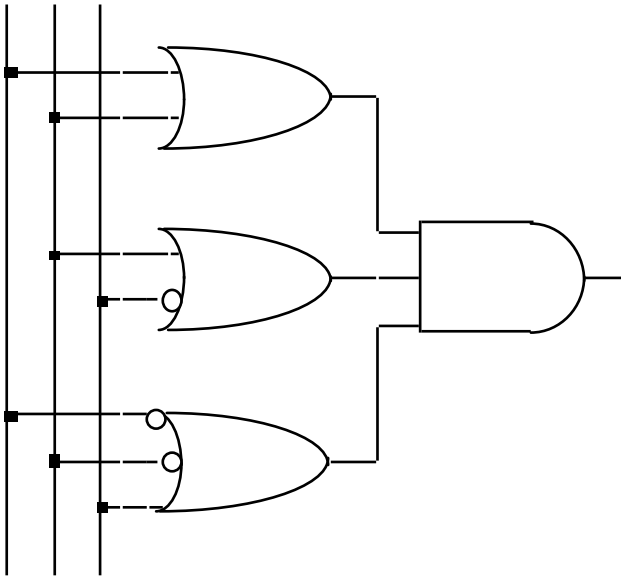# Two-level logic using NAND gates (cont'd)

- **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

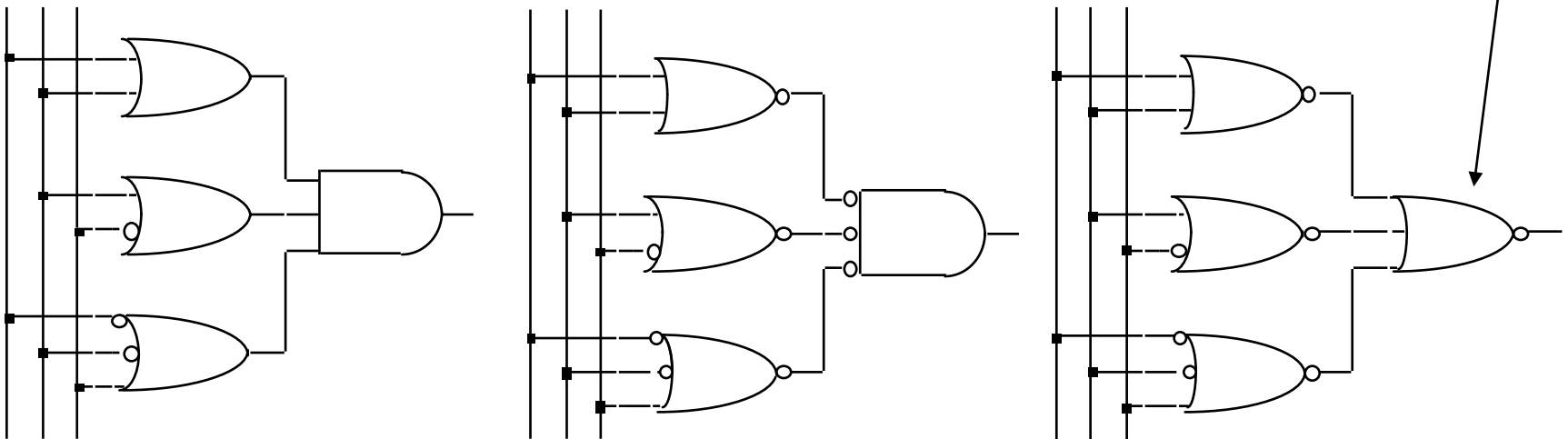# Two-level logic using NOR gates

- 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' • 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
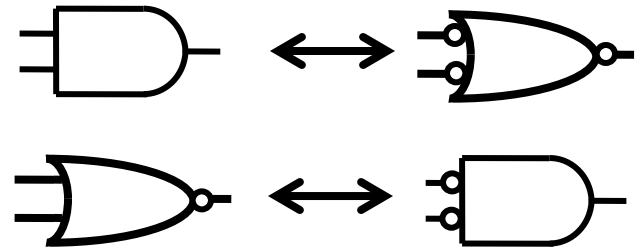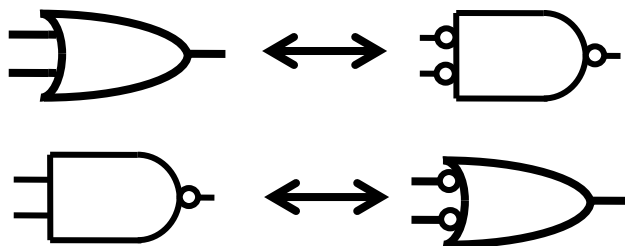
Conversion between forms

