ELPL Embedded Low-Power Laboratory

Digital Logic Design

4190.201

2014 Spring Semester

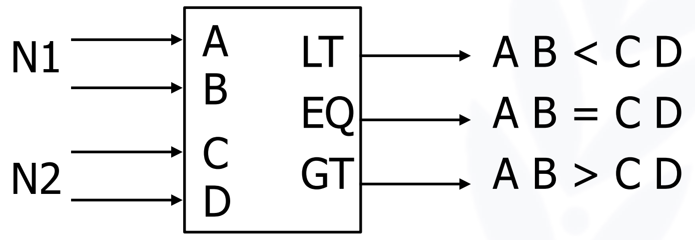
4. Working with combinational logic

Naehyuck Chang
Dept. of CSE
Seoul National University
naehyuck@snu.ac.kr

What to cover

- 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



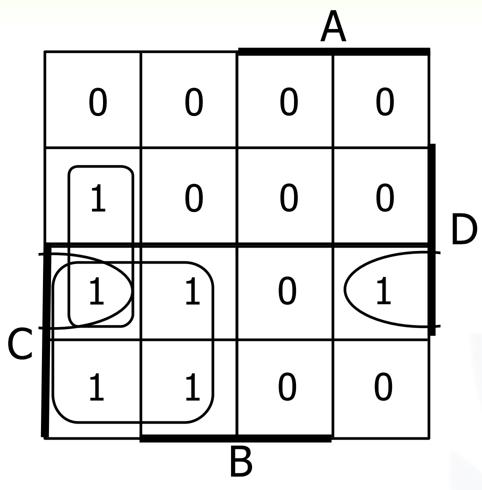
Block diagram
and
truth table

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

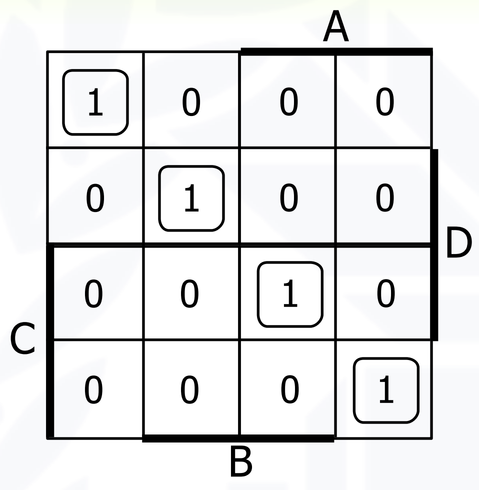
We'll need a 4-variable Karnaugh map
for each of the 3 output functions

Embedded Low-Power Laboratory

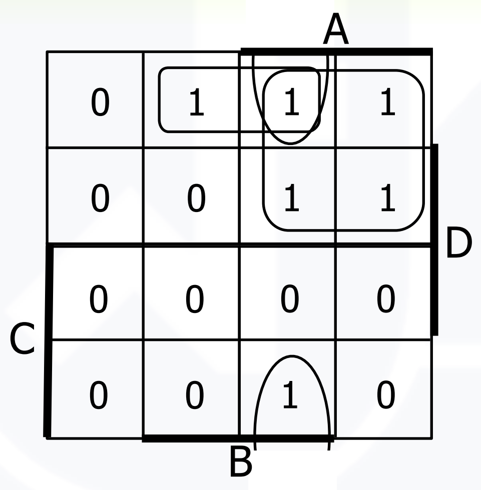
Design example: two-bit comparator



K-map for LT



K-map for EQ



K-map for GT

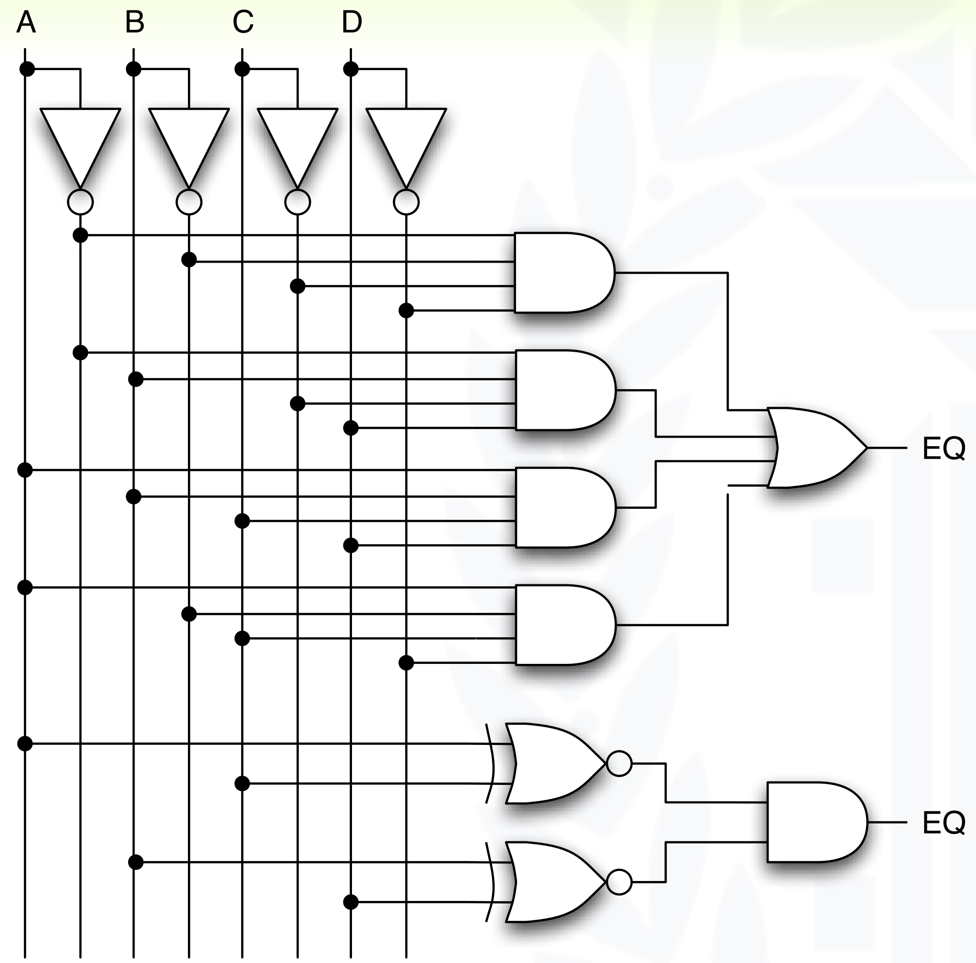
LT = A' B' D + A' C + B' C D

EQ = A' B' C' D' + A' B C' D + A B C D + A B' C D′ = (A xnor C) • (B xnor D)

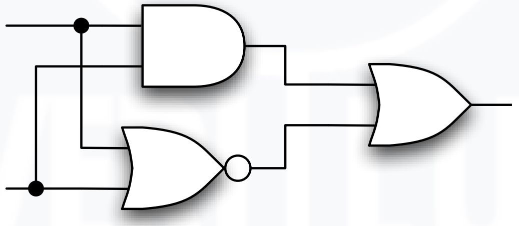
GT = B C' D' + A C' + A B D'

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

Design example: two-bit comparator



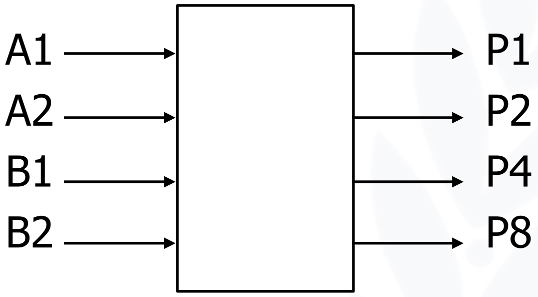
Two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Embedded Low-Power Laboratory

