Digital Logic Design

Gate-Level Minimization

Outline

- The Map Method
- 2,3,4 variable maps
- 5 and 6 variable maps (very briefly)
- Product of sums simplification
- Don’t Care conditions
- NAND and NOR implementation
- Other 2-level implementation
- Hardware Description language (HDL)

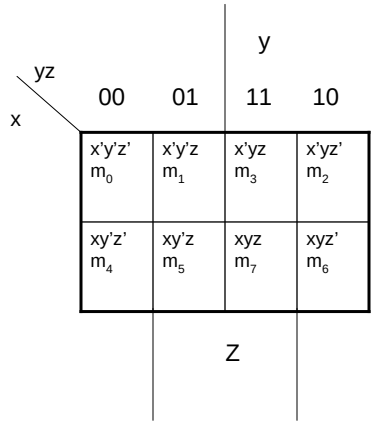
The Map Method

- After constructing the map. We mark the squares whose minterms.
- Any two adjacent squares in the map differ by only one variable, primes in one square and unprimed in the other.
- The sum of the elements in these 2 squares, can be simplified to an and gates that does not contain that literal.
- The more adjacent squares we combine them together, the simple the term will be.

2-variable map

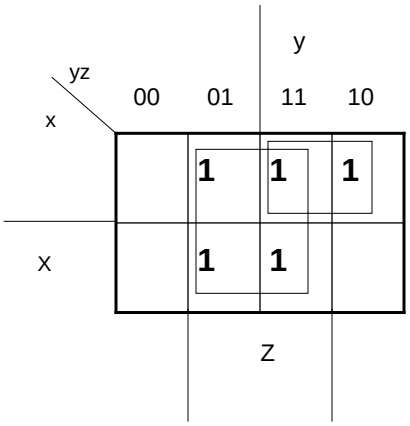
Y X	0	1
0	m_0	m_1
1	m_2	m_3