- 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
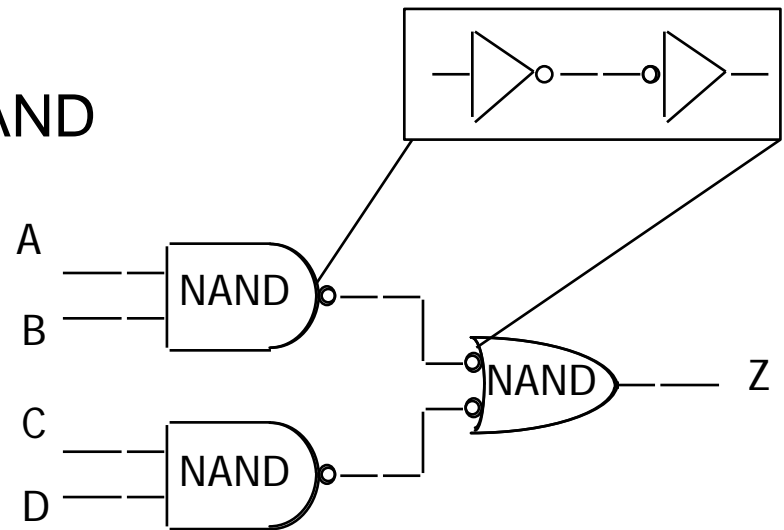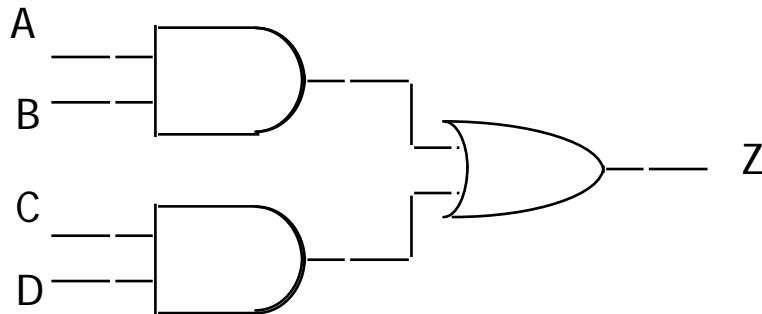
7

Conversion between forms (cont'd)

- Example: verify equivalence of two forms
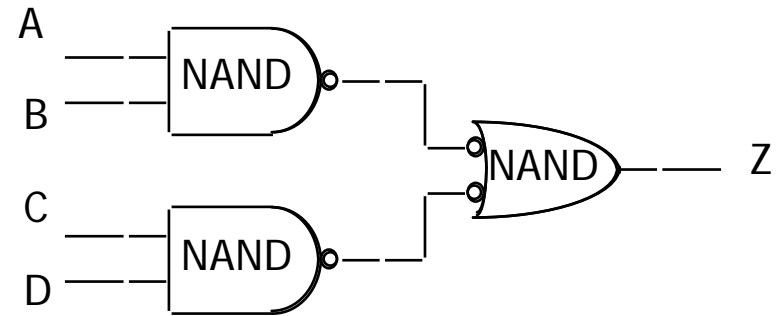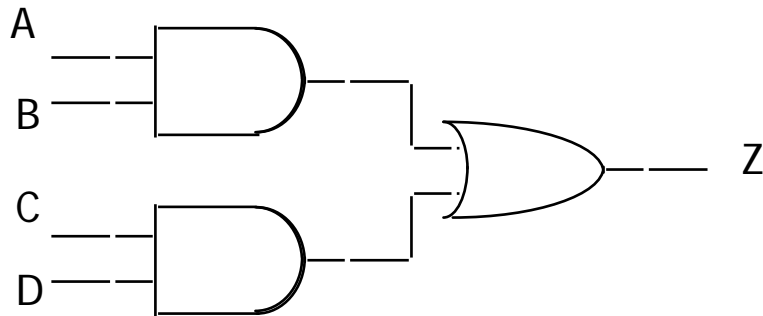
$$Z = [(A \cdot B)' \cdot (C \cdot D)']'$$
$$= [(A' + B') \cdot (C' + D')]'$$
$$= [(A' + B')' + (C' + D')']$$
$$= (A \cdot B) + (C \cdot D) \ \ddot{u}$$

Conversion between forms (cont'd)