Design example: 2x2-bit multiplier



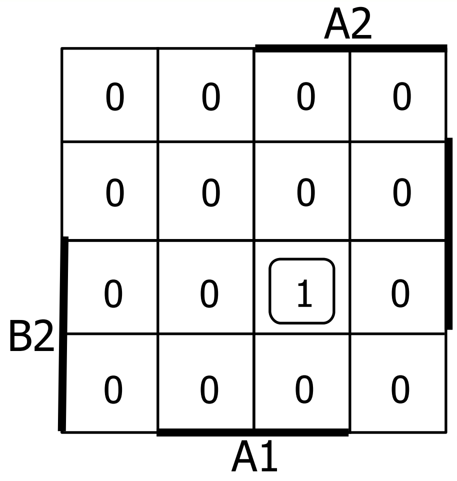
Block diagram
and
truth table

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

K-map for P8

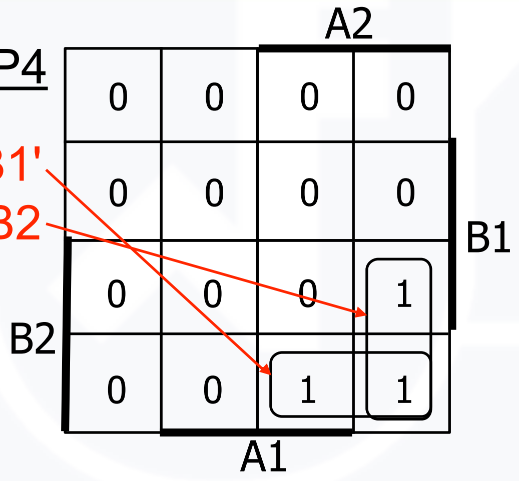


0	0	0	0
0	0	0	0
0	0	1	0
0	0	0	0

P8 = A2A1B2B1

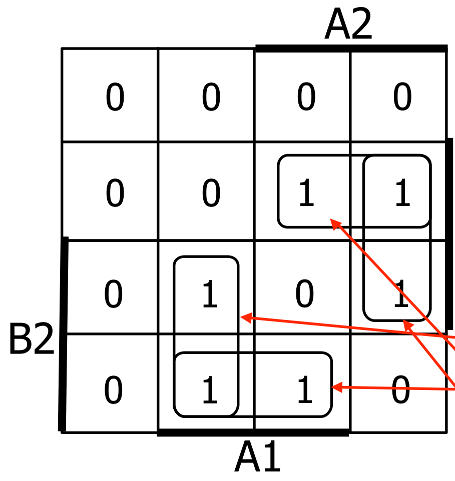
K-map for P4

P4 = A2B2B1'
+ A2A1'B2



0	0	0	0
0	0	0	0
0	0	0	1
0	0	1	1

K-map for P2

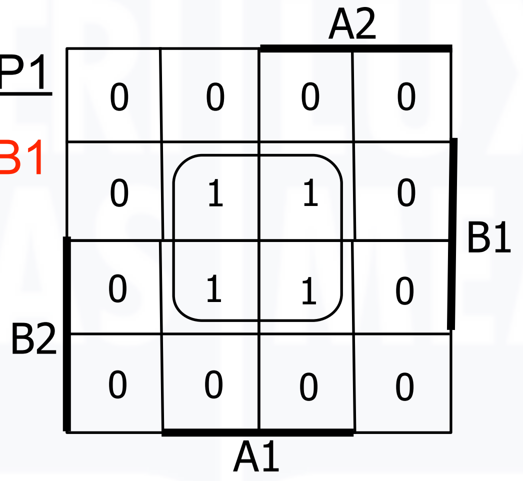


0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0

P2 = A2'A1B2
+ A1B2B1'
+ A2B2'B1
+ A2A1'B1

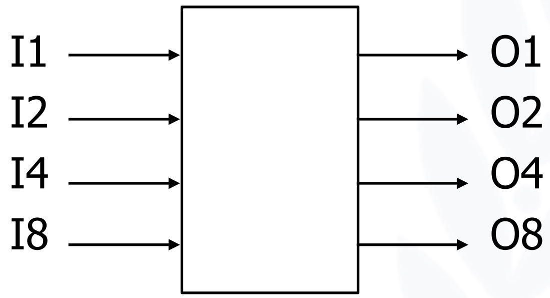
K-map for P1

P1 = A1B1



0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

Design example: BCD increment by 1



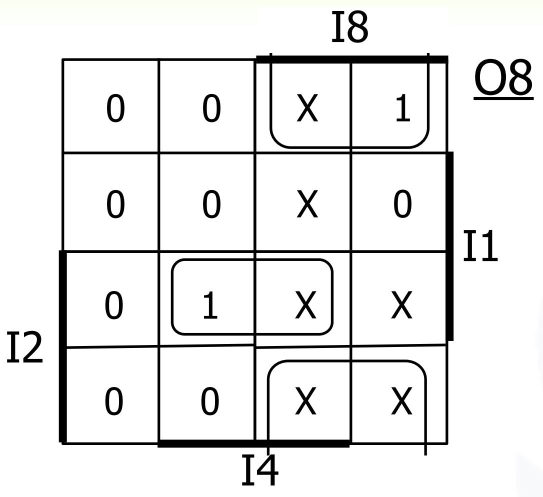
Block diagram
and
truth table

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
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
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

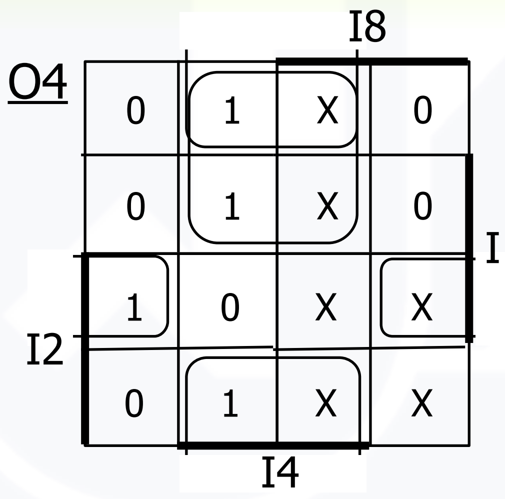
4-variable K-map for each of
the 4 output functions

Embedded Low-Power Laboratory

Design example: BCD increment by 1



0	0	X	1
0	0	X	0
0	1	X	X
0	0	X	X



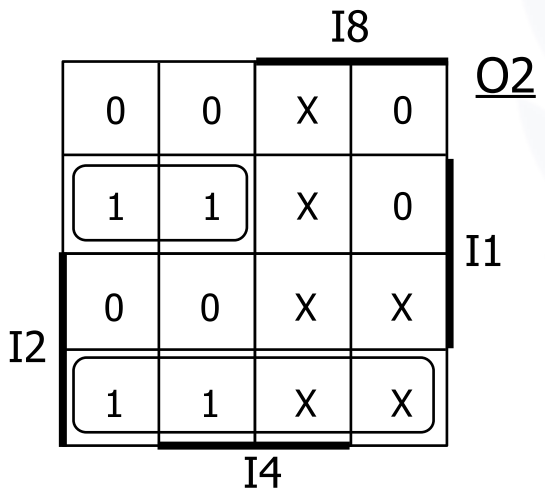
0	1	X	0
0	1	X	0
1	0	X	X
0	1	X	X

O8 = I4 I2 I1 + I8 I1'

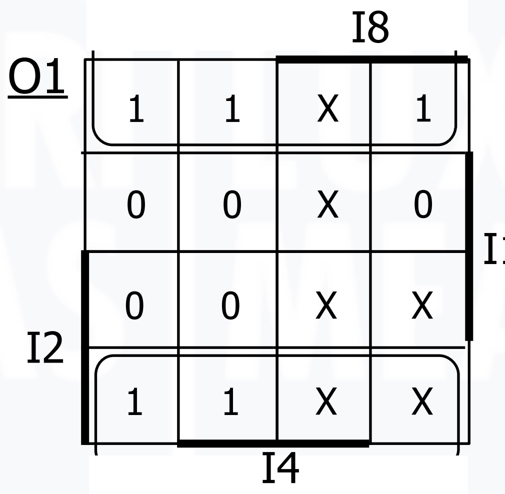
O4 = I4 I2' + I4 I1' + I4′ I2 I1

O2 = I8′ I2′ I1 + I2 I1'

O1 = I1'



0	0	X	0
1	1	X	0
0	0	X	X
1	1	X	X



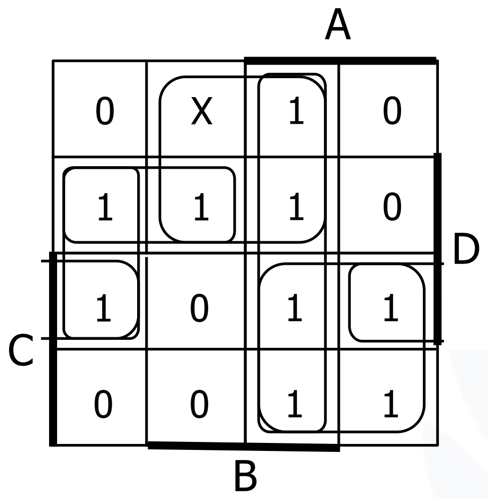
1	1	X	1
0	0	X	0
0	0	X	X
1	1	X	X

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
- Redundant prime implicant
- 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)

Embedded Low-Power Laboratory

Examples to illustrate terms



6 prime implicants:

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

essential

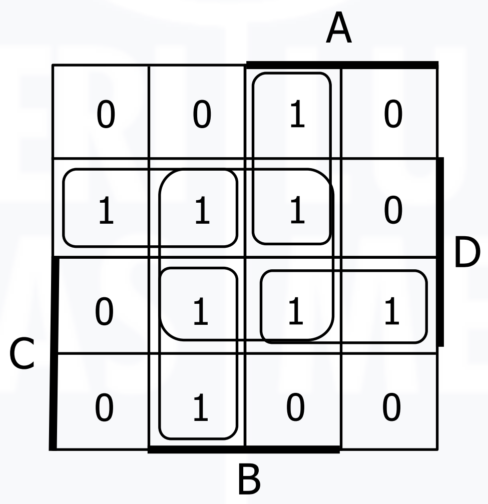
Minimum cover: AC + BC' + A'B'D

5 prime implicants:

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

essential

Minimum cover: 4 essential implicants

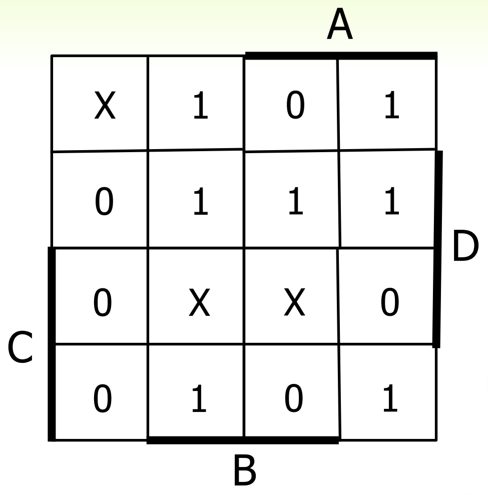


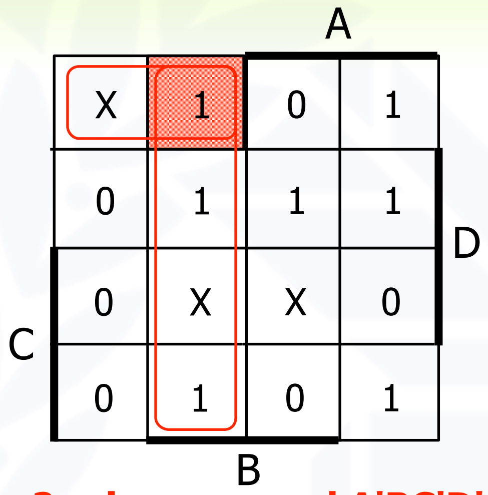
Algorithm for two-level simplification

- 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

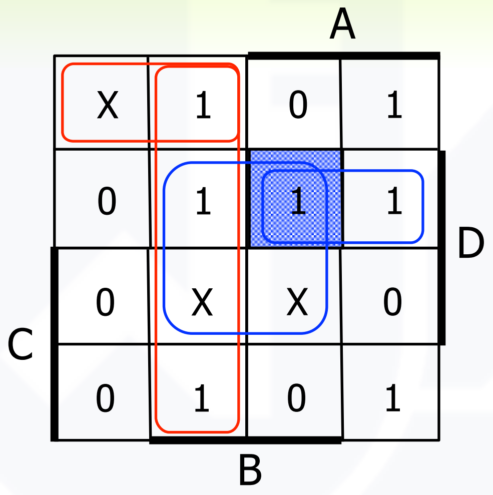
ELPL Embedded Low-Power Laboratory

Algorithm for two-level simplification (example)

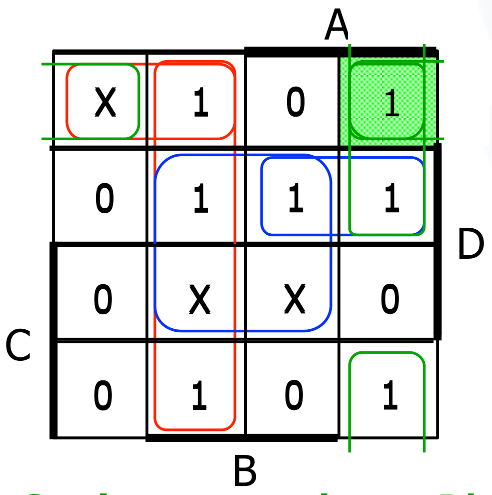




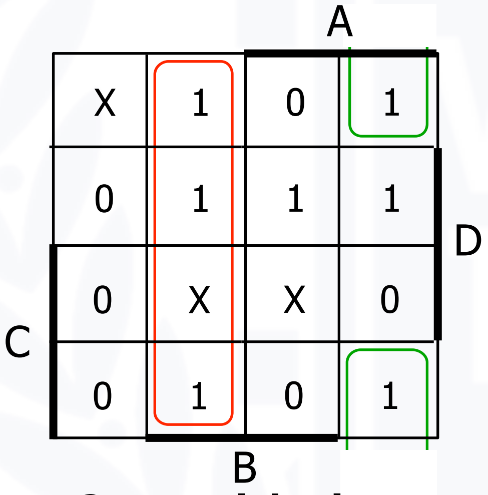
2 primes around A'BC'D'



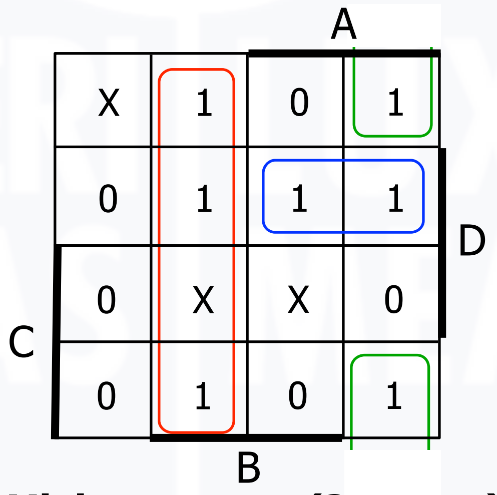
2 primes around ABC'D



3 primes around AB'C'D'



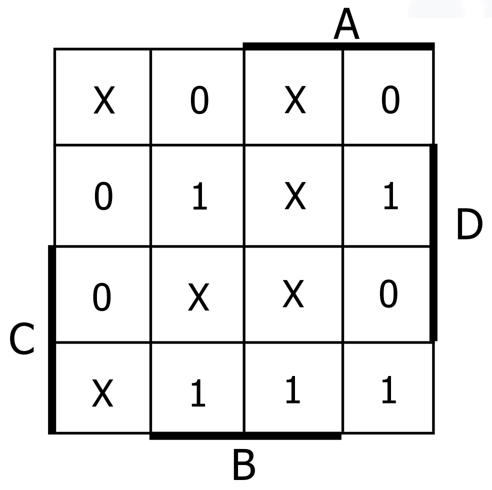
2 essential primes



Minimum cover (3 primes)