3-variable map



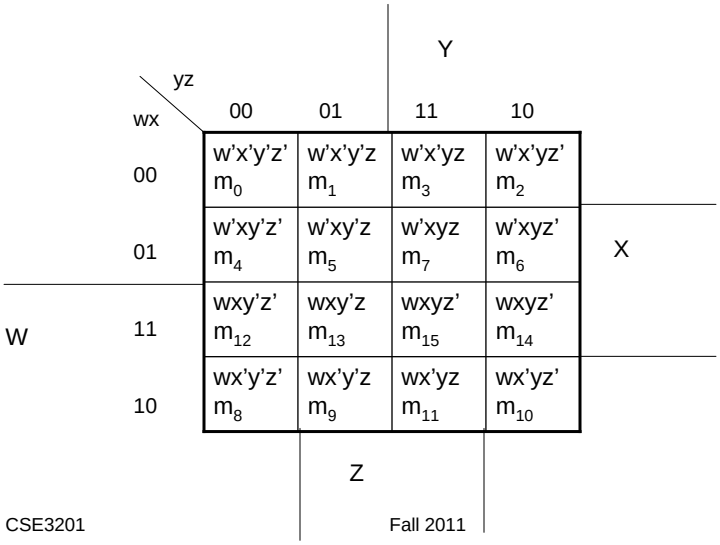
3-variable map

F=X'Z+X'Y+XY'Z+YZ

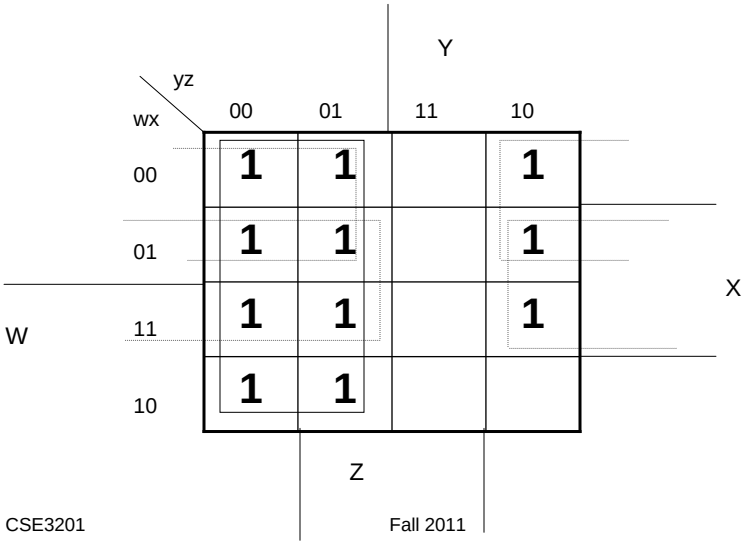


F=Z+X'Y

4-variable map



4-variable map



5-Variable Map

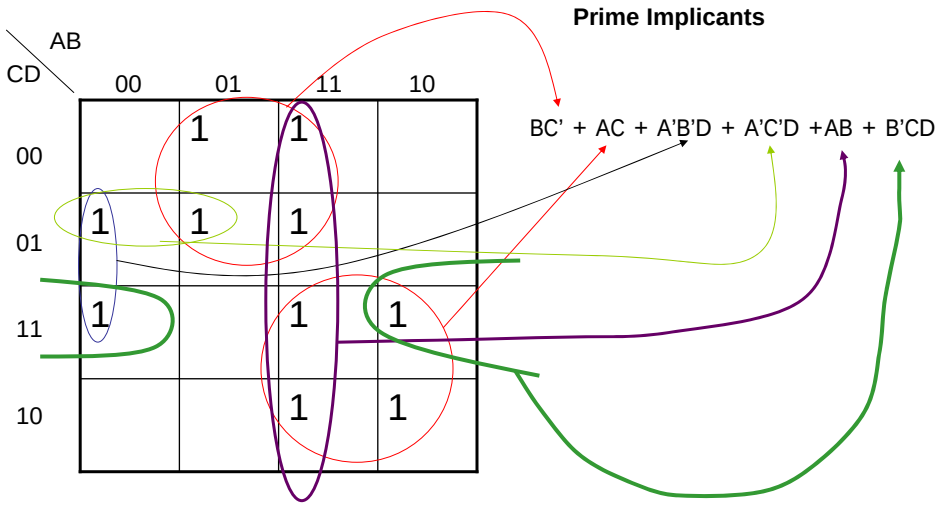
- Mention an example for 5 and 6 very briefly,
- Too complicated

Implicants

- Any single 1, or a group of ones that could be combined together on a Karnaugh map of a function F represents a product term that we call an implicant.
- **prime Implicant**: is a product term obtained by combining together the maximum possible number of adjacent squares in the map

Prime Implicants

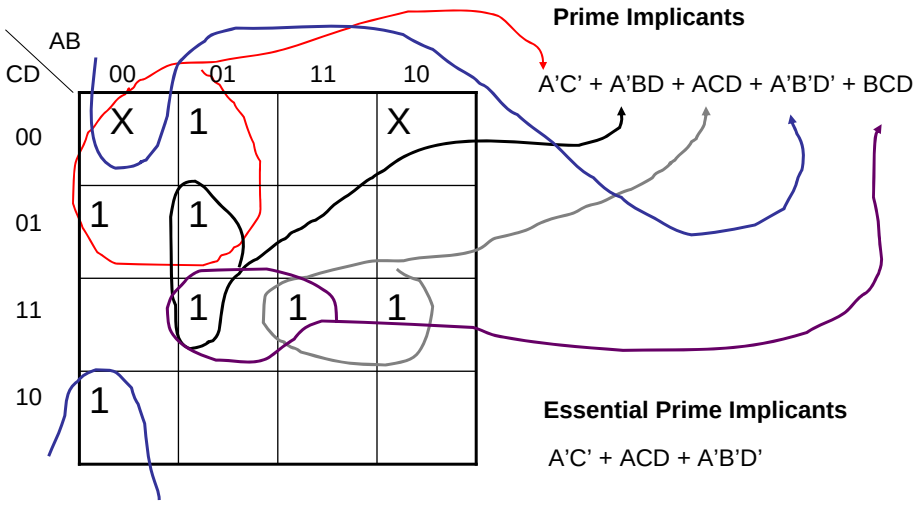
Essential prime implicant: if a minterm in the map is covered by only one prime implicant, this prime implicant is called an essential prime implicant.