- Example: map AND/OR network to NOR/NOR network
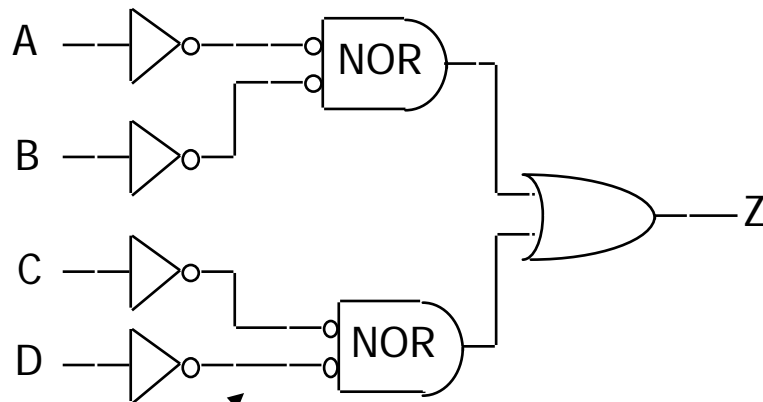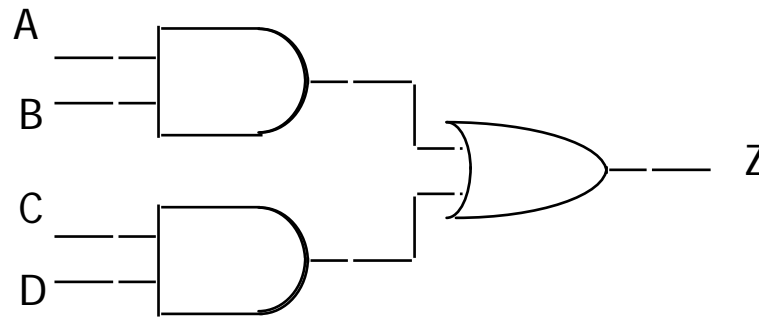
Step 1

conserve "bubbles"
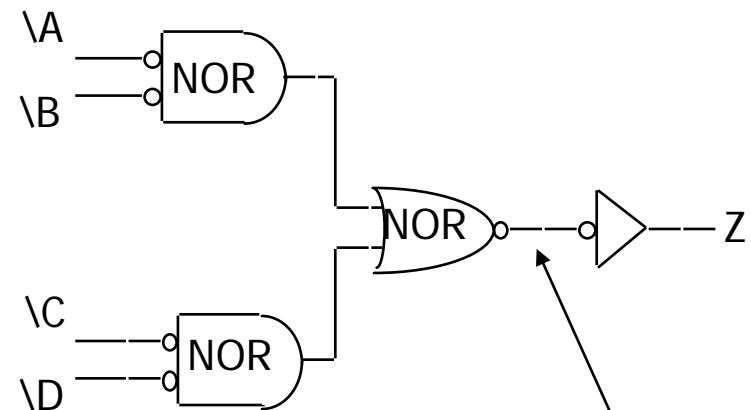
Step 2

conserve "bubbles"

Conversion between forms (cont'd)

- Example: verify equivalence of two forms
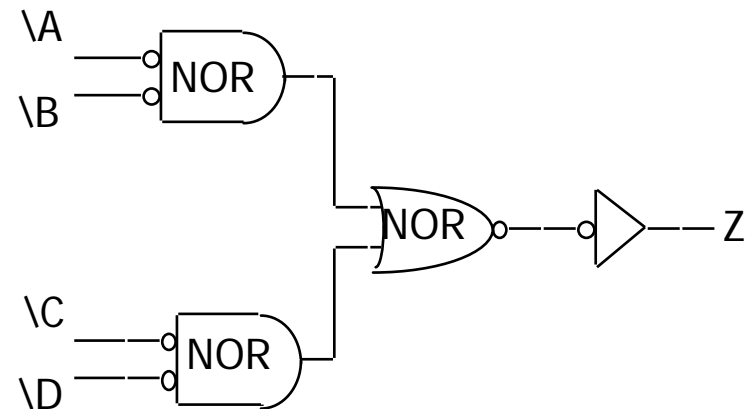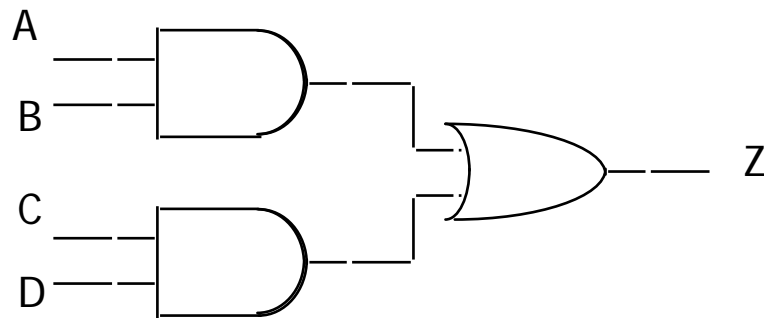
$$Z = \{ \; [\; (A' + B')' + (C' + D')' \;]' \; \}'$$
$$= \{ \quad (A' + B') \; \bullet \; (C' + D') \quad \}'$$
$$= \quad (A' + B')' + (C' + D')'$$
$$= \quad (A \; \bullet \; B) \; + \; (C \; \bullet \; D) \quad ü$$