Activity

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



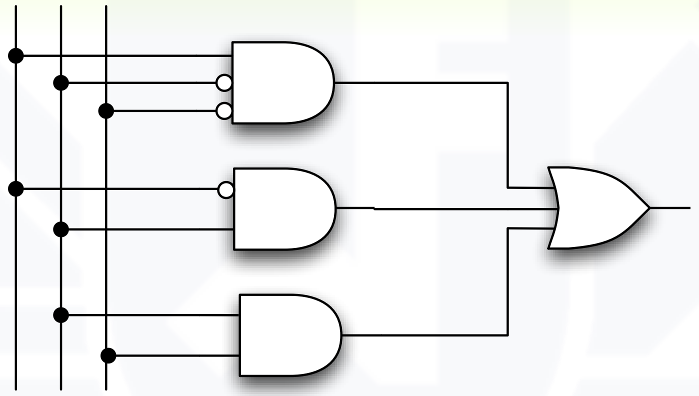
X	0	X	0
0	1	X	1
0	X	X	0
X	1	1	1

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

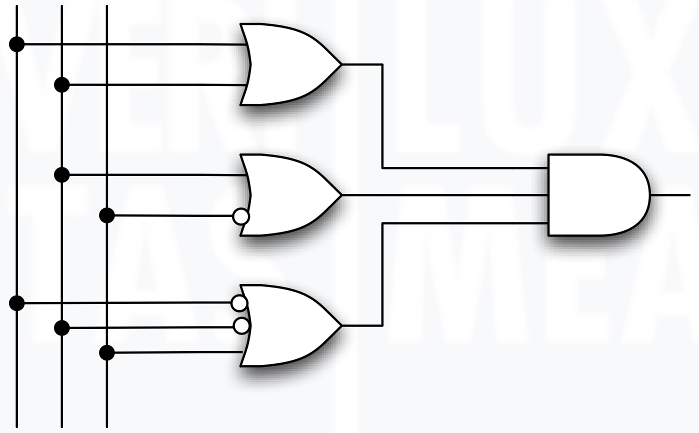
Embedded Low-Power Laboratory

Implementations of two-level logic

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

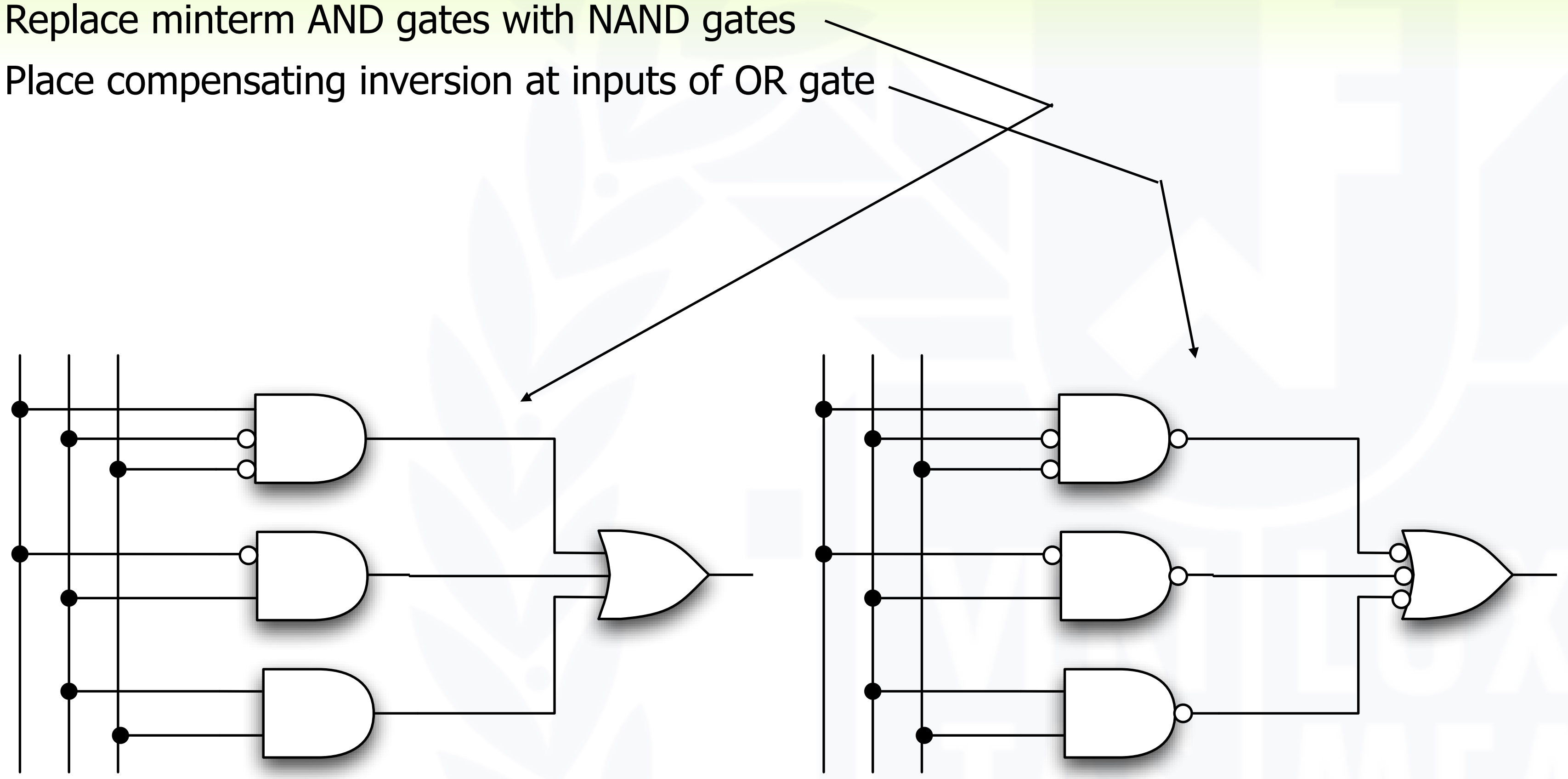


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



Two-level logic using NAND gates

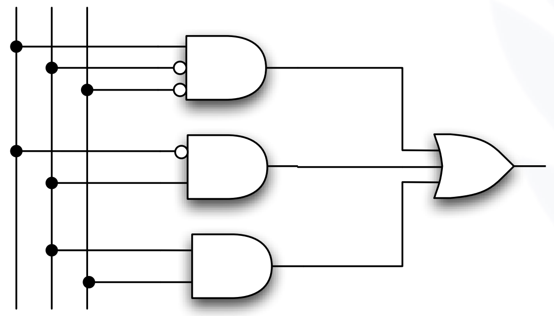
- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate

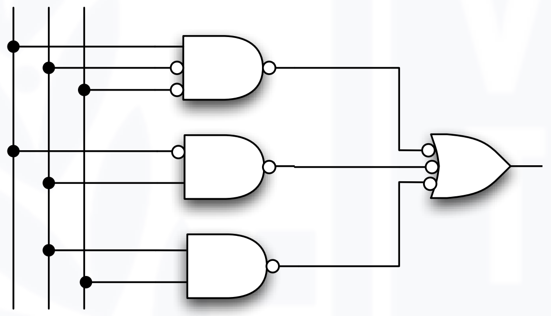


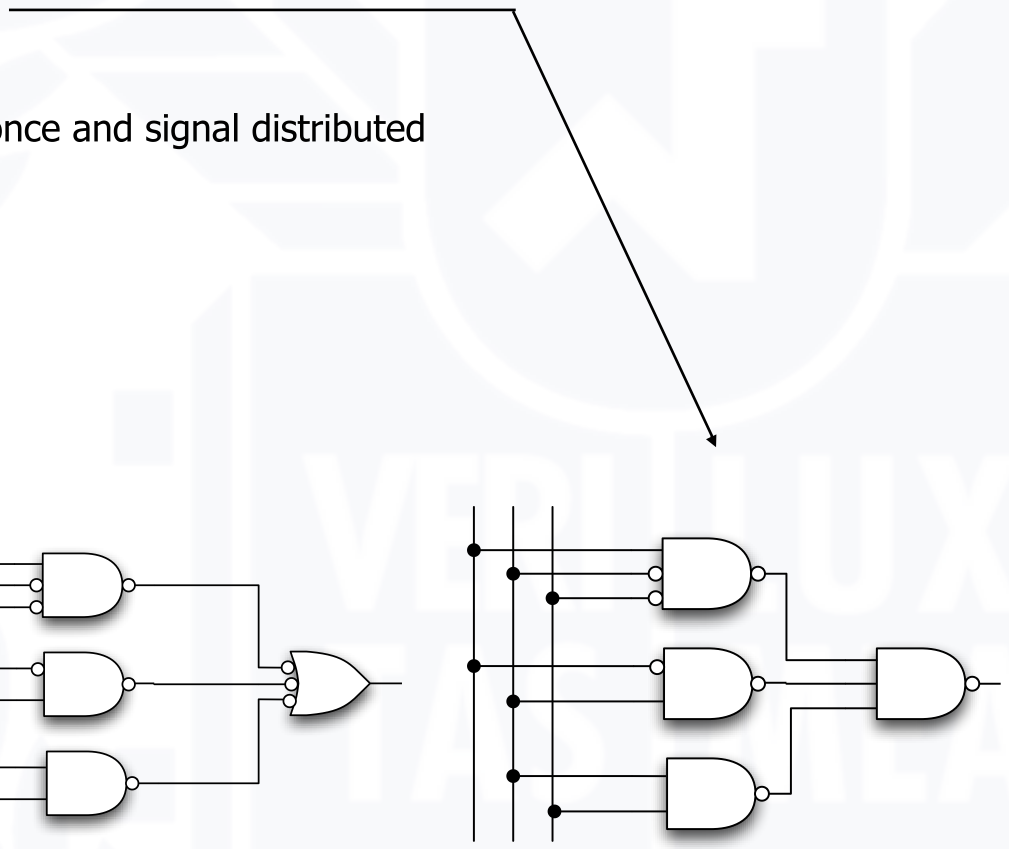
Embedded Low-Power Laboratory

Two-level logic using NAND gates

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



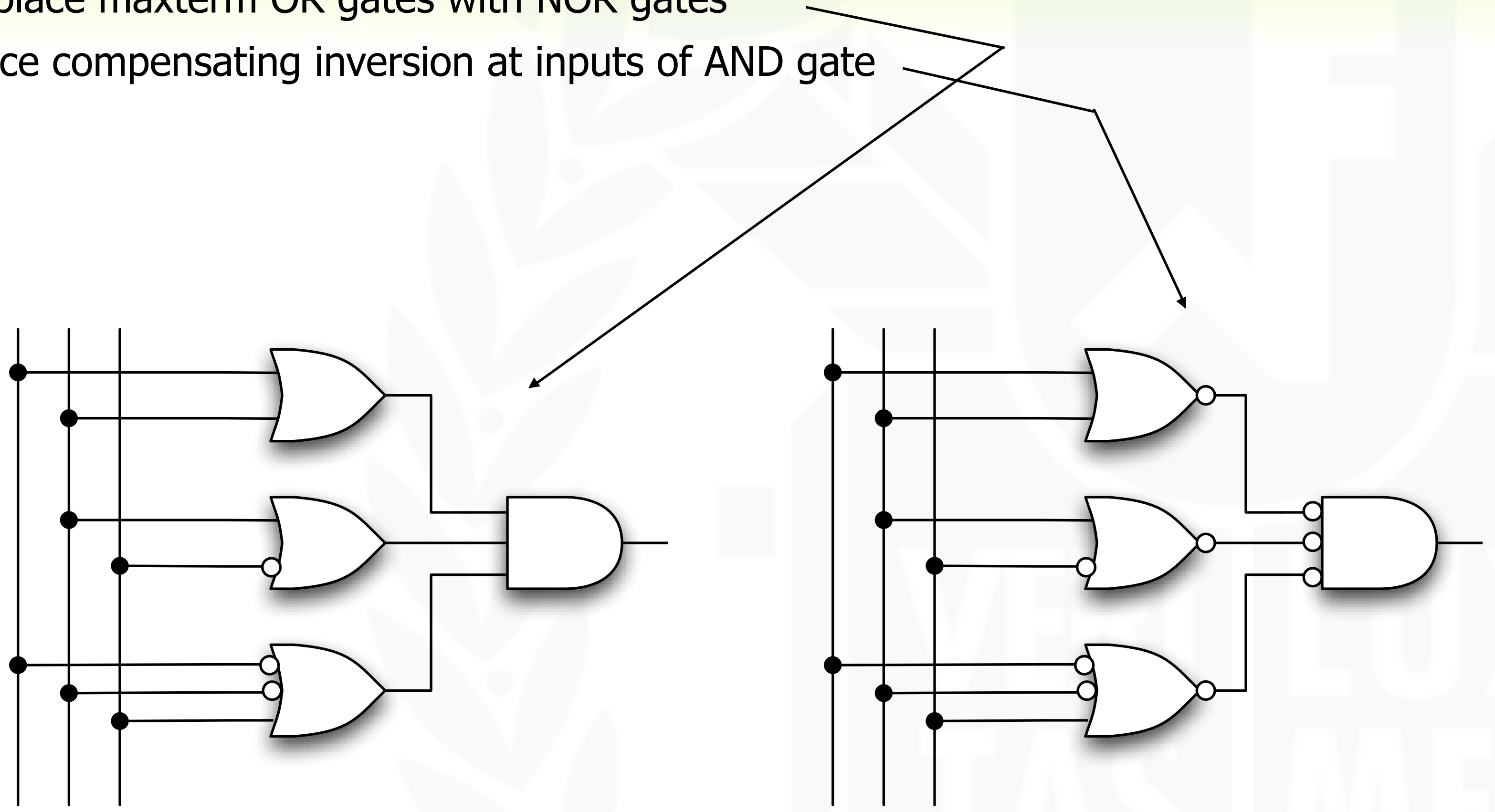




ELPL Embedded Low-Power Laboratory

Two-level logic using NOR gates

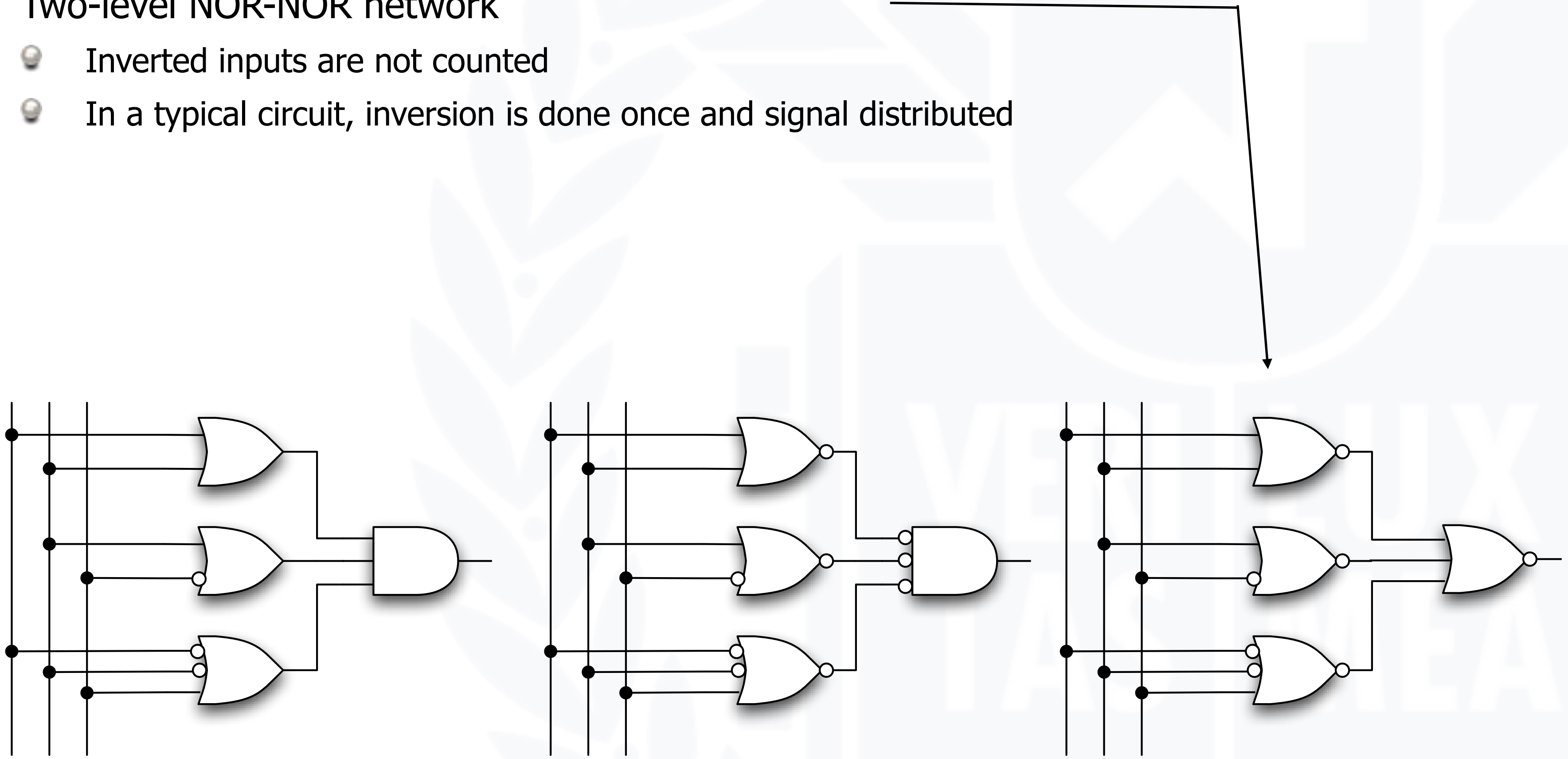
- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate



Embedded Low-Power Laboratory

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

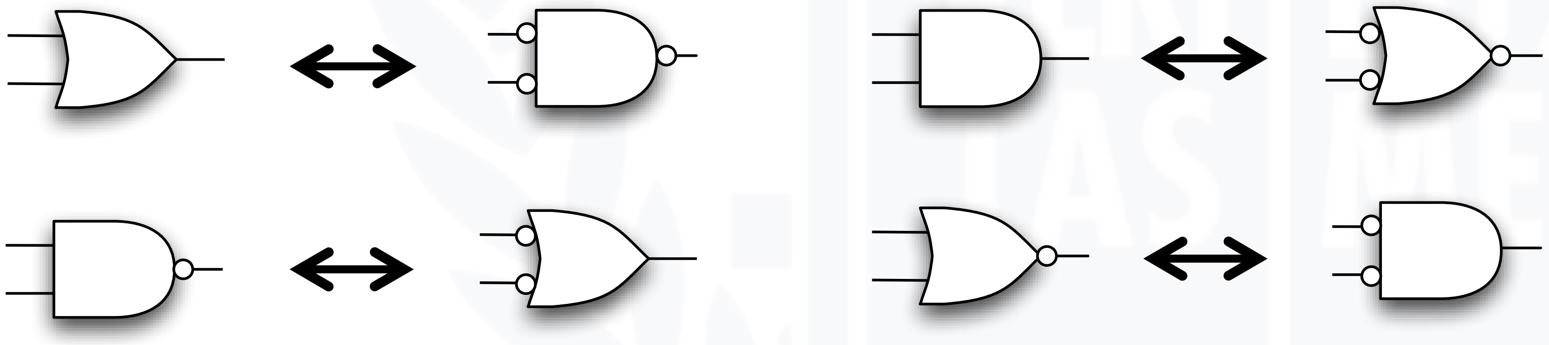
- AND gate with inverted inputs is a NOR gate
  - De Morgan's: A' • B' = (A + B)'
- Two-level NOR-NOR network
  - Inverted inputs are not counted
  - In a typical circuit, inversion is done once and signal distributed



