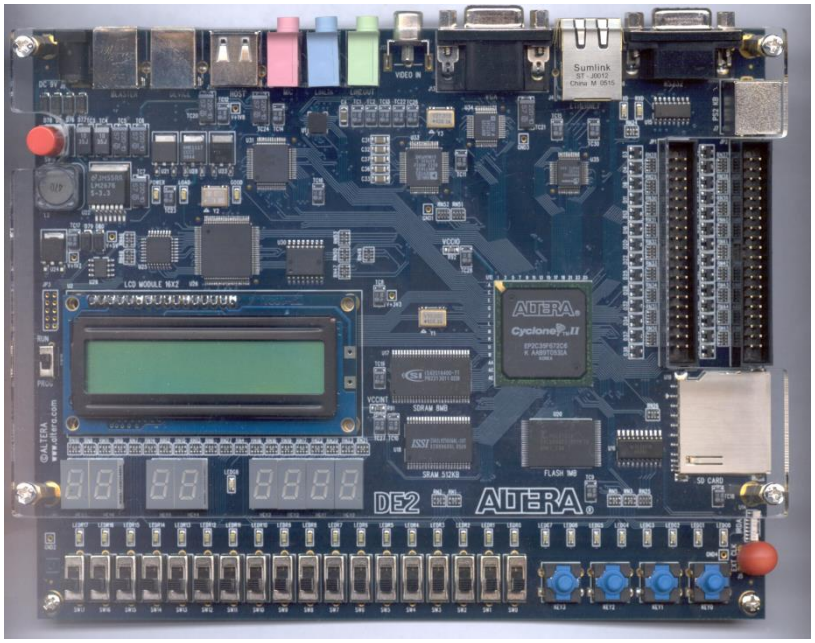
Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Teaching Philosophy &

Combinational Logic Review

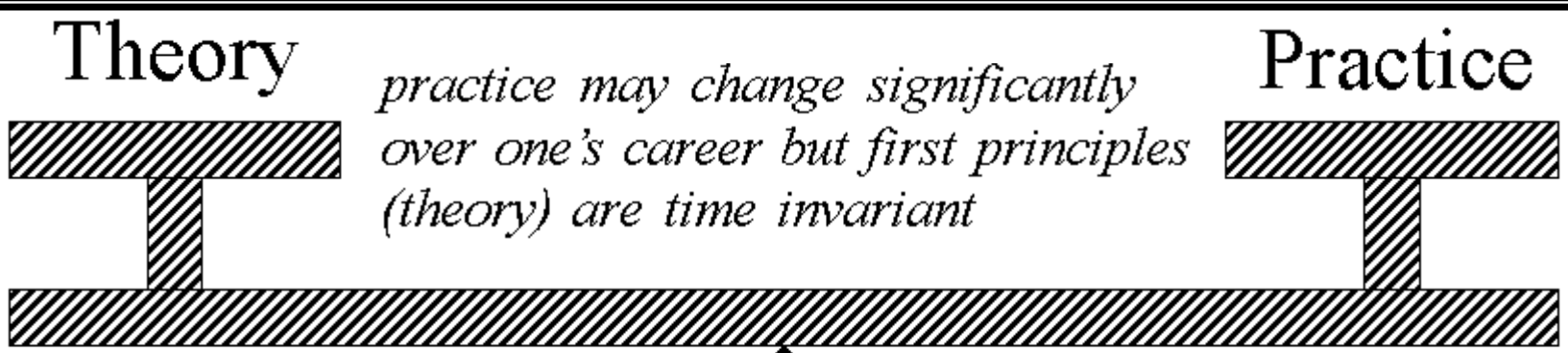


My Teaching Philosophy

I believe that my role as an instructor of engineering students is not simply to train the next generation of technologists, but rather to produce truly *educated* engineers who can advance the state-of-the-art in their discipline while continuously adapting to an ever changing technical landscape.

To accomplish this goal, I believe that there should be an appropriate balance between theory and practice -- the ability to view things in a general and abstract manner should be complemented by the hands-on experience gained by solving specific, nontrivial real world engineering problems.

Theory & Practice



Number Representation

Boolean Algebra

Timing Issues

Modeling Issues

Testability Issues

Quartus II, Design Entry Simulation, and FPGA Circuit Synthesis

ModelSim Design Entry and Simulation

DE-2 Rapid Prototyping Environment

Architectural alternatives for programmable logic

Course Goals

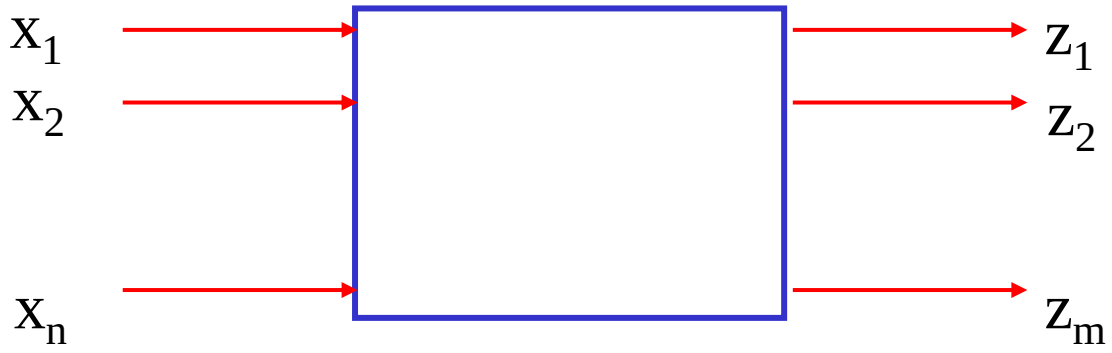
- Students will understand and appreciate that Verilog is not a ***programming language***, but rather is a ***hardware description language***.
- Students will understand the differences, advantages, and disadvantages between embedded software and programmable logic
- CPE and EE students with affinity towards hardware will discover and develop skills to enable them to bet their paycheck on these skills

Combinational Logic

- Has no memory =>
  present state depends only on the present input

$X = x_1\ x_2 \ldots\ x_n$

$Z = z_1\ z_2 \ldots\ z_m$



$$Z(t) = F(X(t))$$

Note:
Positive Logic – low voltage corresponds to a logic 0, high voltage to a logic 1
Negative Logic – low voltage corresponds to a logic 1, high voltage to a logic 0

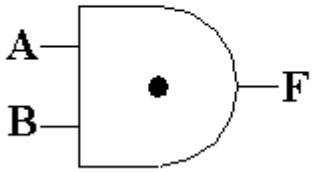
Boolean Algebra

- Basic mathematics used for logic design
  - A ‘first principle’ of ECE
- Terminology:
  - Logic value (two value -- high/low, 1/0)
  - Operator (AND, OR, NOT)
  - Variable (an input or output that can change value)
    - Literal (the appearance of a variable or its complement)
  - Constant (a logic high or low that does not change with time)
  - Boolean Expression (a logical expression that contains one or more Boolean operators, variables, or constants)
    - Note: Boolean Expressions can be converted into logic circuitry

Basic Boolean Logic Gates (operators)

AND

Logic Symbol



Truth Table

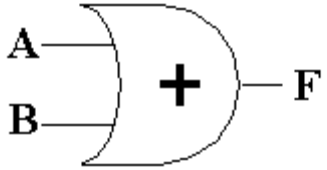
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Equation

F=AB or F=A•B

OR

Logic Symbol



Truth Table

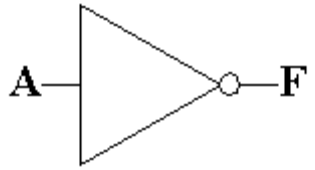
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Equation

F=A+B

NOT

Logic Symbol



Truth Table

A	F
0	1
1	0

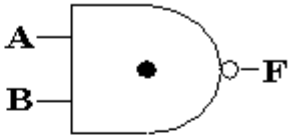
Boolean Equation

F=A' or F=$\overline{A}$

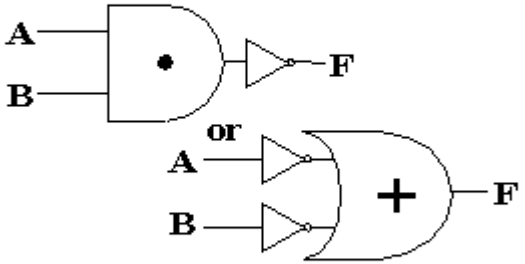
Common Derived Logic Gates (operators)

NAND

Logic Symbol



Equivalent Representations



Truth Table

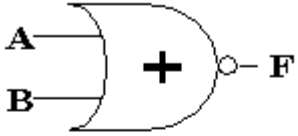
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Boolean Equation

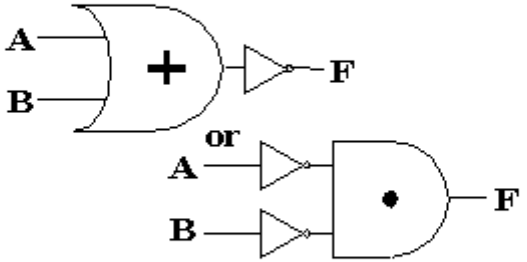
$F=(AB)'$

NOR

Logic Symbol



Equivalent Representations



Truth Table

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

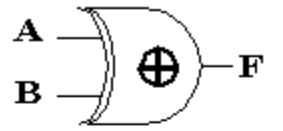
Boolean Equation

$F=(A+B)'$

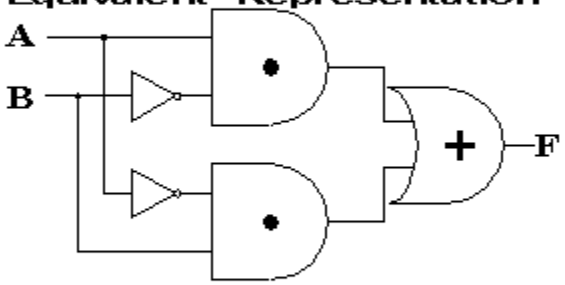
Common Derived Logic Gates (operators)