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)
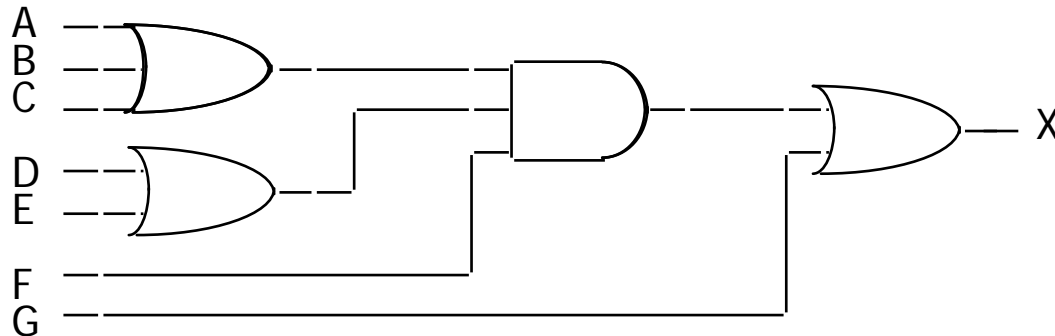
Conversion of multi-level logic to NAND gates

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

original
AND-OR
network

introduction and
conservation of
bubbles

redrawn in terms
of conventional
NAND gates

Level 1   Level 2   Level 3   Level 4
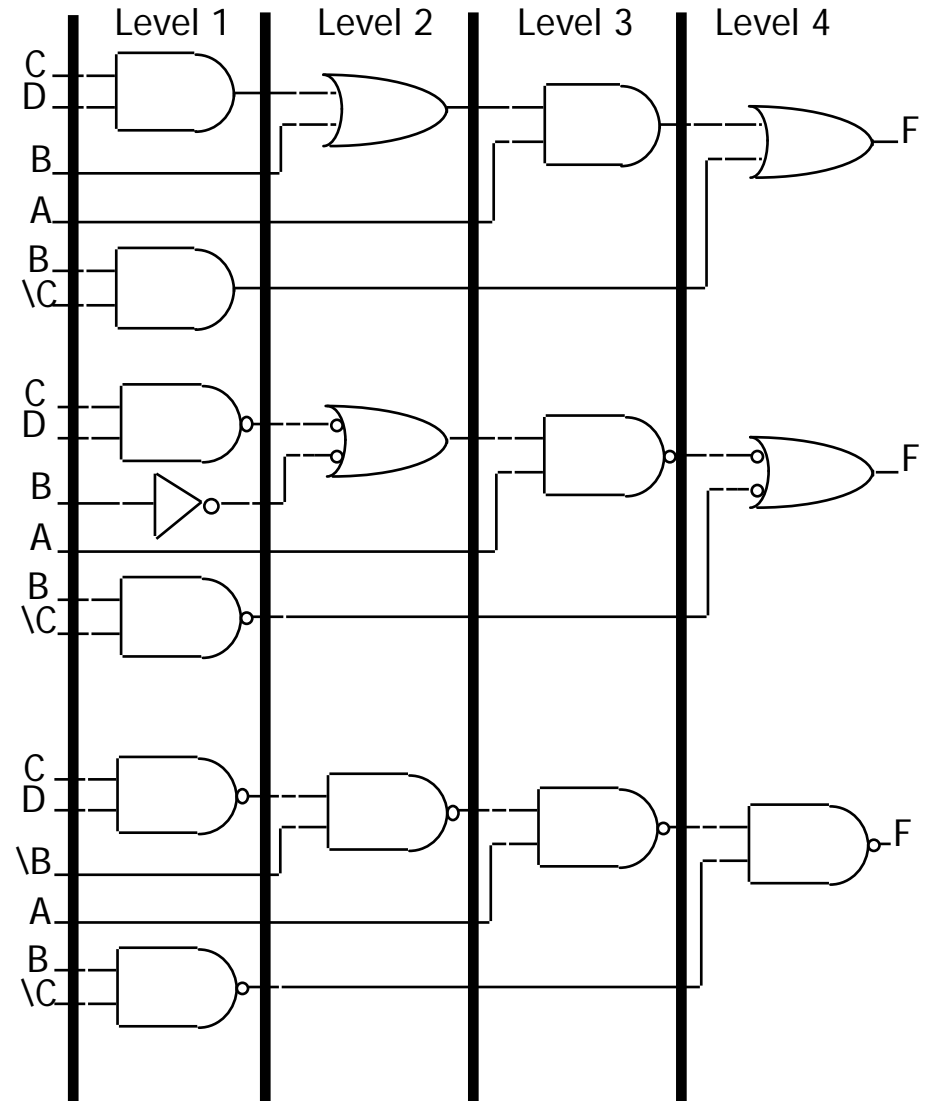
Conversion of multi-level logic to NORs