Embedded Low-Power Laboratory

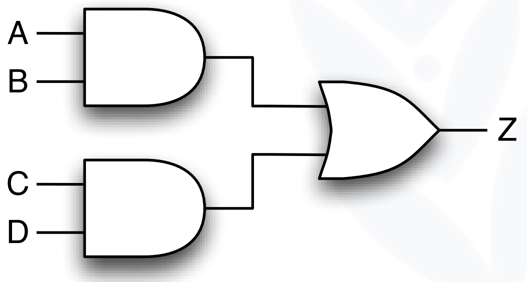
Two-level logic using NAND and NOR gates

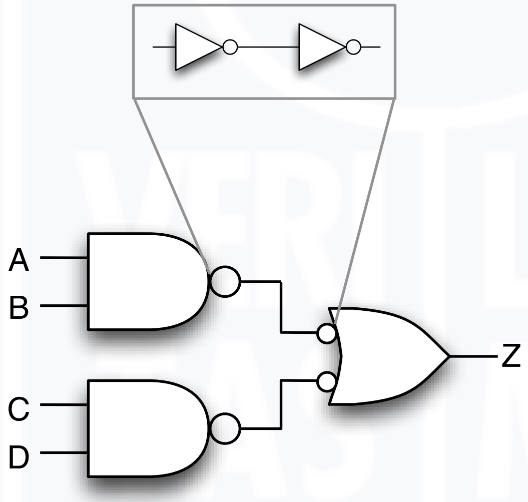
- NAND-NAND and NOR-NOR networks
  - de Morgan's law: $(A + B)' = A' \bullet B'$ $(A \bullet B)' = A' + B'$
  - Written differently: $A + B = (A' \bullet B')'$ $(A \bullet B) = (A' + B')'$
- In other words —
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented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



Conversion between forms

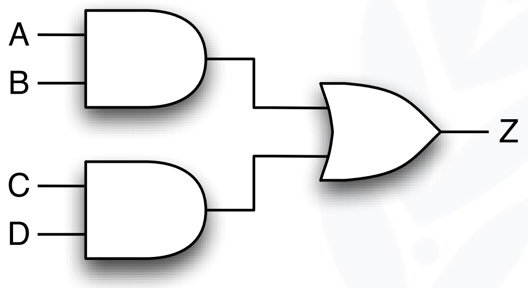
- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - Introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be matched by a corresponding "bubble"
  - Conservation of inversions
  - Do not alter logic function
- Example: AND/OR to NAND/NAND

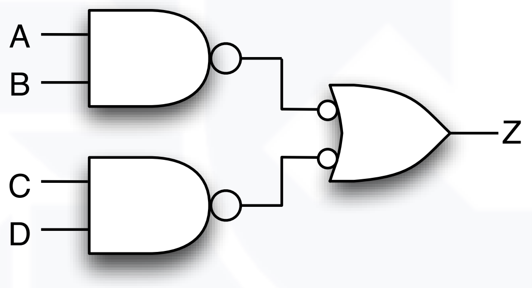




Conversion between forms (cont'd)

- Example: Verify equivalence of two forms

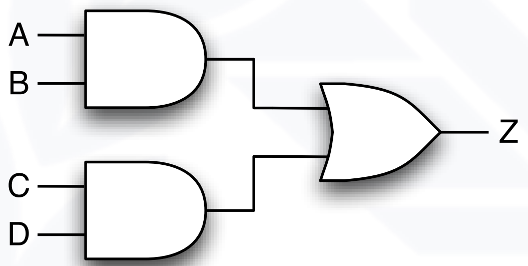


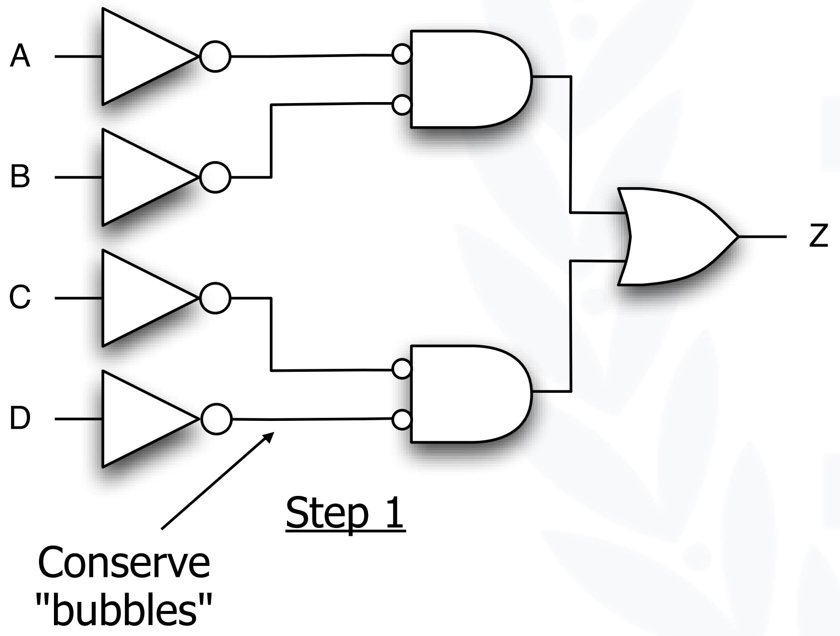


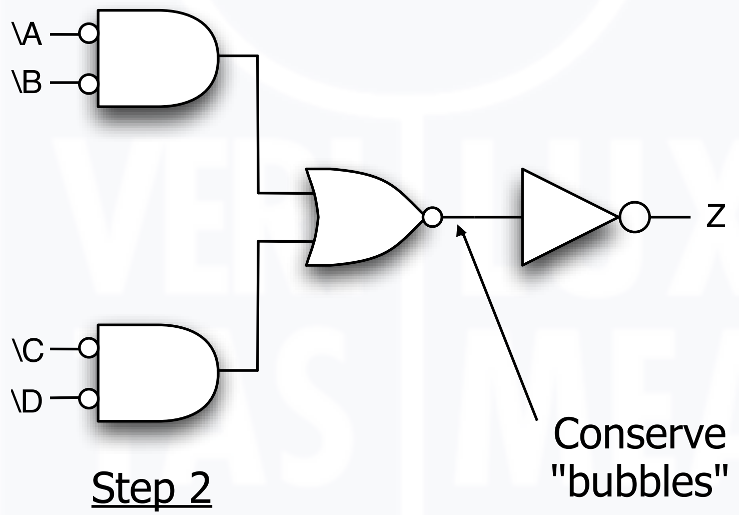
$$Z = [\,(A \bullet B)' \bullet (C \bullet D)'\,]'$$
$$= [\,(A' + B') \bullet (C' + D')\,]'$$
$$= [\,(A' + B')' + (C' + D')'\,]$$
$$= (A \bullet B) + (C \bullet D)$$

Conversion between forms (cont'd)