XOR

Logic Symbol



Equivalent Representation



Truth Table

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

Boolean Equation

$F=AB'+A'B$

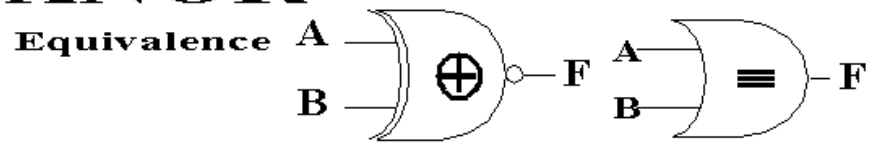
or

$F=A \oplus B$

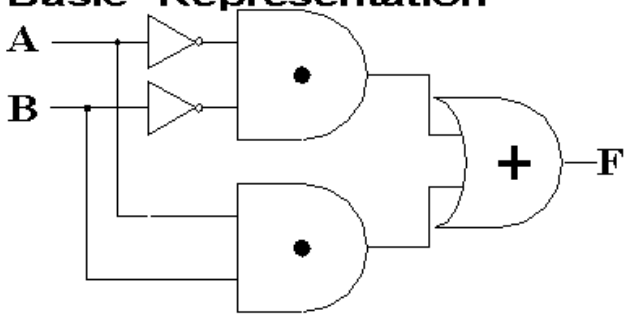
XNOR

Equivalence

Logic Symbols



Basic Representation



Truth Table

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

Boolean Equation

$F=A'B'+AB$

or

$F=A \equiv B$

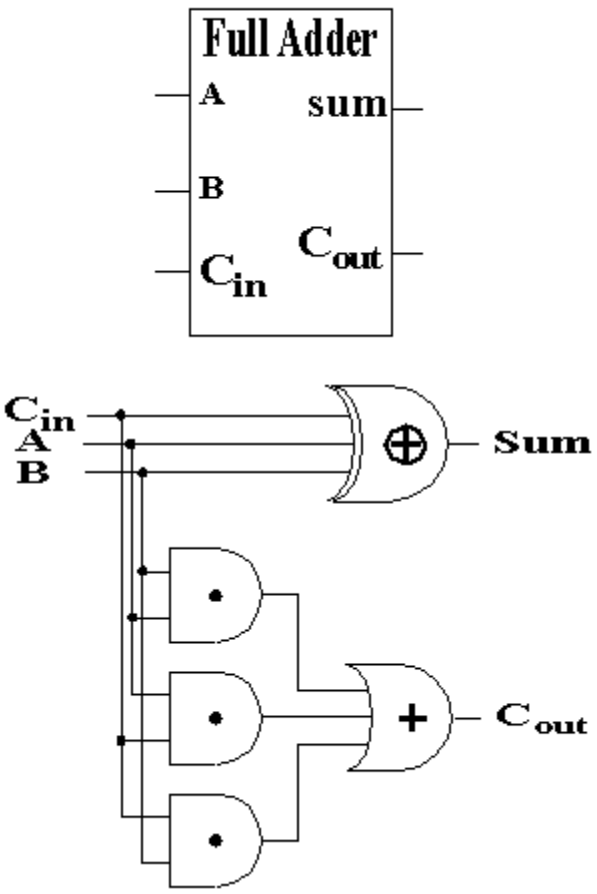
or

$F=(A \oplus B)'$

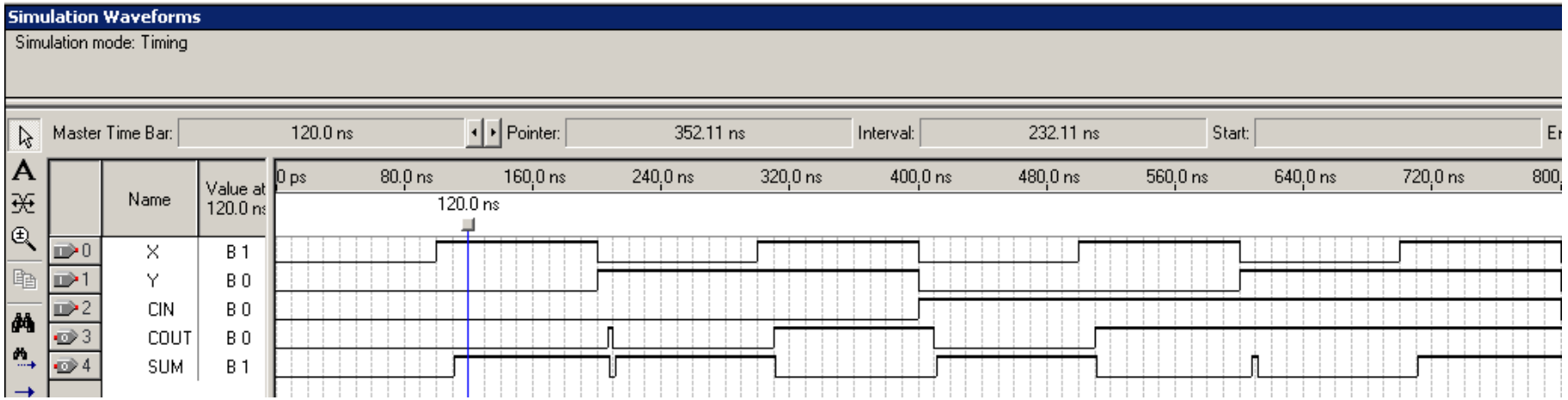
Full Adder

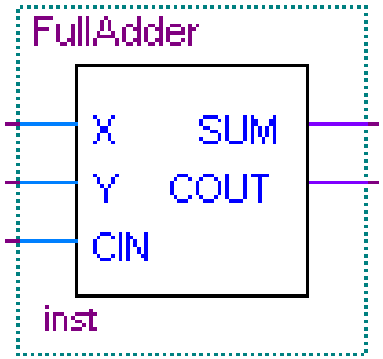
C_{in}	A	B	Sum
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

C_{in}	A	B	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



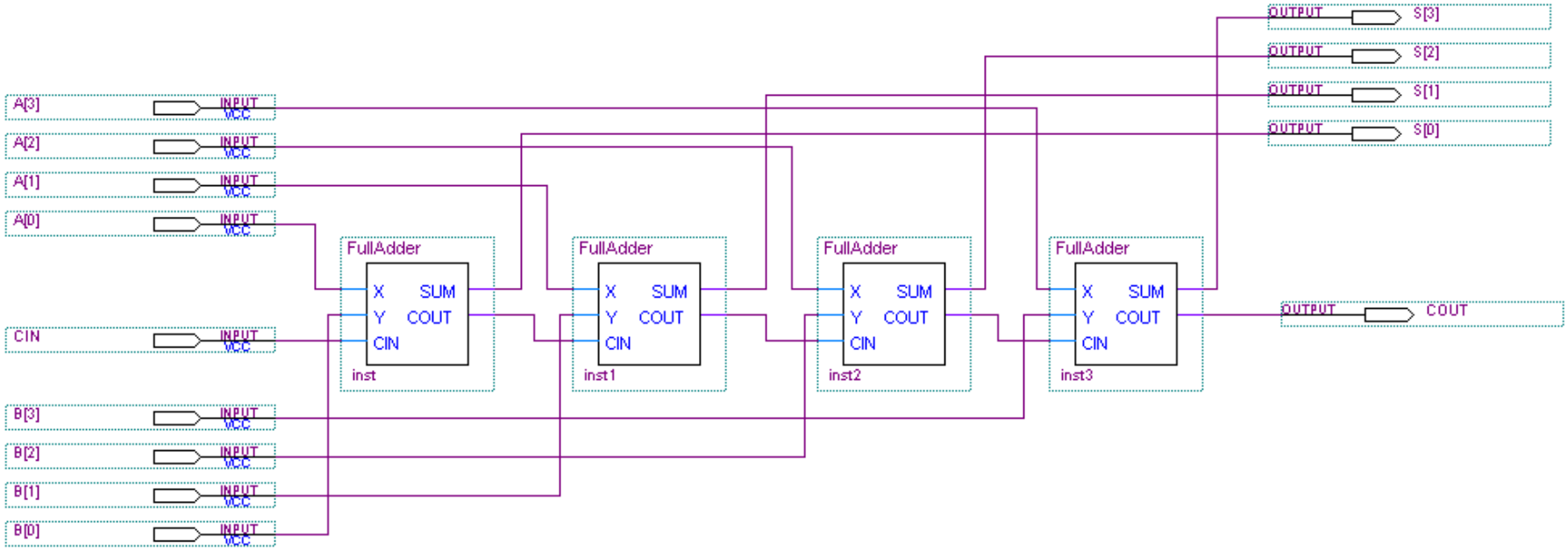
Quartus II Post Synthesis/Place and Route Simulation of Full Adder



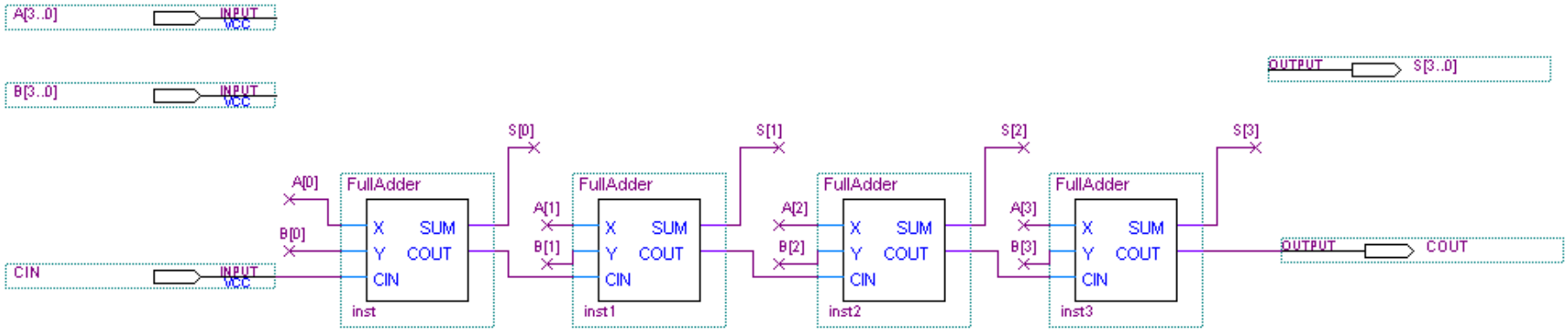


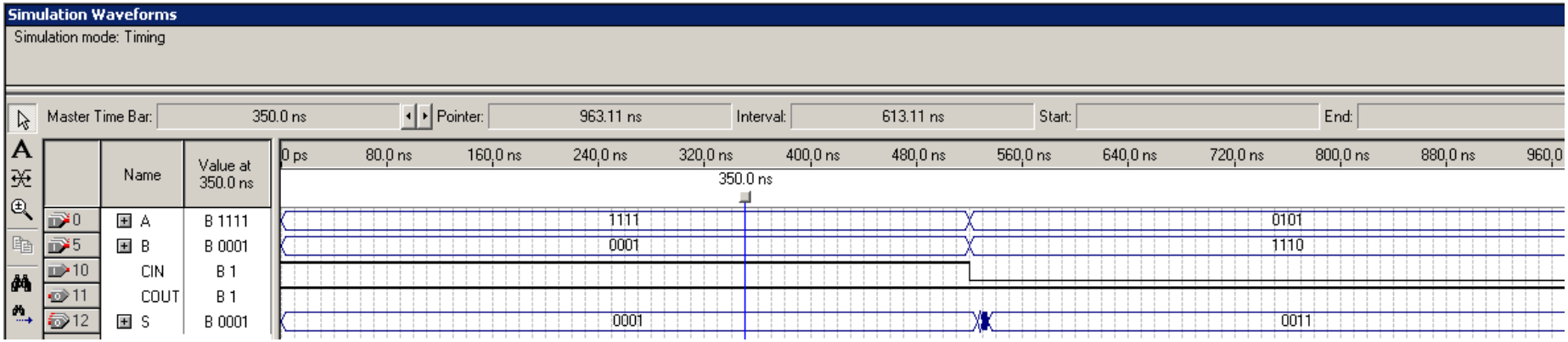
X	Y	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Hierarchical Design of a 4-Bit Adder



Same Design using Signal Naming Convention Instead of Wires





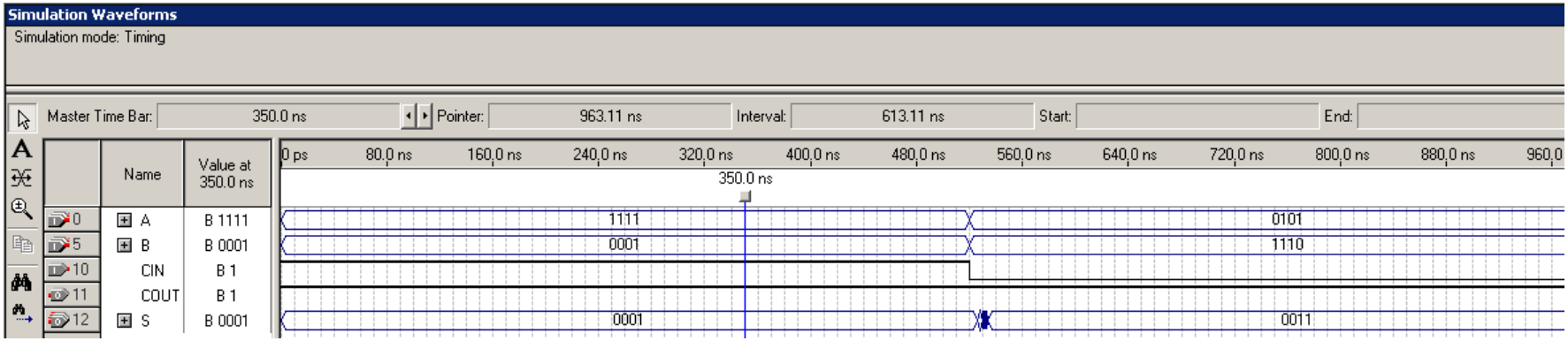
Quartus II Simulation of 4-bit Adder

```
  1 1 1 1
    1 1 1 1   A
    0 0 0 1   B
+         1   CIN
  1 0 0 0 1   S
```

COUT

```
  1 1 1 1
    0 1 0 1   A
    1 1 1 0   B
+         0   CIN
  1 0 0 1 1   S
```

COUT



Inversion

- DeMorgan's Laws:

$(X + Y)' = X'Y'$

The complement of the sum is the product of the complements

$(XY)' = X'+Y'$

The complement of the product is the sum of the complements

Complement (inverse) is formed by replacing each variable with its complement, replacing AND with OR, OR with AND, 0 with 1, and 1 with 0 while keeping the same precedence of operation

Inversion

$$F = X + E'K[C(AB+D')\bullet 1 + WZ'(G'H + 0)]$$

$$F' = X'(E + K' + [C' + (A' + B')\,D + 0][W' + Z + (G + H')\bullet 1])$$

Complement (inverse) is formed by replacing each variable with its complement, replacing AND with OR, OR with AND, 0 with 1, and 1 with 0 while keeping the same precedence of operation

Duality Principle

A Dual of a Boolean Expression is obtained by swapping the two multi-input operators (**AND** and **OR)** at the same time swapping the values (0 & 1) while keeping the same precedence of operation.