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

original AND-OR network

introduction and conservation of bubbles

redrawn in terms of conventional NOR gates
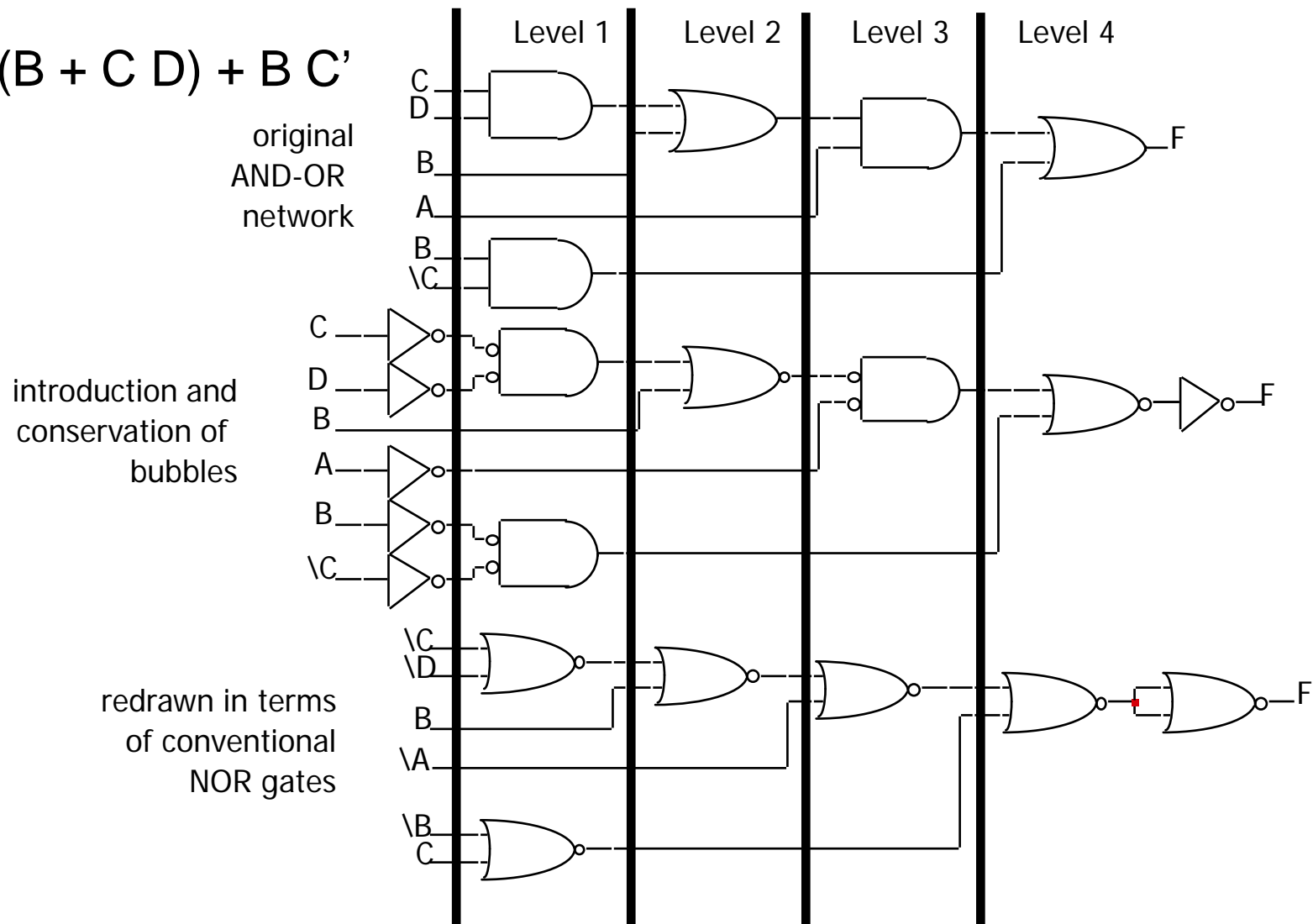
Conversion between forms

- Example
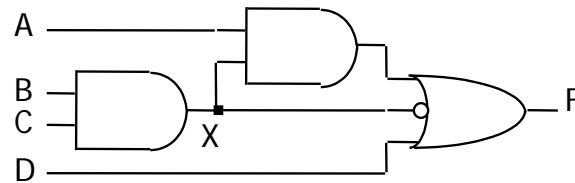
original circuit
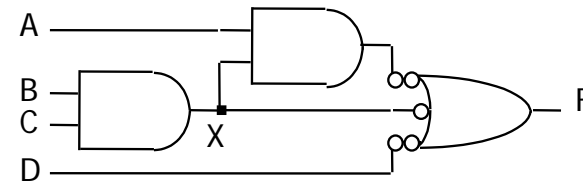
add double bubbles to
invert all inputs of OR gate
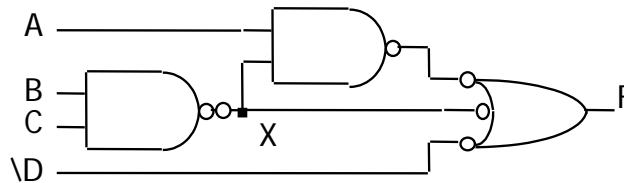
add double bubbles to
invert output of AND gate
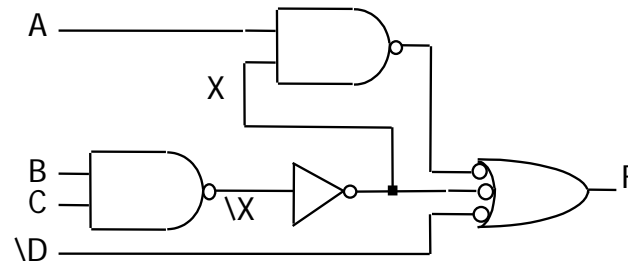
insert inverters to eliminate
double bubbles on a wire