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

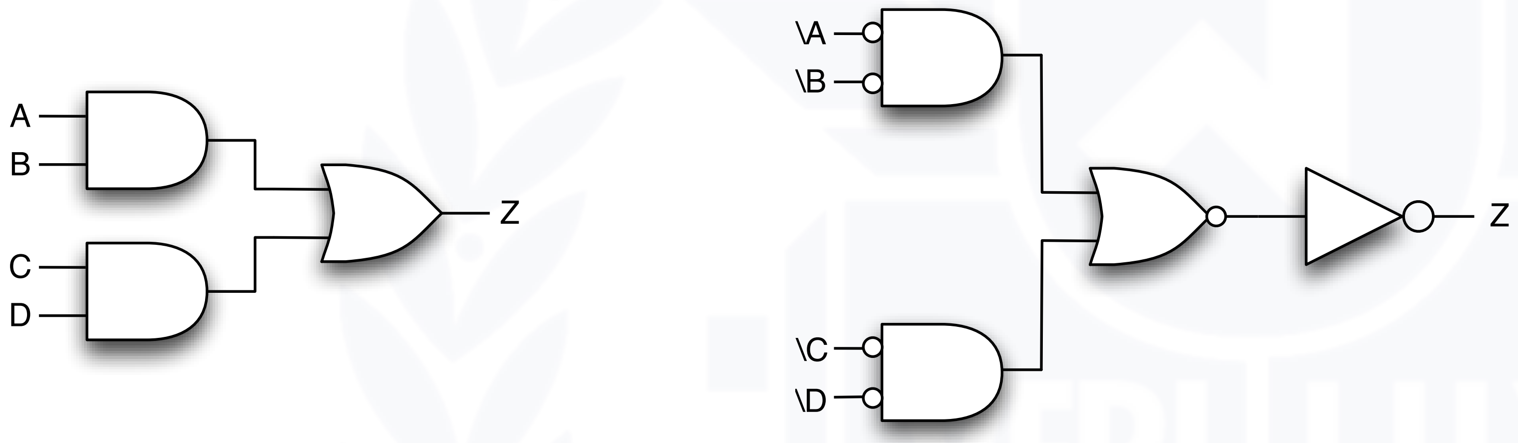






Conversion between forms

- Example: Verify equivalence of two forms

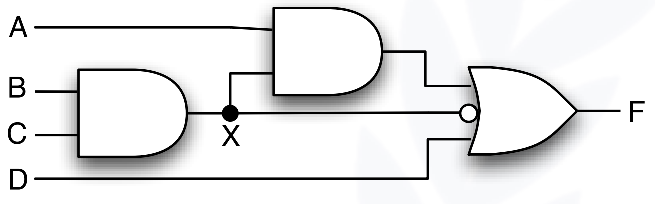


$$
\begin{aligned}
Z &= \{\ [\ (A' + B')' + (C' + D')'\ ]'\ \}' \\
&= \{\ (A' + B') \bullet (C' + D')\ \}' \\
&= (A' + B')' + (C' + D')' \\
&= (A \bullet B) + (C \bullet D)
\end{aligned}
$$

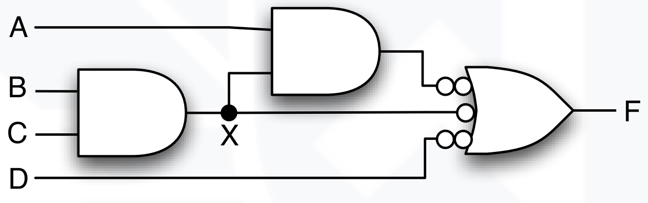
Embedded Low-Power Laboratory

Conversion between forms

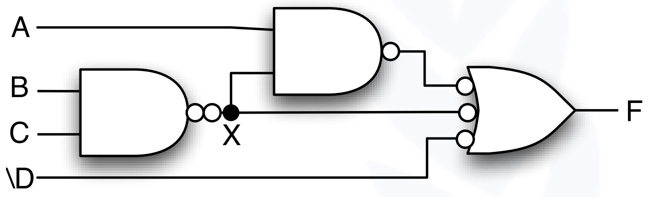
- Example



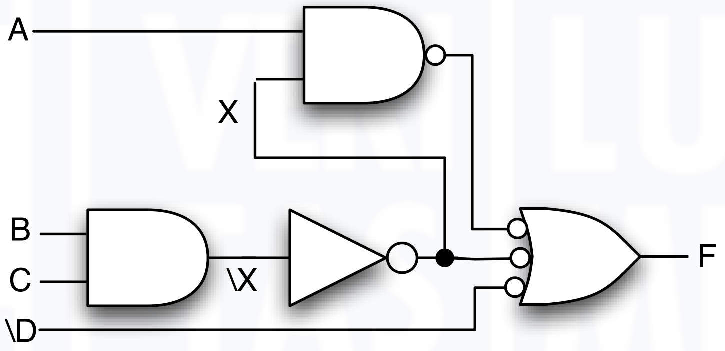
Original circuit



Add double bubbles to
invert all inputs of OR gate



Add double bubbles to
invert output of AND gate

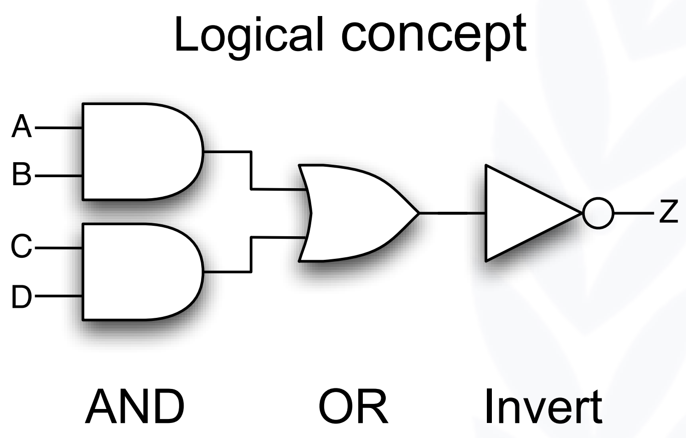


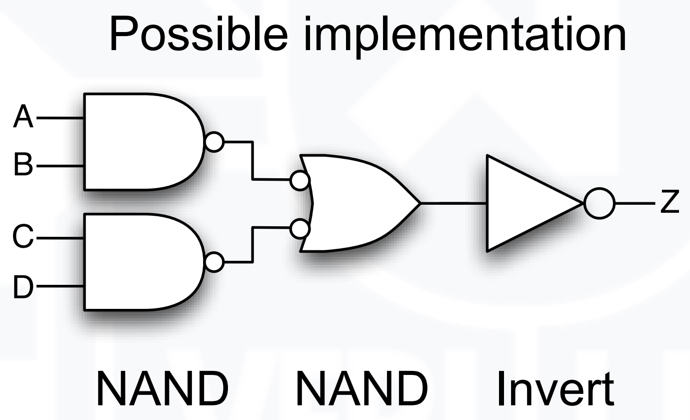
Insert inverters to eliminate
double bubbles on a wire

Embedded Low-Power Laboratory

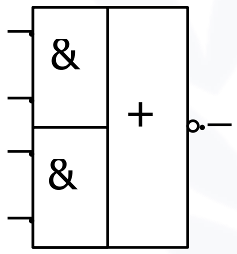
AND-OR-invert gates

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

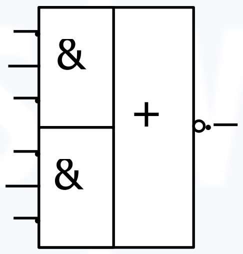




2x2 AOI gate symbol

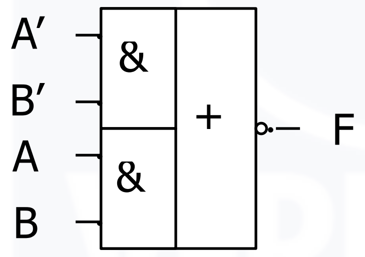


3x2 AOI gate symbol



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)′



Summary for multi-level logic

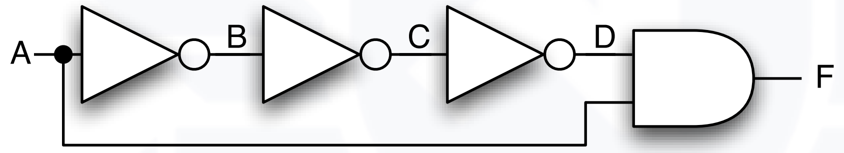
- Advantages
  - Circuits may be smaller
  - Gates have smaller fan-in
  - Circuits may be faster
- Disadvantages
  - More difficult to design
  - Tools for optimization are not as good as for two-level
  - Analysis is more complex