Note: Unlike Inversion the Dual does not involve complementing the variables

Duality Principle states that if two Boolean expressions are equal then the duals are also equal.

Duality and Positive and Negative Logic

- Positive Logic
  - low voltage corresponds to a logic 0
  - high voltage to a logic 1
- Negative Logic
  - Low voltage corresponds to a logic 1
  - High voltage to a logic 0
- A circuit that implements a Boolean expression in Positive Logic will Implement the Dual of that expression in negative logic

Laws and Theorems of Boolean Algebra

Operations with 0 and 1:			
$X + 0 = X$	(1-5)	$X \cdot 1 = X$	(1-5D)
$X + 1 = 1$	(1-6)	$X \cdot 0 = 0$	(1-6D)
Idempotent laws:			
$X + X = X$	(1-7)	$X \cdot X = X$	(1-7D)
Involution law:			
$(X')' = X$	(1-8)		
Laws of complementarity			
$X + X' = 1$	(1-9)	$X \cdot X' = 0$	(1-9D)
Commutative laws:			
$X + Y = Y + X$	(1-10)	$XY = YX$	(1-10D)
Associative laws:			
$(X + Y) + Z = X + (Y + Z) = X + Y + Z$	(1-11)	$(XY)Z = X(YZ) = XYZ$	(1-11D)
Distributive laws:			
$X(Y + Z) = XY + XZ$	(1-12)	$X + YZ = (X + Y)(X + Z)$	(1-12D)

Laws and Theorems of Boolean Algebra

Simplification theorems:

$XY + XY' = X$	(1-13)	$(X + Y)(X + Y') = X$	(1-13D)
$X + XY = X$	(1-14)	$X(X + Y) = X$	(1-14D)
$(X + Y')Y = XY$	(1-15)	$XY' + Y = X + Y$	(1-15D)

DeMorgan's laws:

$(X + Y + Z + \cdots)' = X'Y'Z'\cdots$	(1-16)	$(XYZ\cdots)' = X' + Y' + Z' + \cdots$	(1-16D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \ldots, X_n', 1, 0, \cdot, +)$			(1-17)

Duality:

$(X + Y + Z + \cdots)^D = XYZ\cdots$	(1-18)	$(XYZ\cdots)^D = X + Y + Z + \cdots$	(1-18D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \ldots, X_n, 1, 0, \cdot, +)$			(1-19)

Theorem for multiplying out and factoring:

$(X + Y)(X' + Z) = XZ + X'Y$	(1-20)	$XY + X'Z = (X + Z)(X' + Y)$	(1-20D)

Consensus theorem:

$XY + YZ + X'Z = XY + X'Z$	(1-21)	$(X + Y)(Y + Z)(X' + Z) = (X + Y)(X' + Z)$	(1-21D)

Simplifying Logic Expressions

- Combining terms
  - Use XY+XY'=X (1-13) , X+X=X (1-7)

$$ABC'D' + ABCD' = \underline{AB}C'\underline{D'} + \underline{AB}C\underline{D'} = ABD'$$

$$Cout = X'YCin + XY'Cin + XYCin' + XYCin$$

$$= (X'YCin + XYCin) + (XY'Cin + XYCin) + (XYCin' + XYCin)$$

$$= YCin + XCin + XY$$

Simplifying Logic Expressions

- Combining terms
  - Use XY+XY'=X (1-13) , DeMorgan's Laws (1-16)

$$(A+BC)(D+E') + A'(B'+C')(D+E') =$$

$$[A'(B'+C')]'(D+E') + A'(B'+C')(D+E') =$$

$$D+E'$$

Simplifying Logic Expressions

- Eliminating terms
  - Use X+XY=X (1-14) , X*X=X (1-7D),
  - Consensus theorem XY +YZ +X'Z = XY + X'Z (1-21)

$$A'B + A'BC = A'B$$

$$A'BC' + BCD + A'BD = \underline{C}'A'B + \underline{C}BD + \underline{A'BD} =$$

$$= A'BC' + BCD$$

Simplifying Logic Expressions

- Eliminating literals
  - Use X(Y+Z) = XY + XZ (1-12) , XY' + Y = X + Y (1-15D)

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

$$A'(B + C'D') + ABCD'$$

$$A'B + A'C'D' + ABCD'$$

$$B(A' + ACD') + A'C'D'$$

$$B(A' + CD') + A'C'D'$$

$$A'B + BCD' + A'C'D'$$

Simplifying Logic Expressions

- Adding Redundant terms
  - Add X*X'(=0) or Multiply by (X+X') (=1) or add in consensus term ([XY + YZ + X'Z = XY + X'Z (1-21)]

$$WX + XY + X'Z' + WY'Z' =$$

$$\underline{WX} + XY + \underline{X'Z'} + WY'Z' + WZ'$$

$$WX + XY + X'Z' + \cancel{\underline{WX}}\underline{Y'Z'} + \underline{WZ'}$$

$$[X + XY = X \text{ (1-14)}]$$

$$\underline{WX} + XY + \underline{X'Z'} + \cancel{\underline{WZ'}}$$

$$WX + XY + X'Z'$$

Theorems to Apply to Exclusive-OR

$X \oplus 0 = X$

$X \oplus 1 = X'$

$X \oplus X = 0$

$X \oplus X' = 1$

$X \oplus Y = Y \oplus X$ (Commutative law)

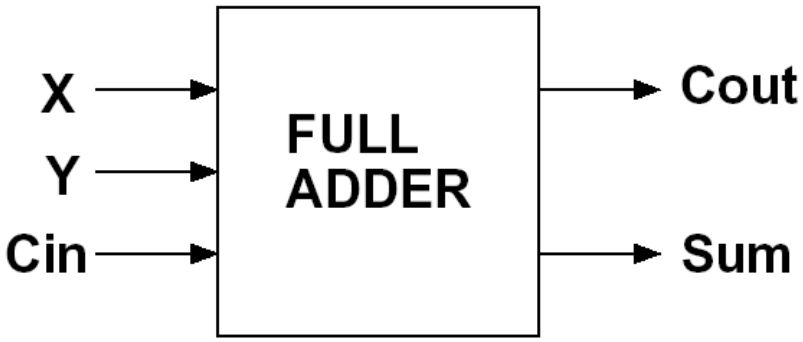
$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$ (Associative law)

$X(Y \oplus Z) = XY \oplus XZ$ (Distributive law)

$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y'$

Full Adder

Module



Truth table

X	Y	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Algebraic expressions F(inputs for which the function is 1):

Minterms

$$Sum = X'Y'Cin + X'YCin' + XY'Cin' + XYCin$$

$$Cout = X'YCin + XY'Cin + XYCin' + XYCin$$

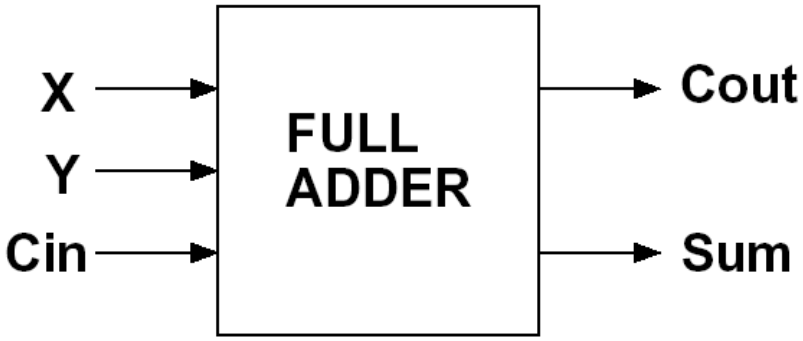
m-notation

$$Sum = m_1 + m_2 + m_4 + m_7 = \sum m(1, 2, 4, 7)$$

$$Cout = m_3 + m_5 + m_6 + m_7 = \sum m(3, 5, 6, 7)$$

Full Adder (cont'd)

Module



Truth table

X	Y	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Algebraic expressions F(inputs for which the function is 0):

Maxterms

$$Sum = (X + Y + Cin)(X + Y' + Cin')(X' + Y + Cin')(X' + Y' + Cin)$$

$$Cout = (X + Y + Cin)(X + Y + Cin')(X + Y' + Cin)(X' + Y + Cin)$$

M-notation

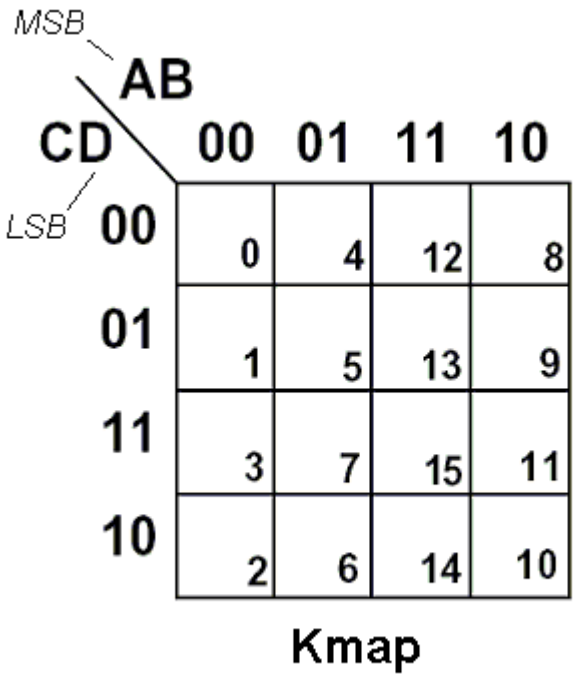
$$Sum = M_0 \cdot M_3 \cdot M_5 \cdot M_6 = \prod M(0,3,5,6)$$

$$Cout = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \prod M(0,1,2,4)$$

Karnaugh Maps

- Convenient way to simplify logic functions of 3, 4, 5, (6) variables
- Four-variable K-map
  - each square corresponds to one of the 16 possible minterms
  - 1 - minterm is present;
    0 (or blank) – minterm is absent;
  - X – don't care
    - the input can never occur, or
    - the input occurs but the output is not specified
  - adjacent cells differ in only one value =>
    can be combined