AND-OR-invert gates

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

logical concept

AND   OR   Invert
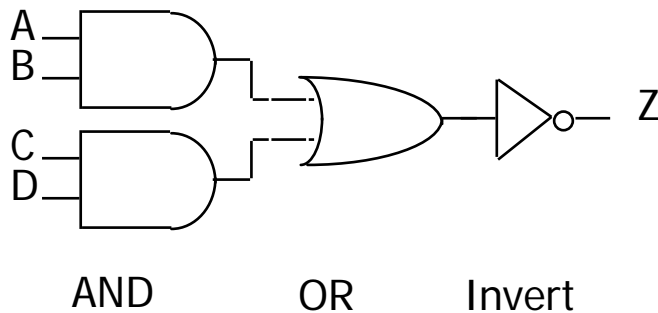
possible implementation

NAND   NAND   Invert
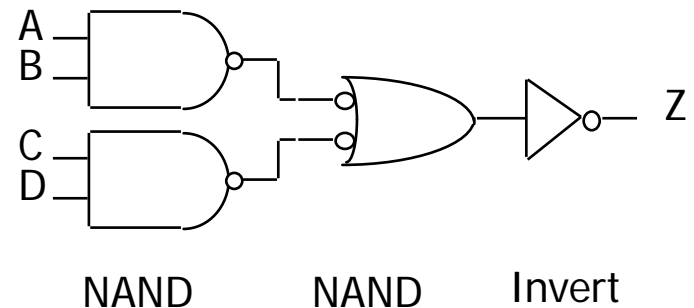
2x2 AOI gate
symbol
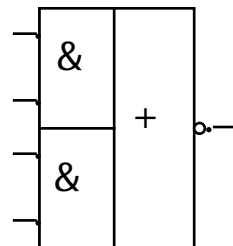
3x2 AOI gate
symbol
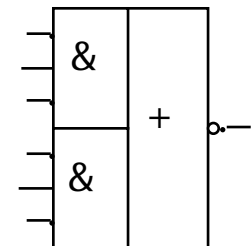
Conversion to AOI forms

- **General procedure to place in AOI form**
  - compute the complement of the function in sum-of-products form
  - by grouping the 0s in the Karnaugh map
- **Example: XOR implementation**
  - A xor B = A' B + A B'
  - AOI form:
    - F = (A' B' + A B)'
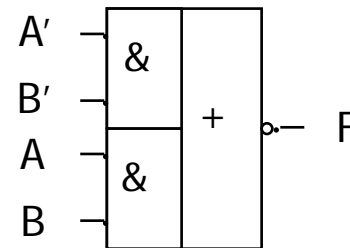
Examples of using AOI gates

- Example:
  - F = A B + A C' + B C'
  - 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 = (A0 B0 + A0' B0')(A1 B1 + A1' B1')(A2 B2 + A2' B2')(A3 B3 + A3' B3')

    each implemented in a single 2x2 AOI gate

Examples of using AOI gates (cont'd)