Prime Implicant

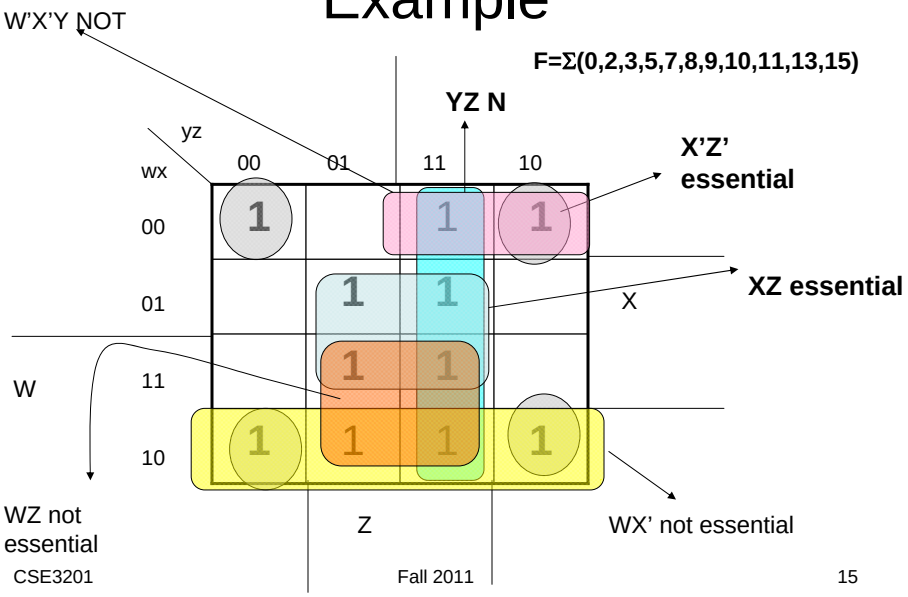
- The procedure of finding the simplified expression from the map is as follows:

1. First, determine all the essential prime implicants.
2. The simplified expression is obtained by combining all the essential prime implicants
3. After that add other prime implicants that may be needed to cover any remaining minterms that was not covered by essential prime implicants.



Minimum solution A'C' + ACD + A'B'D' + either (BCD or A'BD)

Example

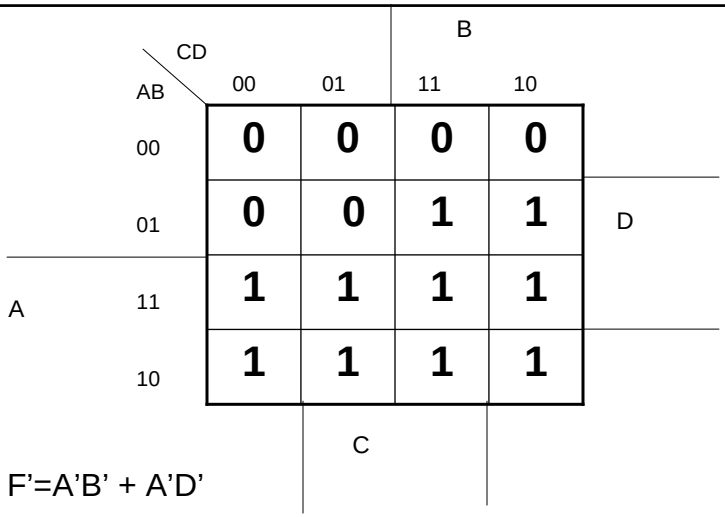


Prime Implicant

- In the previous map, there are 2 essential prime implicant (X'Z' the only way to cover m_0, and XZ the only way to cover m_5).
- Add other prime implicant to cover minterms m_3, m_9, and m_{11}.
- Minterm m_3 can be covered by either YZ or W'X'Y. minterm m_9 can be covered by either WZ or WX'. While minterm m_{11} is covered by any one of the 3 prime implicant.
- There are 4 possible way to describe this function, all of them include both XZ and X'Z' and we can add (YZ+WX), (YZ+WX'), (WX'+W'X'Y), or (WZ+W'X'Y').