Location of minterms



Kmap

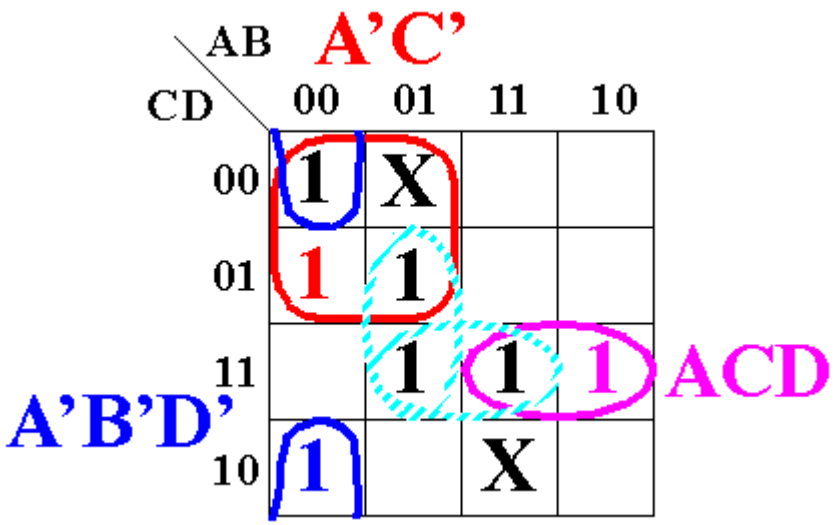
Sum-of-Products Representation

- Function consists of a sum of Prime Implicants
- **Prime Implicant**
  - a group of one, two, four, eight 1s on the Kmap represents a prime implicant if it cannot be combined with another group of 1s to eliminate a variable
- Prime Implicant is **Essential** if it contains a 1 that is not contained in any other Prime Implicant

Selection of Prime Implicants

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals

$$F(A,B,C,D) = \Sigma m(0,1,2,5,7,11,15) + \Sigma d(4,14)$$



Two minimum forms

F = A'C' + A'B'D' + ACD + A'BD

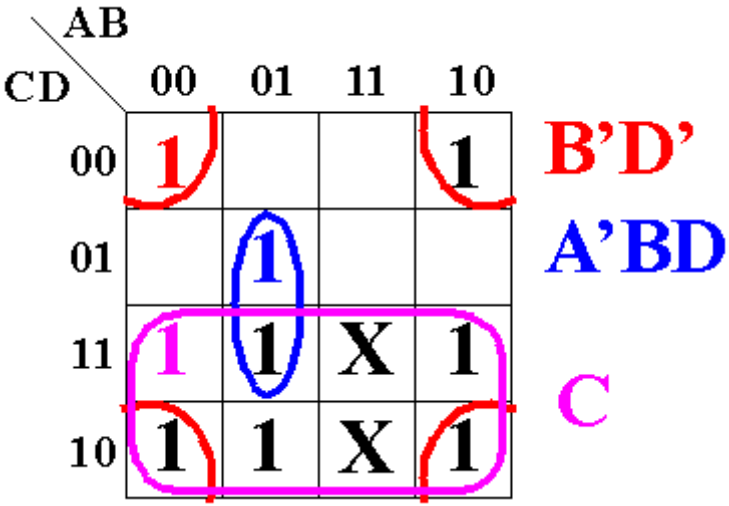
or

F = A'C' + A'B'D' + ACD + BCD

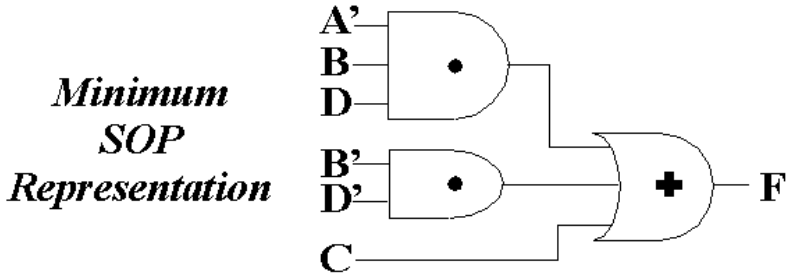
Minimum Sum-of-Products Representation

$$F(A,B,C,D) = \Sigma\, m(0,2,3,5,6,7,8,10,11) + \Sigma d(14,15)$$

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals



$$F(A,B,C,D) = B'D' + A'BD + C$$



Inversion

- DeMorgan's Laws:

$(X + Y)' = X'Y'$

The complement of the sum is the product of the complements

$(XY)' = X'+Y'$

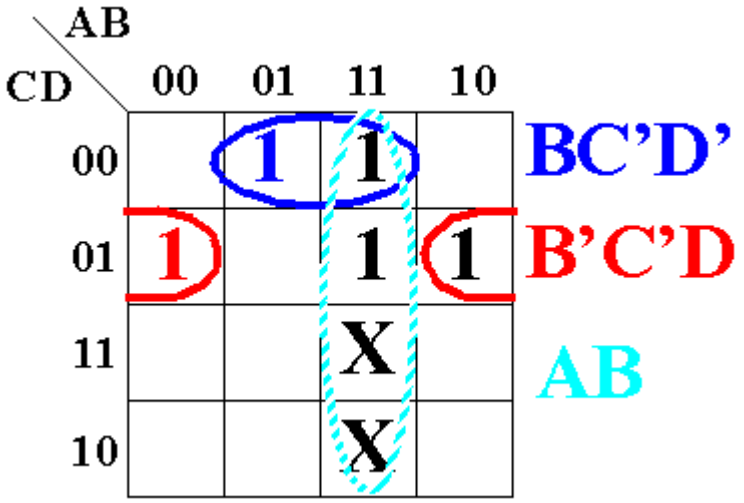
The complement of the product is the sum of the complements

Minimum Product-of-Sums Representation

$$F(A,B,C,D) = \Sigma m(0,2,3,5,6,7,8,10,11) + \Sigma d(14,15)$$

$$F'(A,B,C,D) = \Sigma m(1,4,9,12,13) + \Sigma d(14,15)$$

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals



$$F'(A,B,C,D) = BC'D' + B'C'D + AB$$

$$F(A,B,C,D) = [F'(A,B,C,D)]' \text{ (Involution law)}$$

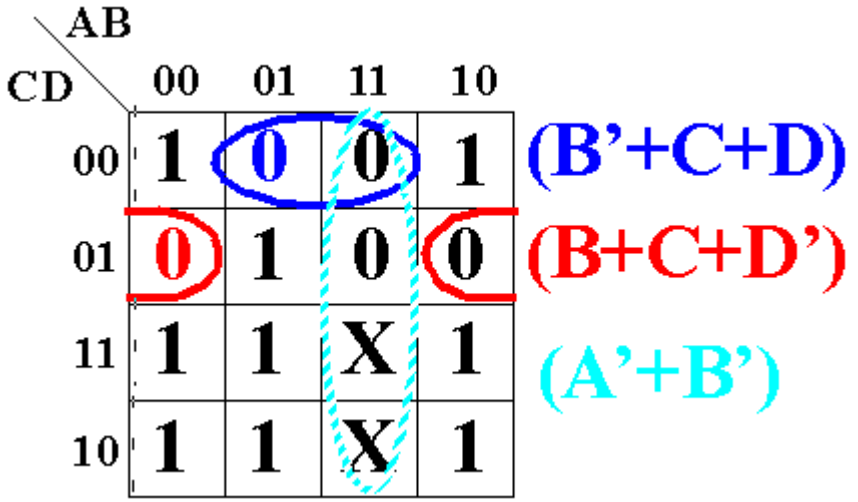
$$F(A,B,C,D) = (B'+C+D)(B+C+D')(A'+B')$$

(DeMorgan's Laws)

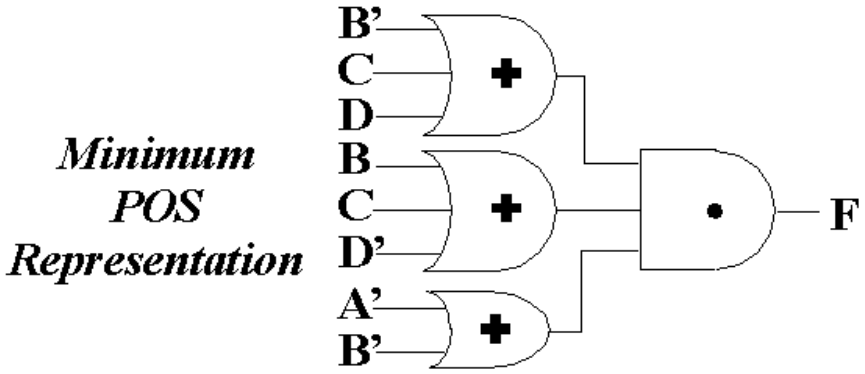
Minimum Product-of-Sums Representation

$$F(A,B,C,D) = \Sigma\, m(0,2,3,5,6,7,8,10,11) + \Sigma d(14,15)$$

- 1. Choose a 0 term that has not been covered yet
- 2. Find all 0s and Xs adjacent to that minterm
- 3. If a single term covers the 0 term and all adjacent 0s and Xs, then that term is essential, so select that term
- 4. Repeat steps 1, 2, 3 until all essential groupings have been chosen
- 5. Find a minimum set of groupings that cover the remaining 0s on the map. If there is more than one such set, choose a set with a minimum number of literals
- 6. Interpret results in POS manner



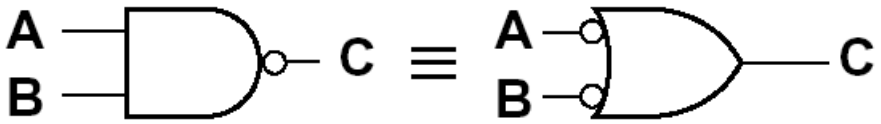
$$F(A,B,C,D) = (B'+C+D)(B+C+D')(A'+B')$$



Designing with NAND and NOR Gates (1)

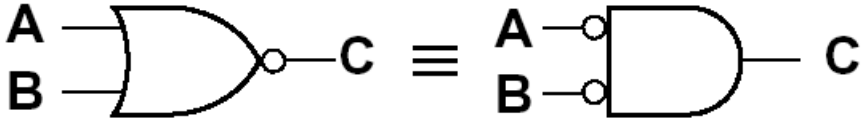
- Implementation of NAND and NOR gates is easier than that of AND and OR gates (e.g., CMOS)

NAND:



C = (AB)' = A' + B'

NOR:



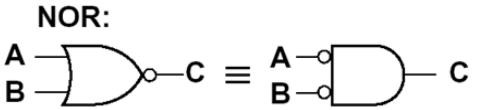
C = (A+B)' = A'B'

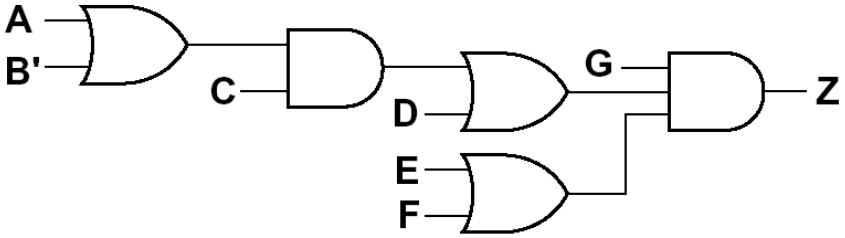
Designing with NAND and NOR Gates (2)

- Any logic function can be realized using only NAND or NOR gates => NAND/NOR is functionally complete
  - NAND function is complete –
    can be used to generate any logical function;
  - 1: (a * a')' = (0)' = 1
  - 0: [(a * a')' * (a*a')'] = [1 * 1]' = [1]' = 0
  - a': (a * a)' = (a)' = a'
  - ab: [(a * b)' * (a * b)']' = [(a * b)']' = a * b = ab
  - a+b: (a' * b')' = a + b

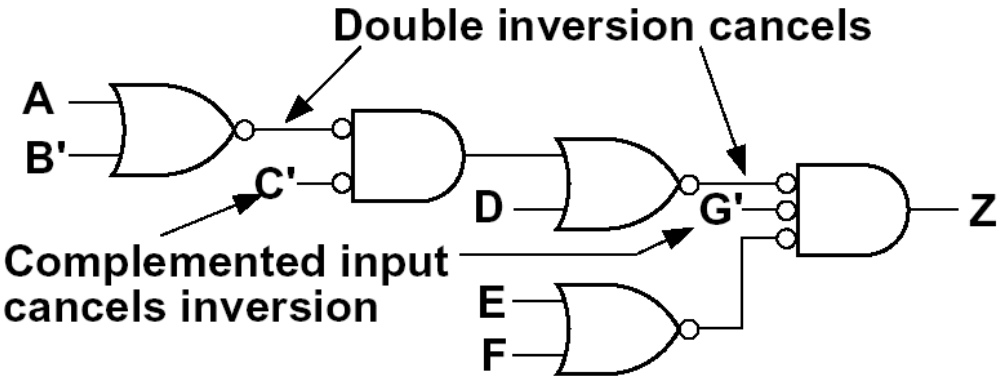
Using only NOR Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example:





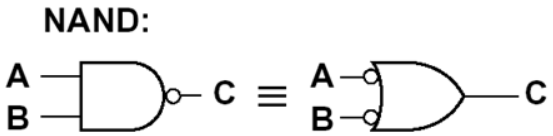
(a) AND-OR network

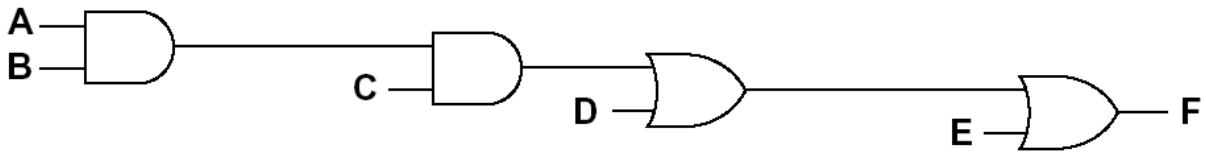


(b) Equivalent NOR-gate network

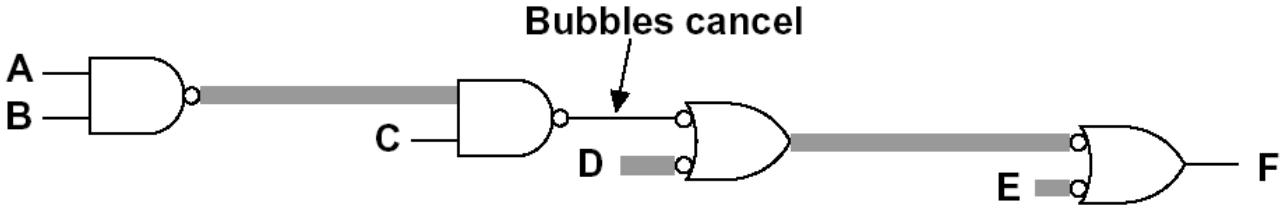
Using only NAND Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example

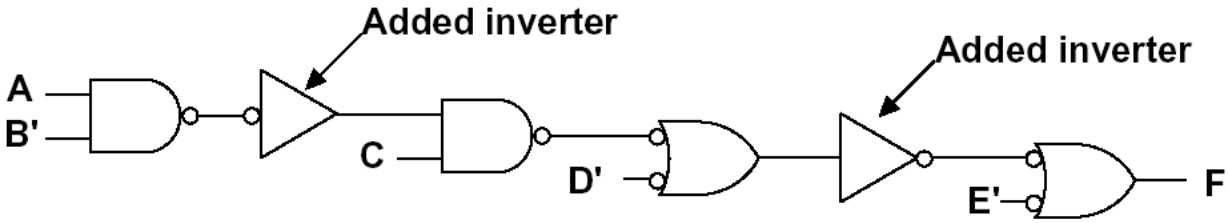




(a) AND_OR network



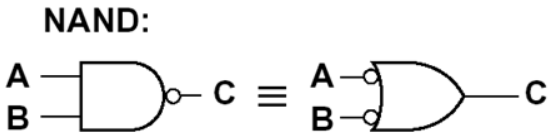
(b) First step in NAND conversion

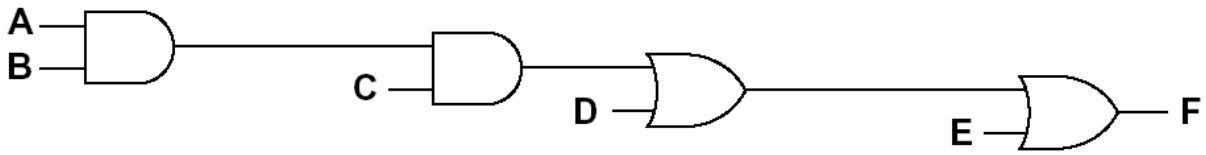


(c) Completed conversion

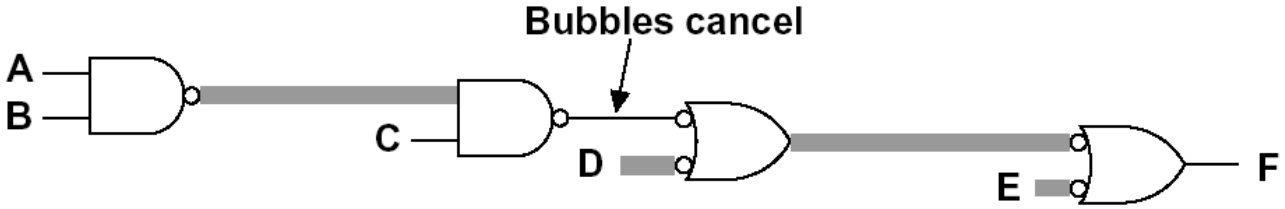
Using only NAND Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example

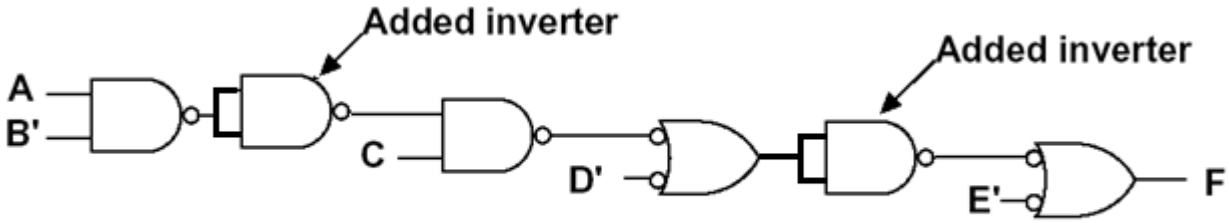




(a) AND_OR network



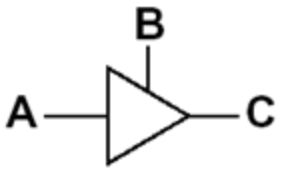
(b) First step in NAND conversion



(c) Completed conversion

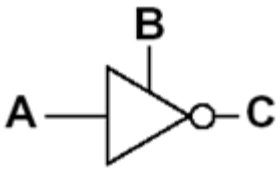
Tristate Logic and Busses

- Four kinds of tristate buffers
  - B is a control input used to enable and disable the output
  - Tristate Hi-Z state can be viewed simplistically as disconnecting the output C from the rest of the logic (more accurate description will be discussed later)



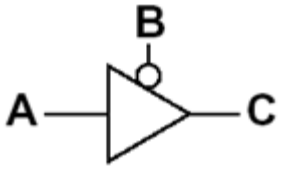
B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(a)



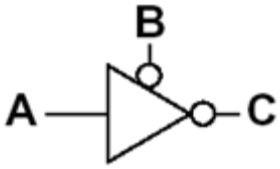
B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

(b)



B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

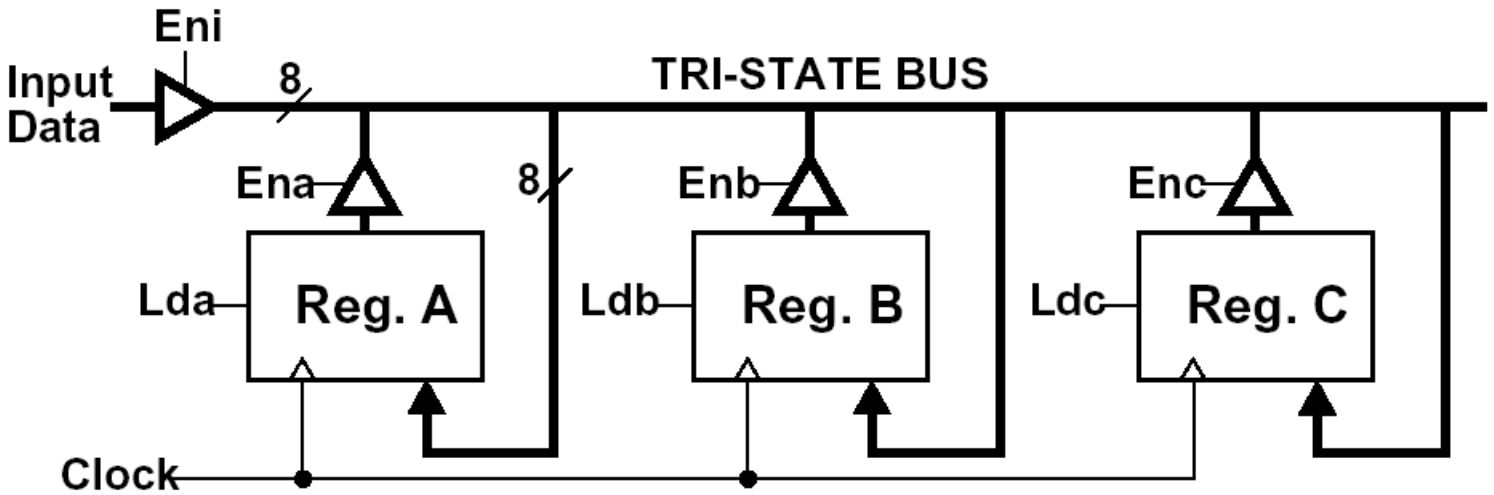
(c)



B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

(d)

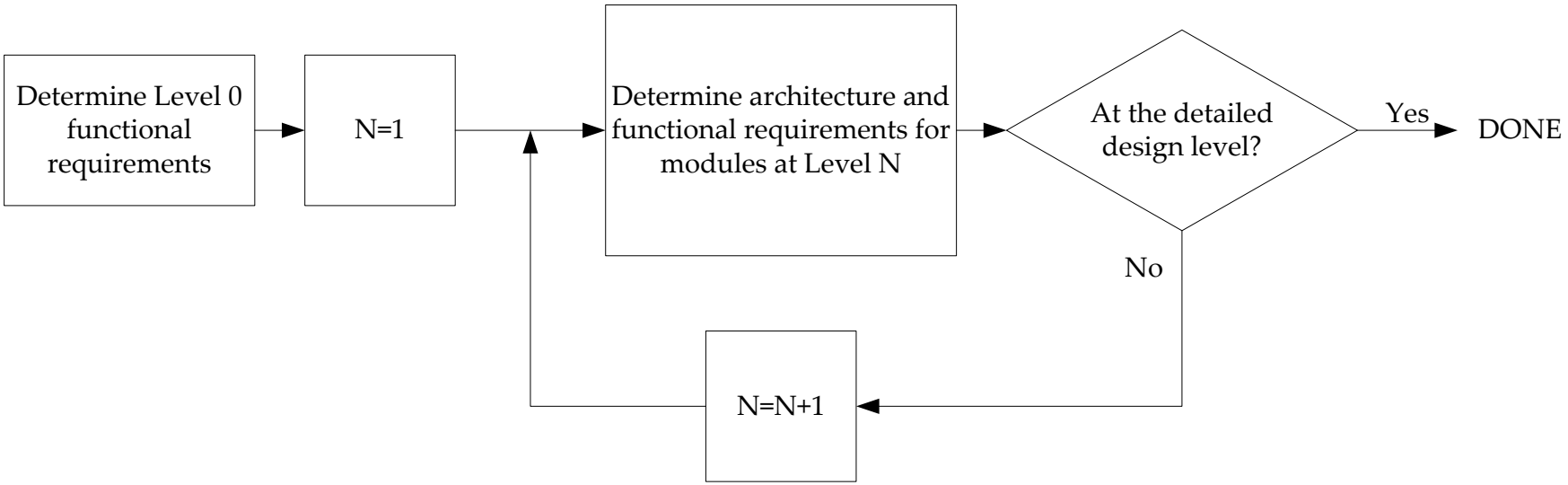
Data Transfer Using Tristate Bus



Hierarchical Design Using Schematic Capture

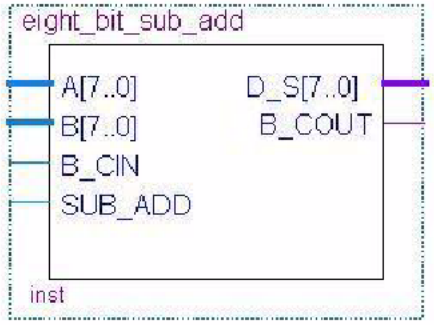
Form of Functional Decomposition

- **Recursively divide and conquer**
  - **Split a module into several sub-modules**
  - **Define the input, output, and behavior**
  - **Stop when you reach realizable components**



Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, *eight_bit_sub_add* symbol)

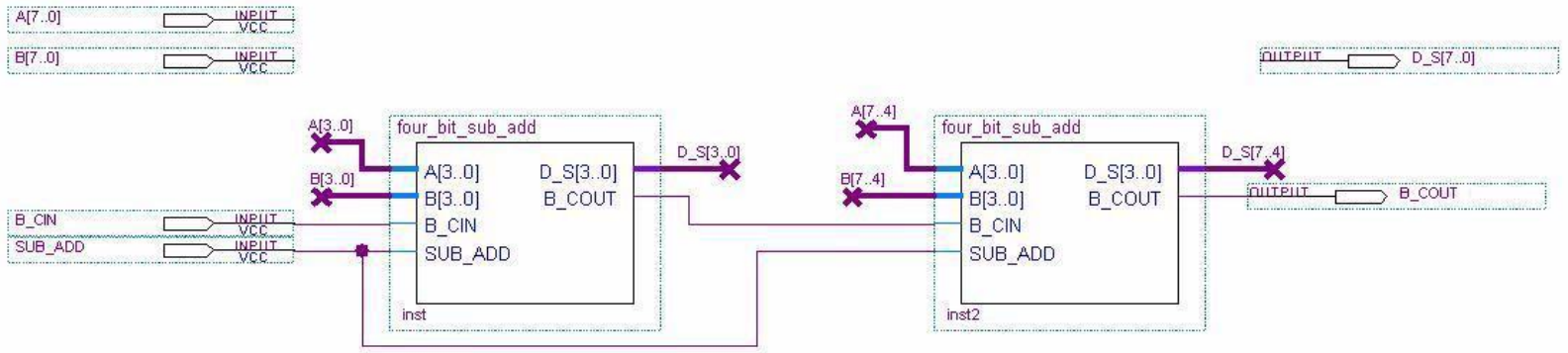


Function:

```
if (SUB_ADD) = '1' then
  D_S <= A - B - B_Cin --subtract B and B_Cin from A
  if (B>A) B_Cout <= '1' {borrow}
  else B_Cout <= '0' {no borrow}
else { -- if SUB_ADD='0'
  D_S <= (A + B )[7:0]+B_Cin; {lower 8 bits of A + B + B_Cin}
  B_Cout <=  (A+B+B_Cin)[8]; {most significant bit of A+B+B_Cin}
}
```

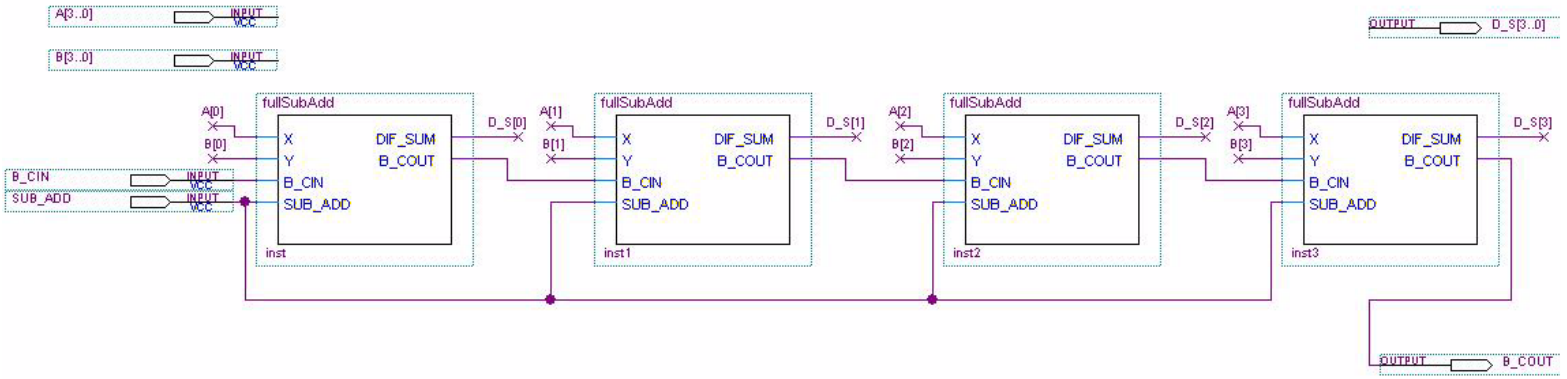
Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 1 -- *eight_bit_sub_add* module)



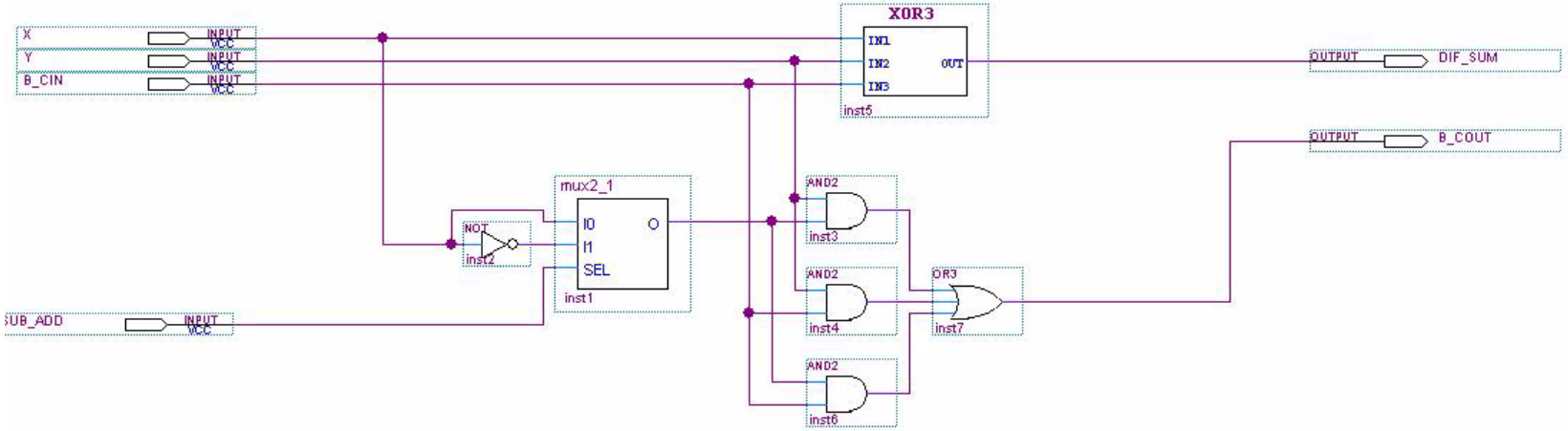
Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 2 -- *four_bit_sub_add* module)

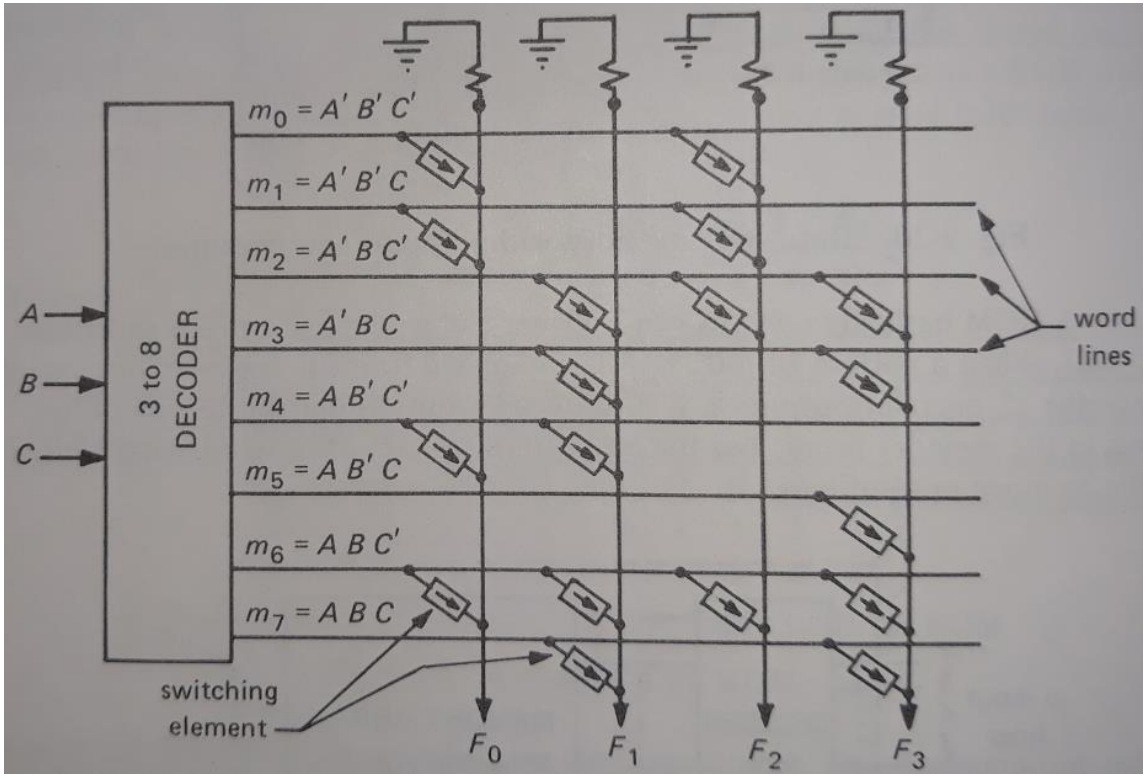


Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 3 -- *fullSub_Add module*)



8 word by 4 bit ROM used to Implement 4 Boolean Functions



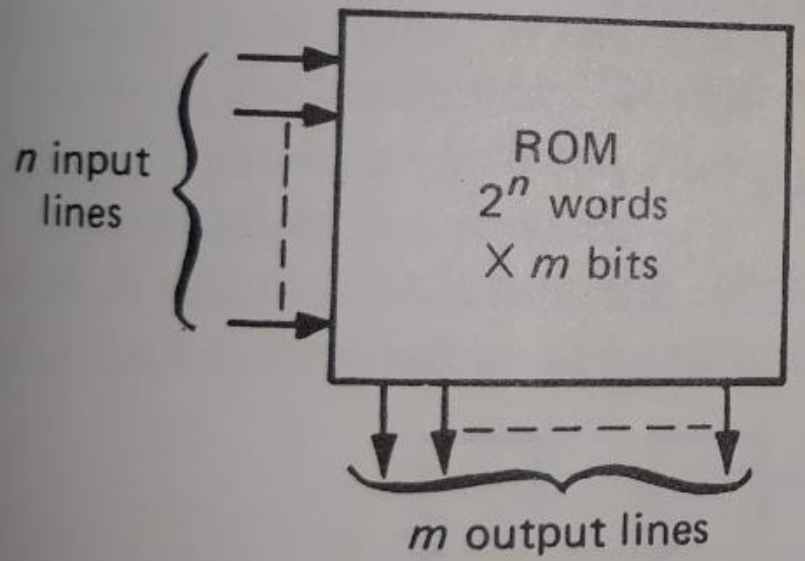
$$F_0 = \Sigma m(0, 1, 4, 6) = A'B' + AC'$$

$$F_1 = \Sigma m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \Sigma m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \Sigma m(2, 3, 5, 6, 7) = AC + B$$