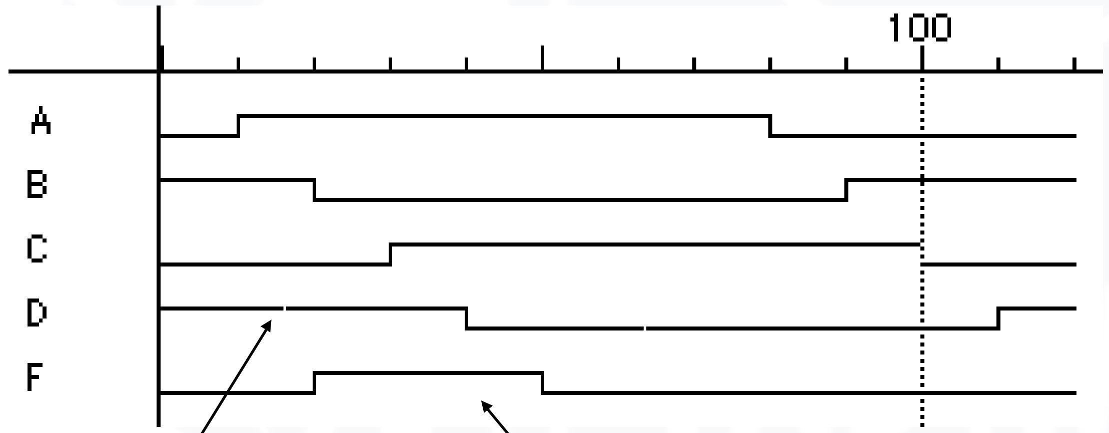
Time behavior of combinational networks

- 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





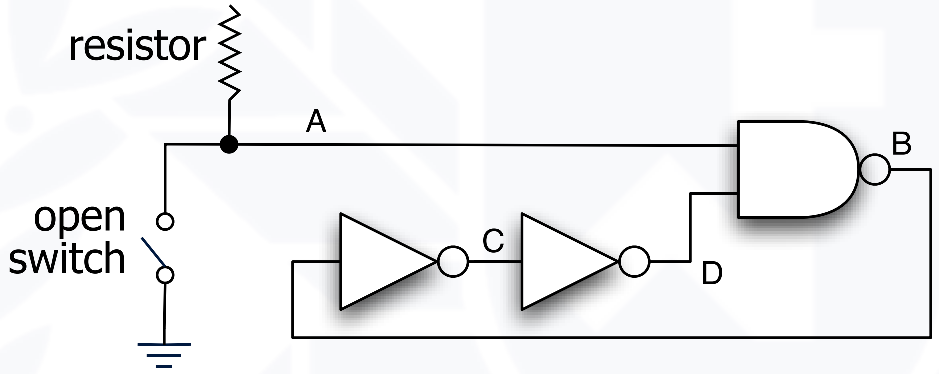
D remains high for three gate delays after A changes from low to high

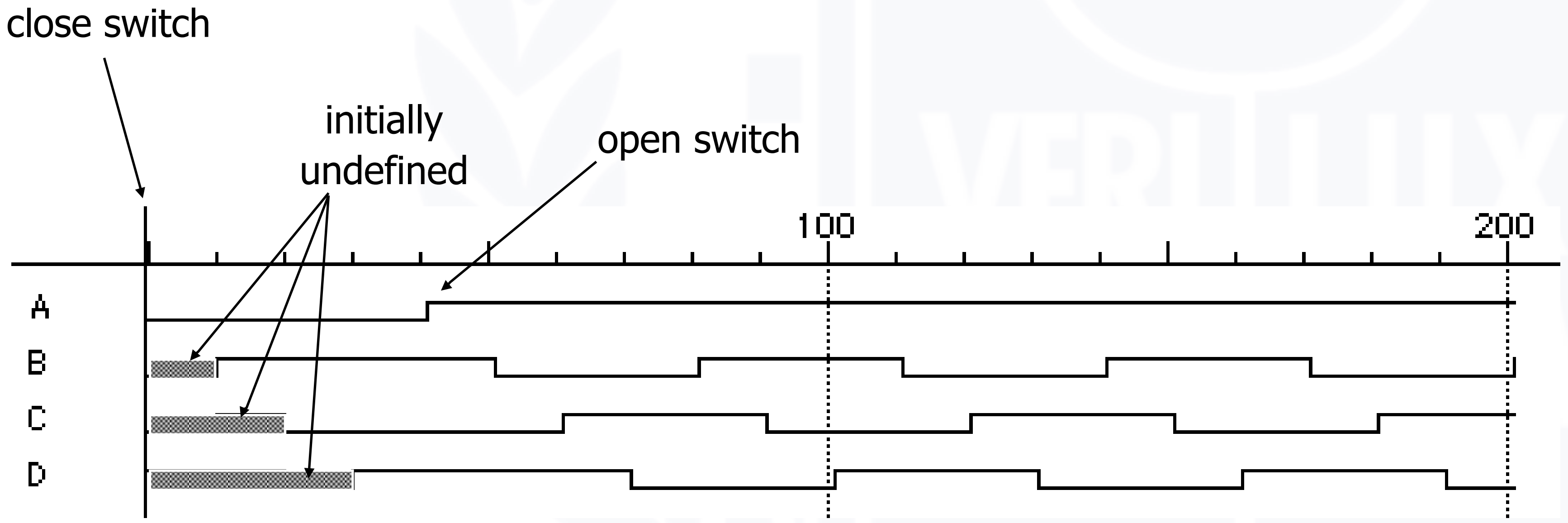
F is not always 0 pulse 3 gate-delays wide

Embedded Low-Power Laboratory

Oscillatory behavior

- Another pulse shaping circuit





Hazards and glitches

- Glitch
  - Unwanted pulse of a combinational logic network
- Hazard
  - A circuit has a potential to generate glitch
  - Intrinsic characteristic
    - A circuit with a hazard may or may not generate glitch depending on the input pattern
- Static hazard
  - Static 0-hazard: momentarily 1 while the output is 0
  - Static 1-hazard: momentarily 0 while the output is 1
- Dynamic hazard
  - Generate glitch more than once for a single transition 0 to 1 or 1 to 0
- Hazard-free circuit generation
  - Can avoid hazard if there is only single input change
  - Hazard caused by simultaneous multiple input changes is unavoidable

Hazard-free circuit

- No hazard
  - When the initial and final inputs are covered by the same prime implicant
- Hazard
  - When the input change spans prime implicants
- Generalized static hazard-free circuits
  - Add a redundant prime implicants so that all the single-input transitions are covered by one prime implicant
- Dynamic hazard-free circuits
  - Extension of the static hazard-free method
  - Beyond the scope of this class

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

PALASM

```
TITLE      <Design title>
     PATTERN    <Identification such as file name>
     REVISION   <Version or other ID>
     AUTHOR     <Name of designer>
     COMPANY    <Organization name>
     DATE       <Relevant date>

     CHIP  <Description>  <Device name>
     ; <Pin numbers, eg 1   2   3   4  5  6  7  8>
       <pin names,   eg Clk Clr Pre I1 I2 I3 I4 GND>
     ; <Pin numbers, eg 9   10  11  12  13  14  15  16>
       <pin names,   eg NC  NC  Q1  Q2  Q3  Q4  NC  Vcc>

     STRING  <Name>  '<Characters to substitute>'
             <more string definitions>
     EQUATIONS
       <combinatorial equations of the form
         OutName = Name1 Op1 Name2 .... OpN NameM>

       <registered equations of the form
         OutName := Name1 Op1 Name2 .... OpN NameM>
```

PALASM

- Operators
  - / NOT or active-low
  - * AND
  - + OR
  - :+: XOR
  - = Combinational output
  - *= Latched output
  - := Registered output
- Simulation feature

ABEL

- Extended version of PALASM for PLD design
- Logic is expressed by
  - Equations
  - Truth Tables
  - State Diagrams
  - Fuses
  - XOR Factors
- Operators
  - Logical: similar to PALASM

!	!A	NOT: one’s complement
&	A & B	AND
#	A # B	OR
$	A $ B	XOR: exclusive OR
!$	A !& B	XNOR exclusive NOR

ABEL

- Operators
  - Very limited arithmetic
    - - -A Twos complement (negation)
    - - A - B Subtraction
    - + A + B Addition
    - The following operators are not valid for sets:
      - * A*B Multiplication
      - / A/B Unsigned integer division
      - % A%B Modulus remainder from division
      - << A<<B Shift A left by B bits
      - >> A>>B Shift A right by B bits
      - ABC = 3 * 17;
  - Relational
    - == A == B Equal
    - != A != B Not equal
    - < A < B Less than
    - <= A <= B Less than or equal
    - > A > B Greater than
    - >= A >= B Greater than or equal

ABEL

- Statements
  - IF THEN ELSE
  - STATE MACHINE

- Architecture independent extensions:
  - .CLK: Clock input to an edge triggered flip-flop
  - .OE: Output enable
  - .PIN: Pin feedback
  - .FB: Register feedback
- Architecture specific dot extensions:
  - .J: J input to an JK-type flip-flop
  - .K: K input to an JK-type flip-flop
  - .R: R input to an SR-type flip-flop
  - And many more

Verilog

- Supports structural and behavioral descriptions
- Structural
  - Explicit structure of the circuit
  - Net list
  - 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

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

ELPL Embedded Low-Power Laboratory

Driving a simulation through a "testbench"

```
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

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

ELPL Embedded Low-Power Laboratory

Not included in the lecture

- Advanced Boolean optimization
    - Quine_McCluskey method
    - Espresso method