- Example: AOI implementation of 4-bit equality function

high if A0 ≠ B0
low if A0 = B0

conservation of bubbles

if all inputs are low
then Ai = Bi, i=0,…,3
output Z is high
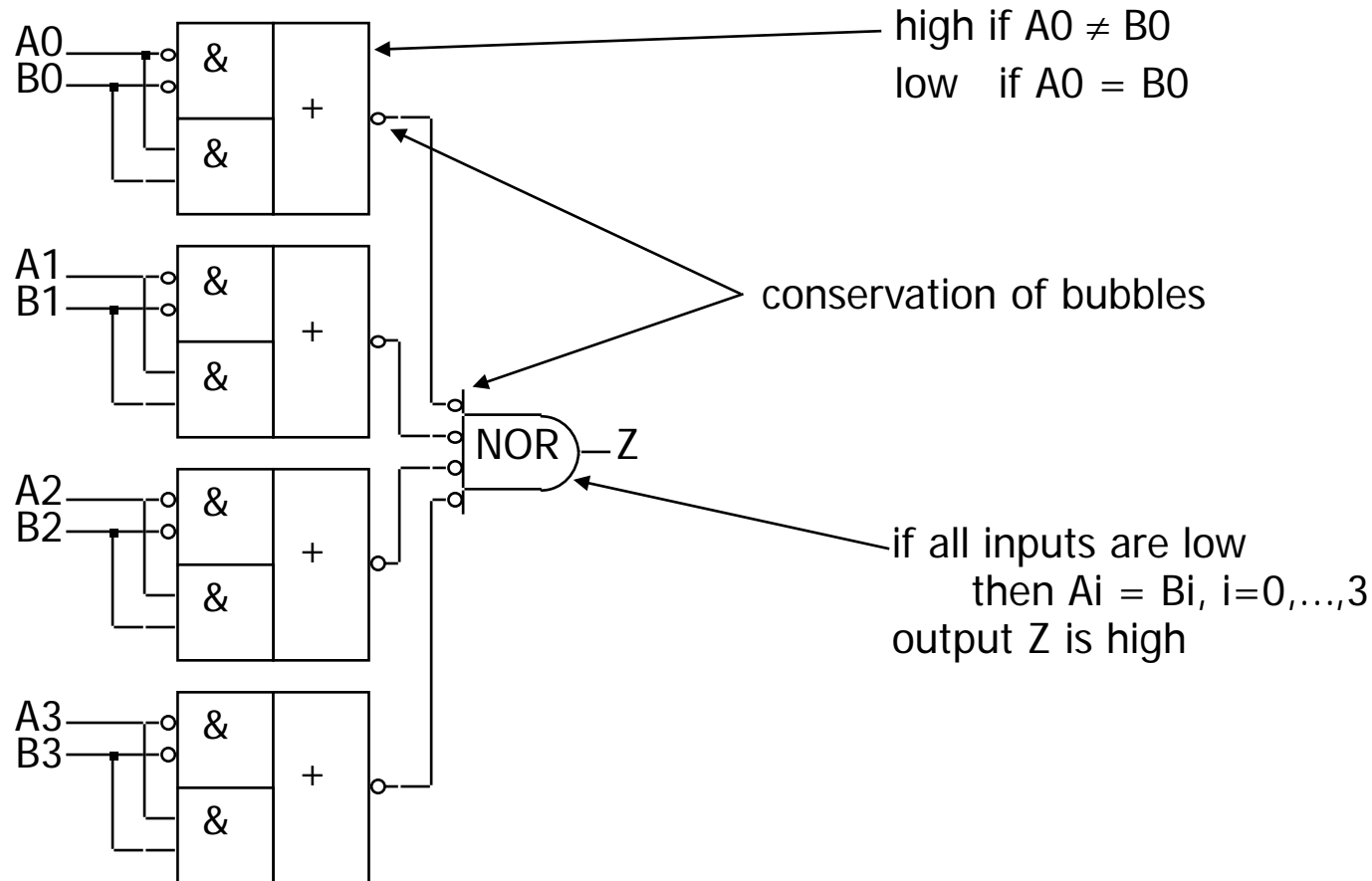
Summary for multi-level logic

- **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' • A = 0
  - delays matter
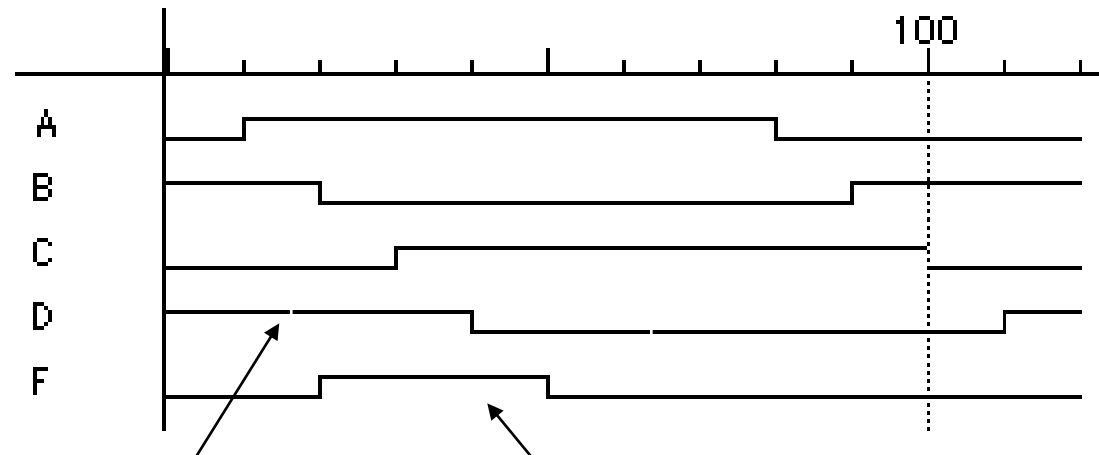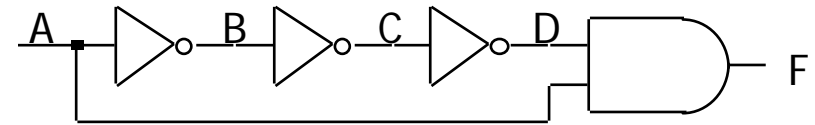
D remains high for three gate delays after A changes from low to high