ROM Memory with n Inputs and m Outputs



n input variables	m output variables
00 . . . 00	100 . . . 110
00 . . . 01	010 . . . 111
00 . . . 10	101 . . . 101
00 . . . 11	110 . . . 010
.	.
.	.
.	.
11 . . . 00	001 . . . 011
11 . . . 01	110 . . . 110
11 . . . 10	011 . . . 000
11 . . . 11	111 . . . 101

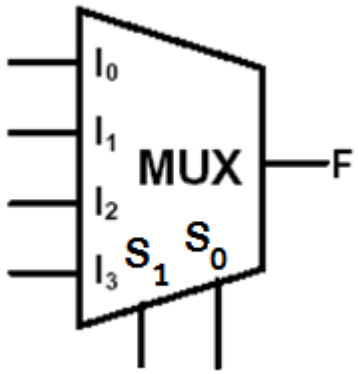
typical data array stored in ROM (2^n words of m bits each)

Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

F = A'B' + B'C

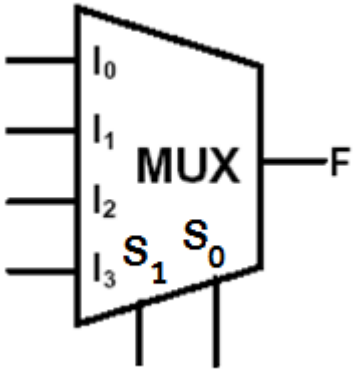
Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

F = A'B' + B'C



Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

$F = A'B' + B'C$

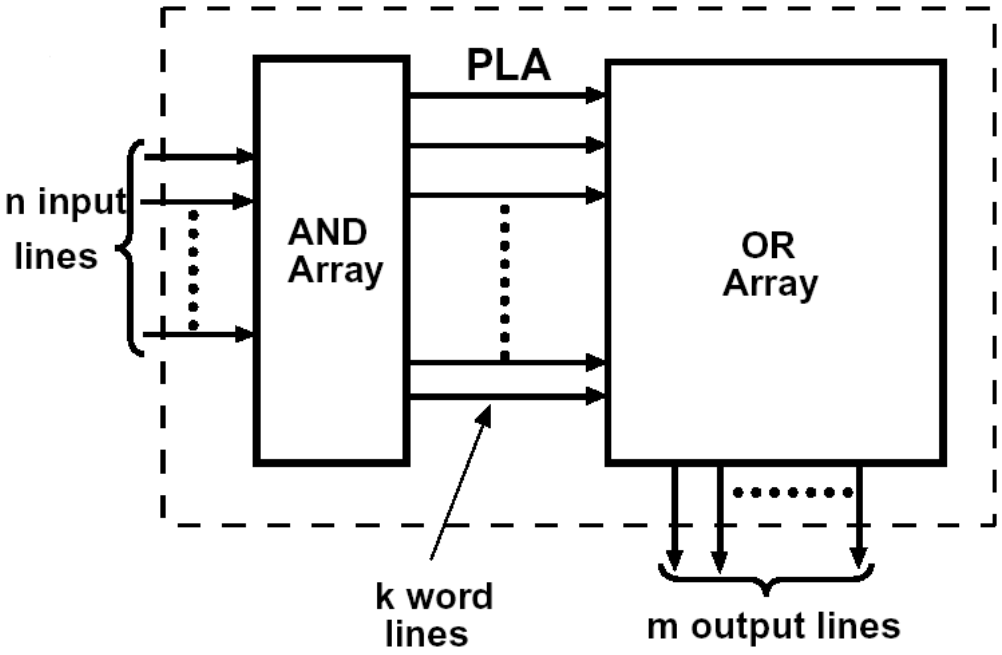


Truth Table

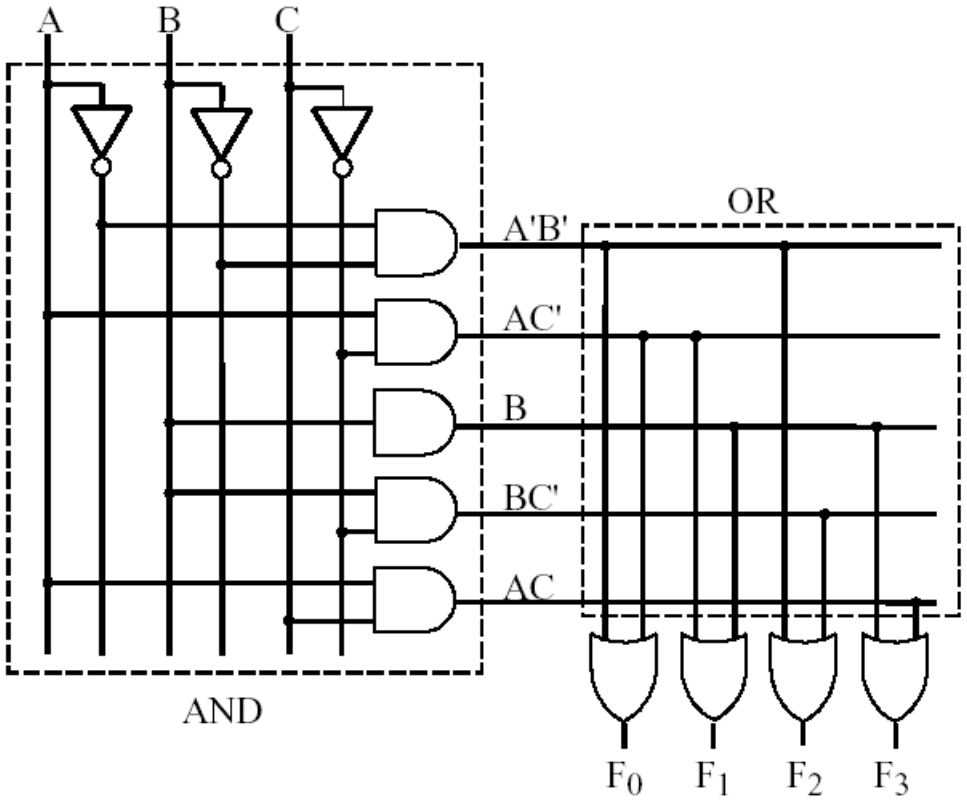
A	B	C	F

Programmable Logic Arrays (PLAs)

- Legacy Programmable Logic Device that incorporates two-level logic
  - *n* inputs and *m* outputs – *m* functions of *n* variables
  - AND array – realizes product terms of the input variables
  - OR array – ORs together the product terms



AND-OR Array Equivalent

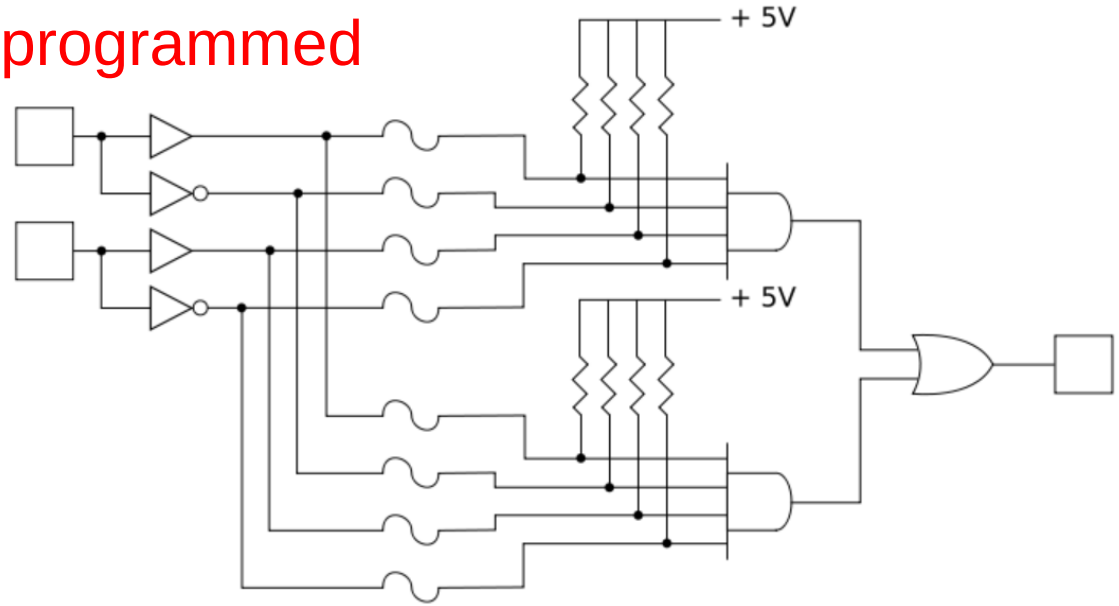


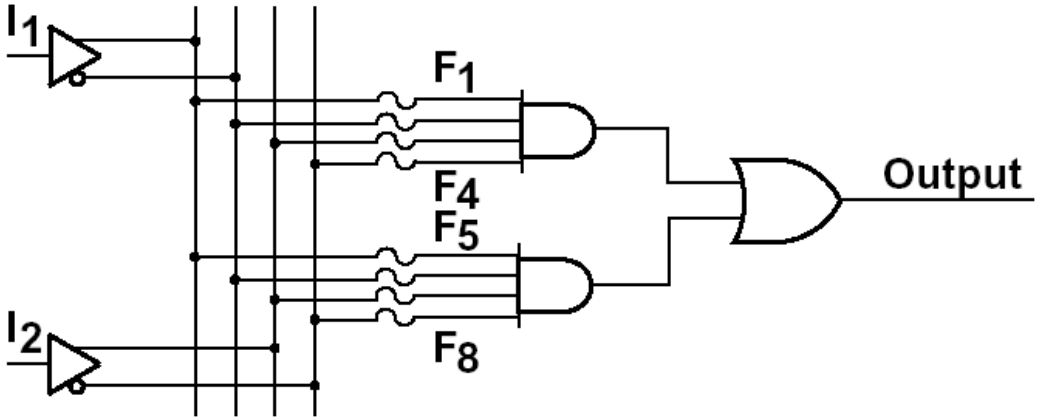
Programmable Array Logic (PALs)

- PAL is a special case of PLA
  - AND array is programmable and OR array is fixed
- PAL is
  - less expensive
  - easier to program

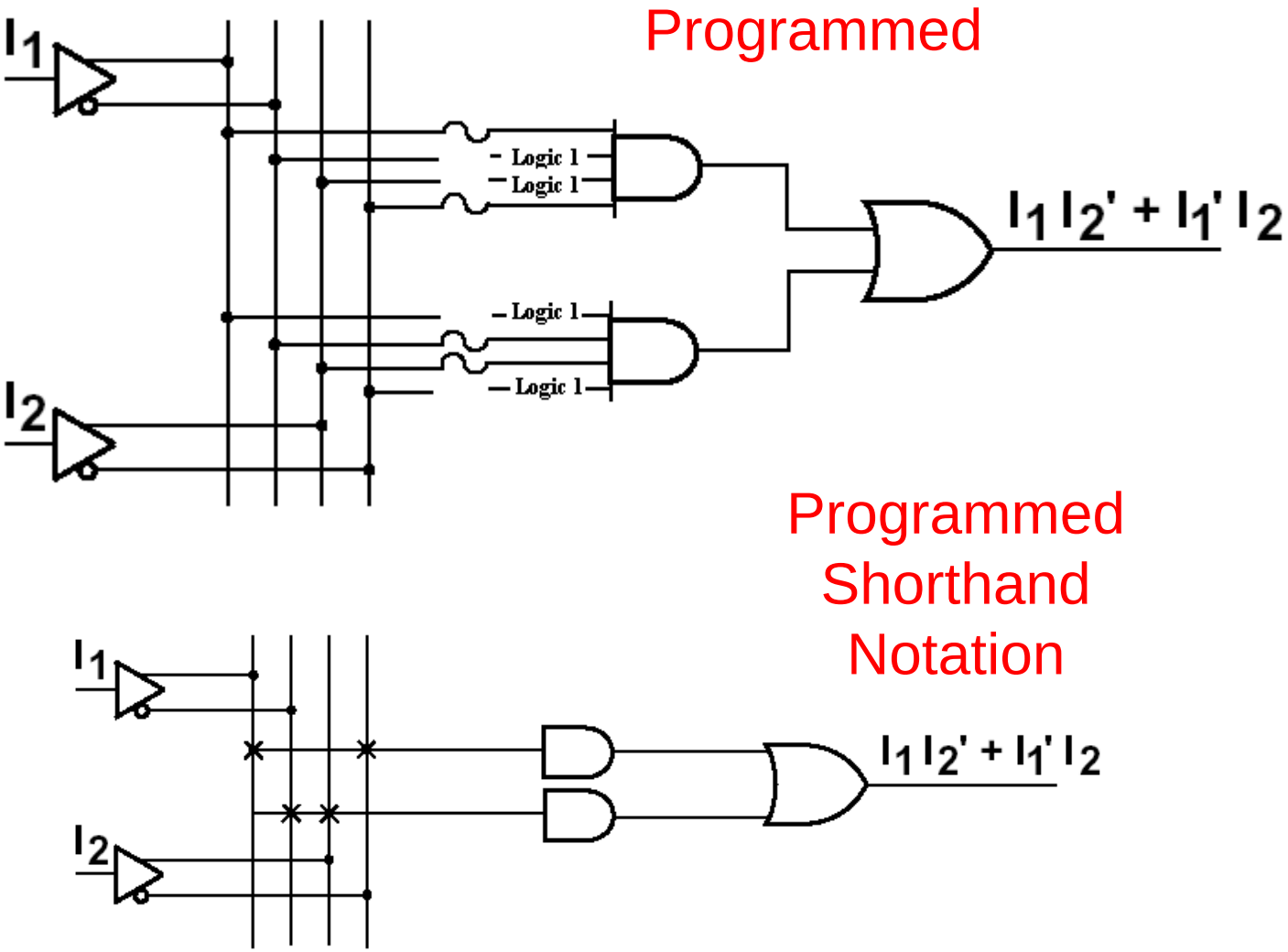
Programmable Array Logic (PALs)