Product of Sum Simplification

- F=A+DB+C'A
- Using maxterm
- F=(A+D)(A+B)
- F=A+AB+DA+DB
- O.K.



F'=A'B' + A'D'

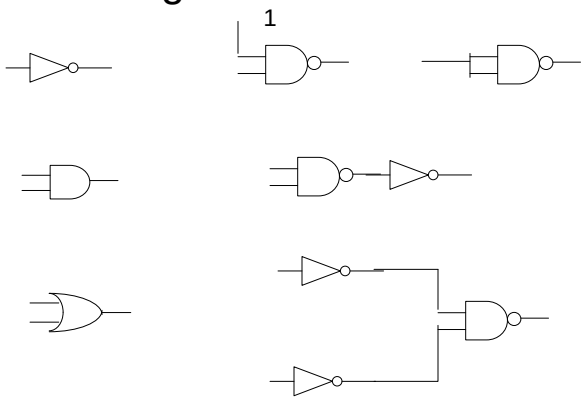
$\overline{F} = \overline{A'B' + A'D'}$

$F = \overline{(A'B')}\ \overline{(A'D')}$

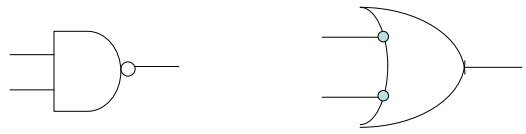
F =(A+B) (A+D)

NAD and NOR Implementation

- Universal gate



NAND Gate



AND-invert

Invert-OR

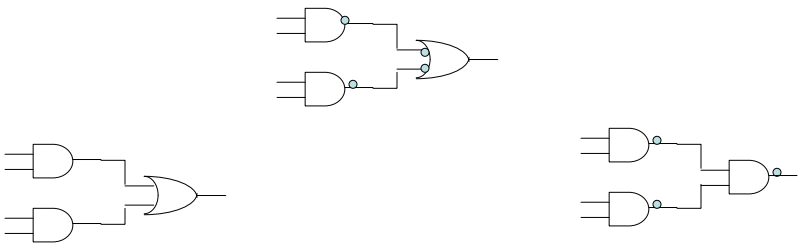
2 graphic symbols for NAND gate

NAND Implementation

- Express the function in sum of products
- Replace every AND by a NAND
- Replace the OR by Invert-OR
- If a single element is an input to the OR invert it.
- Change invert-OR to AND-invert
- Example on 3 variables

2-Level Implementation

- Start with sum of product F=AB+CD



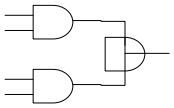
Multilevel NAND Circuits

- Convert all AND gates to NAND gates with AND-invert symbols
- Convert all OR gates to NAND gates with invert-OR symbols
- Check all the bubbles in the diagram, for every bubble that is not compensated by another bubble on the same line, add an inverter (or compliment the input literal)

NOR Implementation

Wired Logic

- Wired AND in open collector TTL
- Wired-OR in ECL gates



Wired-AND in TTL

AND-OR-INVERT AOI

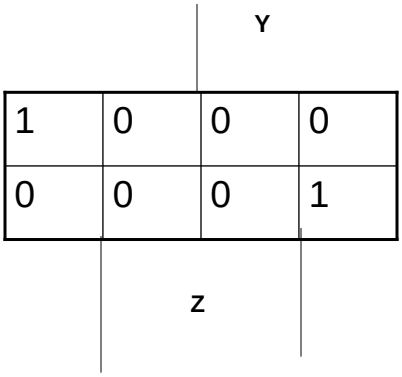
- In CMOS, and in most other logic families, the simplest gates are inverters, then NAND and NOR gates.
- It is typically not possible to design a non-inverting gate with less transistors than an inverting gate.
- CMOS circuits can perform two level of logic with just a single level of transistors. (AOI gate).
- The speed an other electrical characteristics of a CMOS AOI or OAI gate is quite comparable to those of a single CMOS NAND or NOR.