F is not always 0 pulse 3 gate-delays wide

Oscillatory behavior

- **Another pulse shaping circuit**
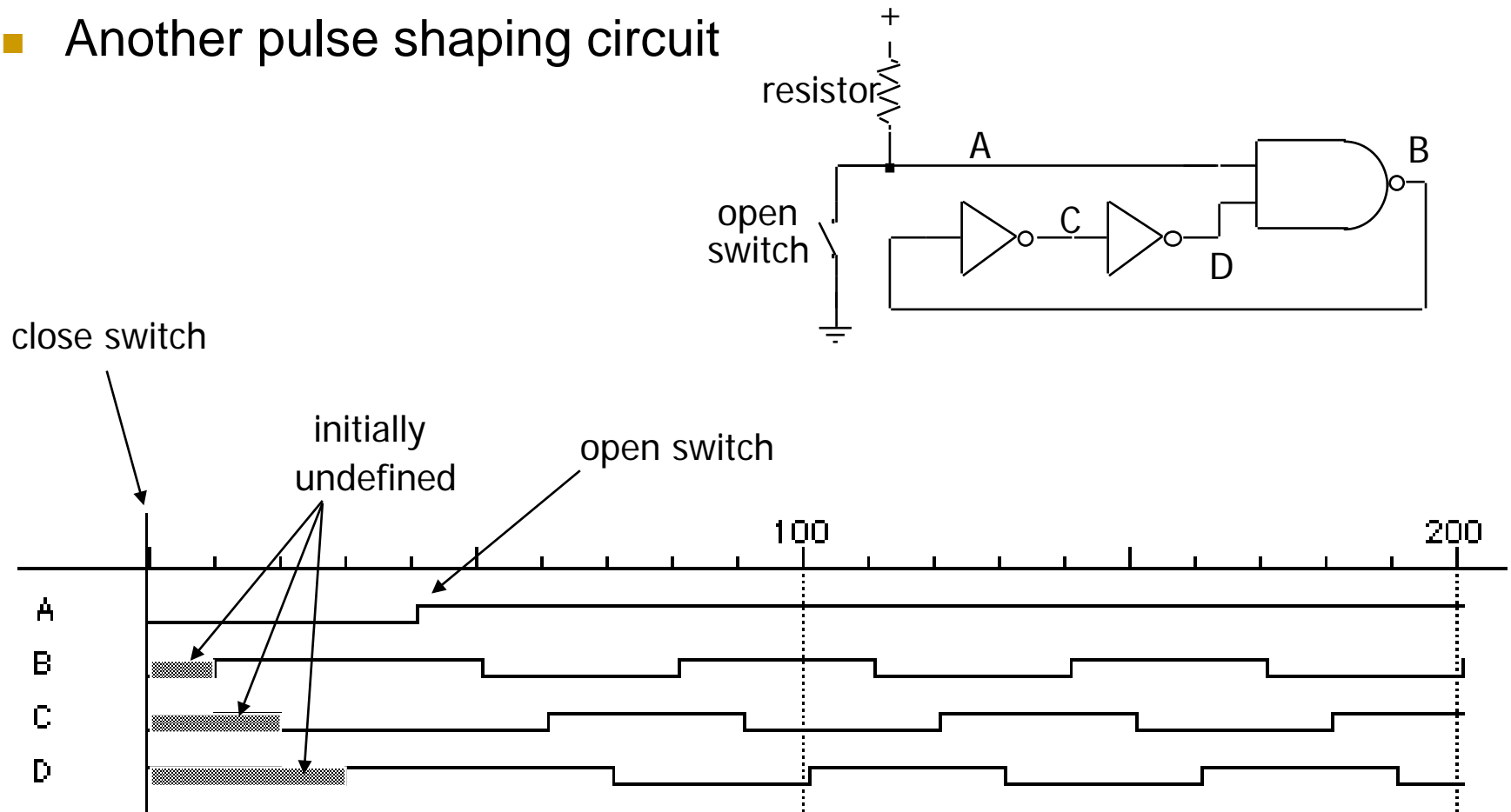
Hardware description languages

- 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"

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

  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

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

2-bit vector

initial block executed only once at start of simulation

print to a console

directive to stop simulation

Complete simulation

- Instantiate stimulus component and device to test in a schematic
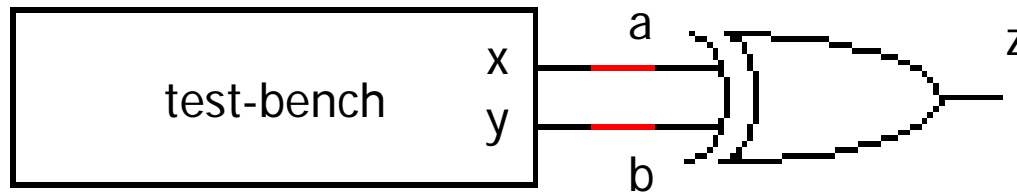
Comparator example

```
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

```verilog
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