Unprogrammed

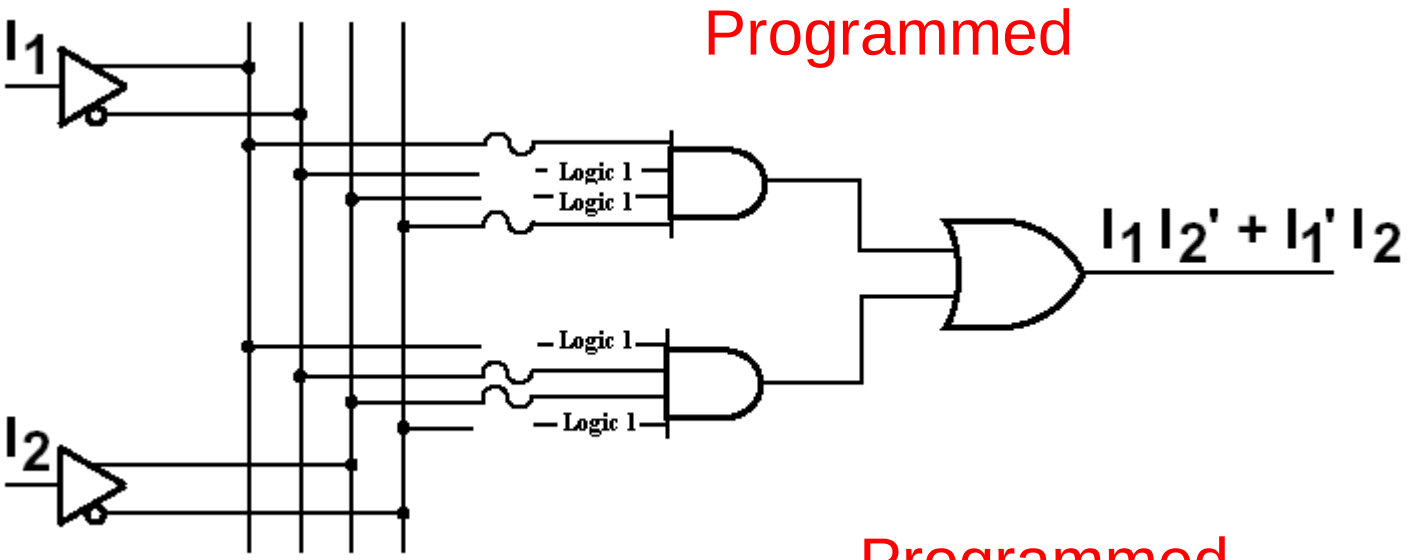


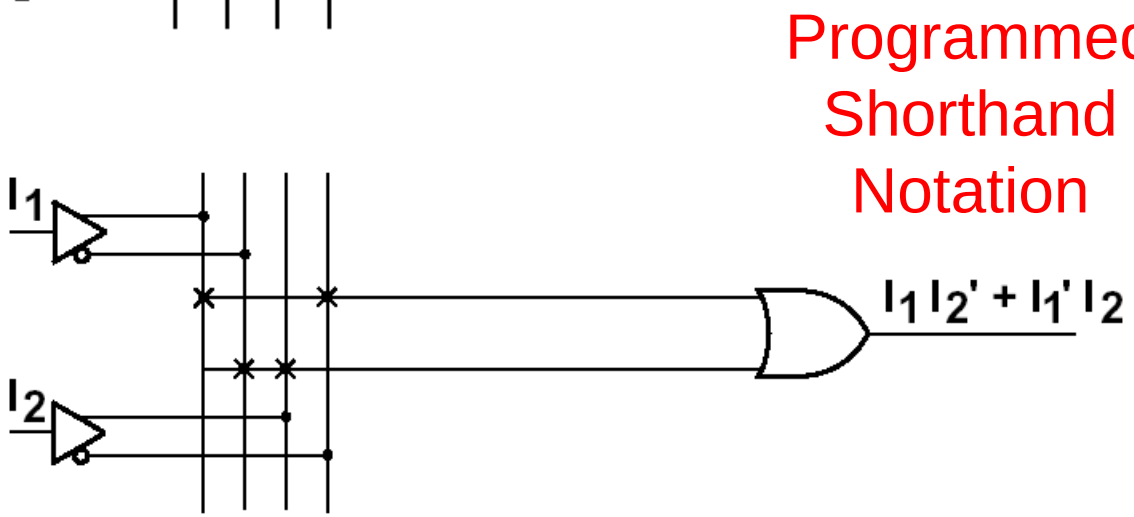


Programmable Array Logic (PALs)



Programmable Array Logic (PALs)

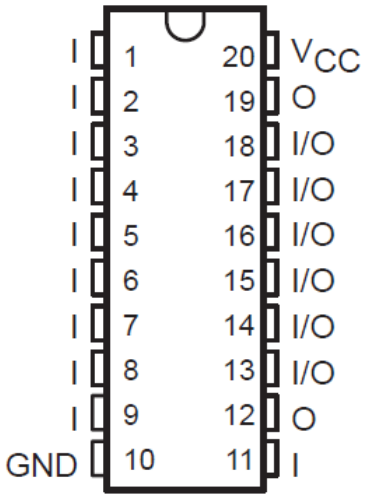




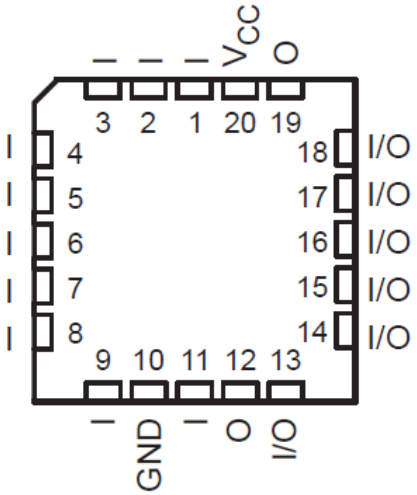
PALs

- Typical PALs have
  - from 10 to 20 inputs
  - from 2 to 10 outputs
  - from 2 to 8 AND gates driving each OR gate
  - often include tri-state logic

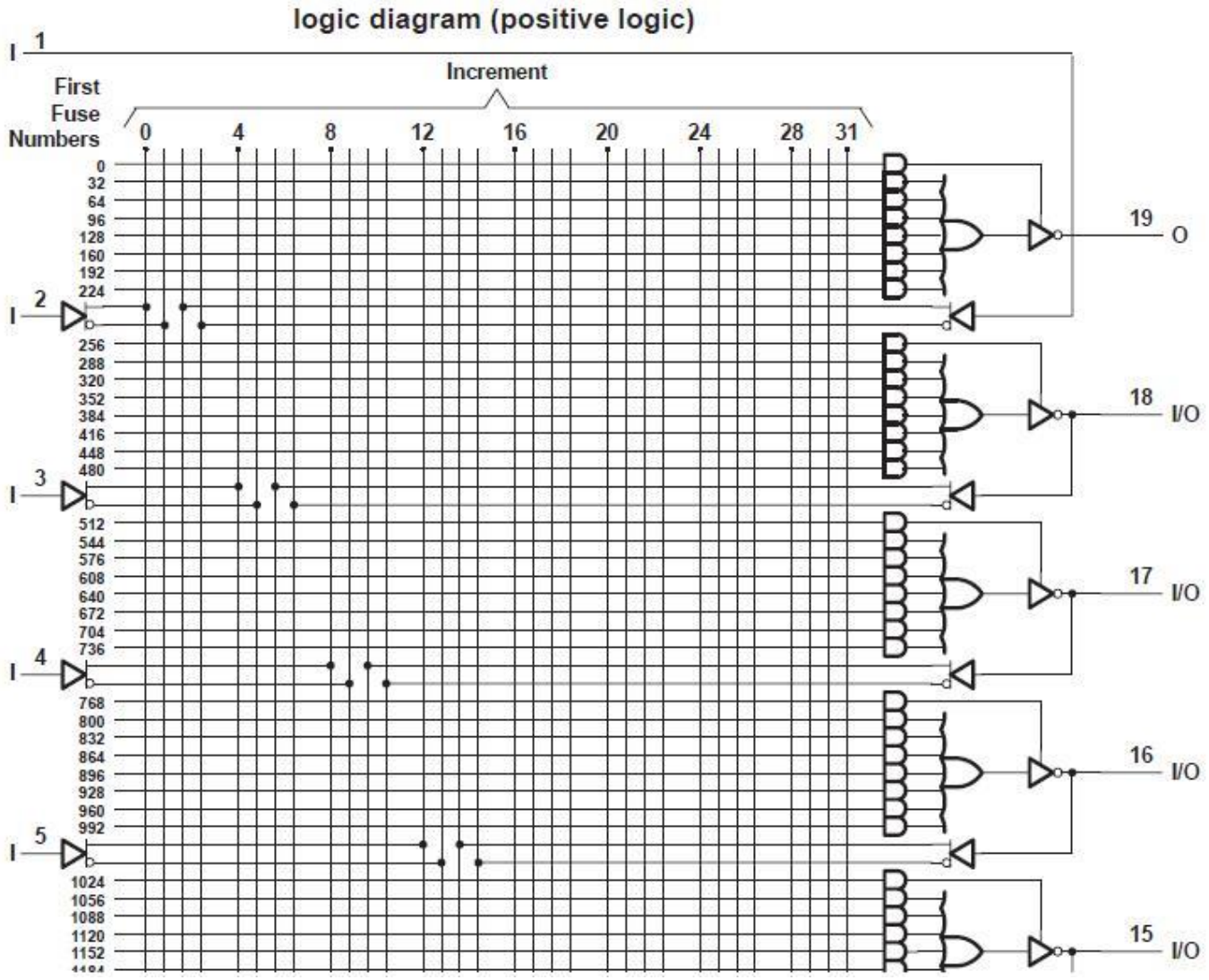
TIBPAL16L8'
C SUFFIX . . . J OR N PACKAGE
M SUFFIX . . . J OR W PACKAGE
(TOP VIEW)



TIBPAL16L8'
C SUFFIX . . . FN PACKAGE
M SUFFIX . . . FK PACKAGE
(TOP VIEW)



Logic Diagram for 16L8 PAL



Logic Diagram for 16L8 PAL