AOI Gates

- If you implement the complement of the function in sum of products it results in AOI circuit

Other 2-level Implementation

- Consider the function F=x'y'z'+zyz'
- Take the complement F'=x'y+xy'+z
- You can implement it as AOI using F'
- Change it to NAND-AND by moving the bubble from the output of the OR to its inputs (and changing it to AND) NAND-AND implementation

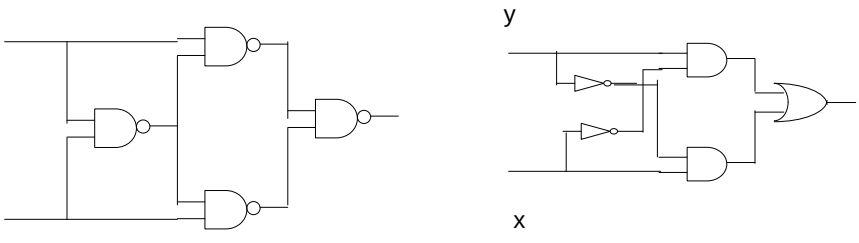


Other 2-level Implementation

- Using product of sums
- F=z'(x+y')(x'+y), OR we can say
- F'=[(x'+y'+z)(x+y+z)]
- We can implement the above equation using OR and NAND (OR-NAND implementation).
- Then we can move the bubble of the NAND to its inputs and changing it to OR (NOR-OR implementation)

EX-OR

- x⊕y=x'y+y'x



- 3 and 4 inout EX-OR Parity

HDL

- Can be used to represent logic diagram, Boolean expressions, and finite state machine representation.
- Used to document digital systems
- Used in simulation and synthesis
- 2 main languages, Verilog, and VHDL

Verilog

- C-like syntax
- Case sensitive, // for comments

```
module smpl_circuit(A,B,C,x,y);
   input A,B,C;
   output x,y;
   wire e;
   and g1(e,A,B);
   not g2(y,c);
   or g3(x,e,y);
endmodule
```


Verilog

- We can introduce delay

```
module smpl_circuit(A,B,C,x,y);
   input A,B,C;
   output x,y;
   wire e;
   and #(30) g1(e,A,B);
   not #(20) g2(y,c);
   or #(10) g3(x,e,y);
endmodule
```

Verilog

```
// Behavioral Model of a Nand gate
// By Dan Hyde, August 9, 1995
module NAND(in1, in2, out);
input in1, in2;
output out;
// continuous assign statement
assign out = ~(in1 & in2);
endmodule
```

- The continuous assignment statement is used to model **combinational circuits** where the outputs change when one wiggles the input.

Verilog

```
module AND(in1, in2, out);
// Structural model of AND gate from two NANDS
input in1, in2;
output out;
wire w1;
// two instantiations of the module NAND
NAND NAND1(in1, in2, w1);
NAND NAND2(w1, w1, out);
endmodule
```

Testing

```
module test_AND;
// High level module to test the two other modules
reg a, b;
wire x,y;
Circuit_with_delay cwd(A,B,C,x,y);
initial
    begin  // Test data
    A=1'b0; B=1'b0; C=1'b0;
    #100  A=1'b1;  B=1'b1;  C=1'b1;
    #100 $finish;
    end
endmodule
Module circuit_with_delay(A,B,C,x,y);
     input A,B,C;
     output x,y;
     wire e;
     and #(30) g1(e,A,B);
     not #(20) g2(y,c);
     or #(10) g3(x,e,y);
endmodule
```


User Defined Primitives

```
//User defined primitive(UDP)
primitive crctp(x,A,B,C);
    output x;
    input A,B,C;
//Now the truth table
    table
//  A     B         C         :         x
    0     0         0         :         1;
    0     0         1         :         0;
    0     1         0         :         1;
    0     1         1         :         0;
    1     0         0         :         1;
    1     0         1         :         0;
    1     1         0         :         1;
    1     1         1         :         1;
    endtable
Endprimitive
module abcdef;
    reg x,y,z;
    wire w;
    crctp(w,x,y,z);
endmodule
```