CMSC 313 Lecture 19

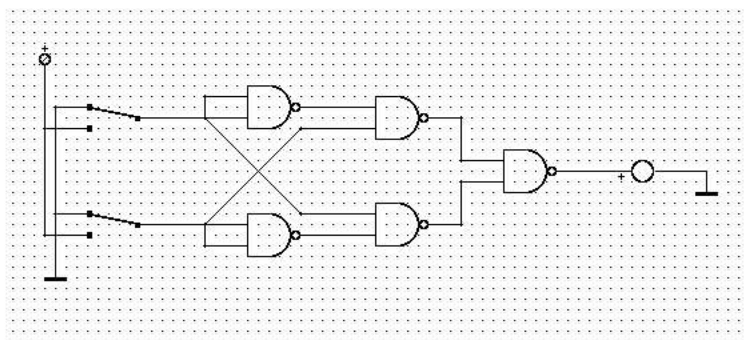
- Homework 4 Questions
- Combinational Logic Components
- Programmable Logic Arrays
- Introduction to Circuit Simplification

Due: Thursday, November 6, 2003

1. (10 points) Question 3.8, page 96, Murdocca & Heuring
2. (10 points) Question 3.9, page 96, Murdocca & Heuring
3. (20 points) Read the instructions on how to run the DigSim digital simulator on the course web page:

   `http://www.csee.umbc.edu/~chang/cs313.f03/digsim-info.shtml`

   Using DigSim, wire up the following circuit diagram, play with the switches, create a text box with your name, and save the circuit diagram. (This is the same as the circuit we used in the in-class lab.)

   The file which has your circuit diagram should be a plain text file that starts with something like:

   ```
   # Digsim file
   version 1 0
   describe component TwoNandPort
    pos 23 13
   ```

   Use a text editor to look at the file and make sure that the file is not empty and has some data similar to the above. Next, use DigSim to load the file and make sure that this still works. If all is well, submit the circuit file using the Unix `submit` command as in previous assignments. The submission name for this assignment is: `digsim0`.The UNIX command to do this should look something like:

   ```
   submit cs313_0101 digsim0 xor.sim
   ```

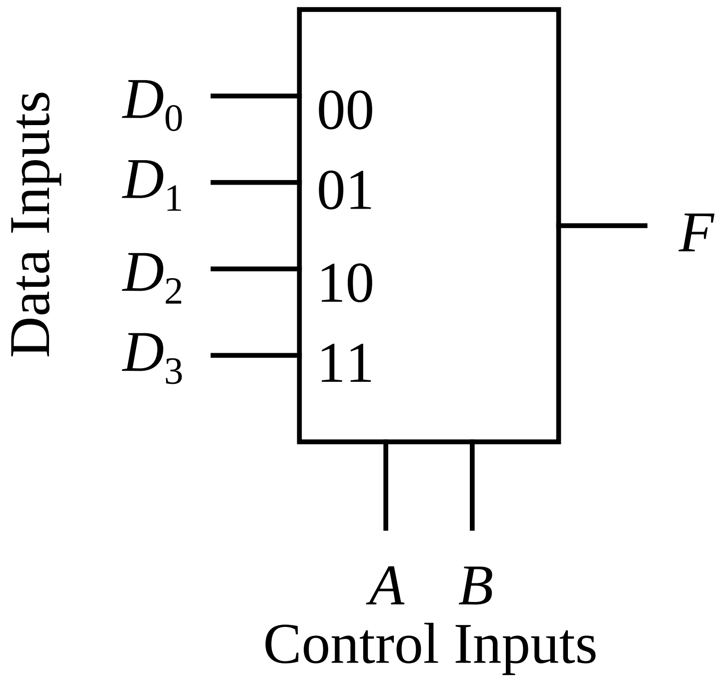
Last Time & Before

- In-class lab: next time Tuesday 11/18
- Half adders & full adders
- Ripple carry adders vs carry lookahead adders
- Propagation delay

Digital Components

- **High level digital circuit designs are normally created using collections of logic gates referred to as *components*, rather than using individual logic gates.**
- **Levels of integration (numbers of gates) in an integrated circuit (IC) can roughly be considered as:**
  - **Small scale integration (SSI): 10-100 gates.**
  - **Medium scale integration (MSI): 100 to 1000 gates.**
  - **Large scale integration (LSI): 1000-10,000 logic gates.**
  - **Very large scale integration (VLSI): 10,000-upward logic gates.**
  - **These levels are approximate, but the distinctions are useful in comparing the relative complexity of circuits.**

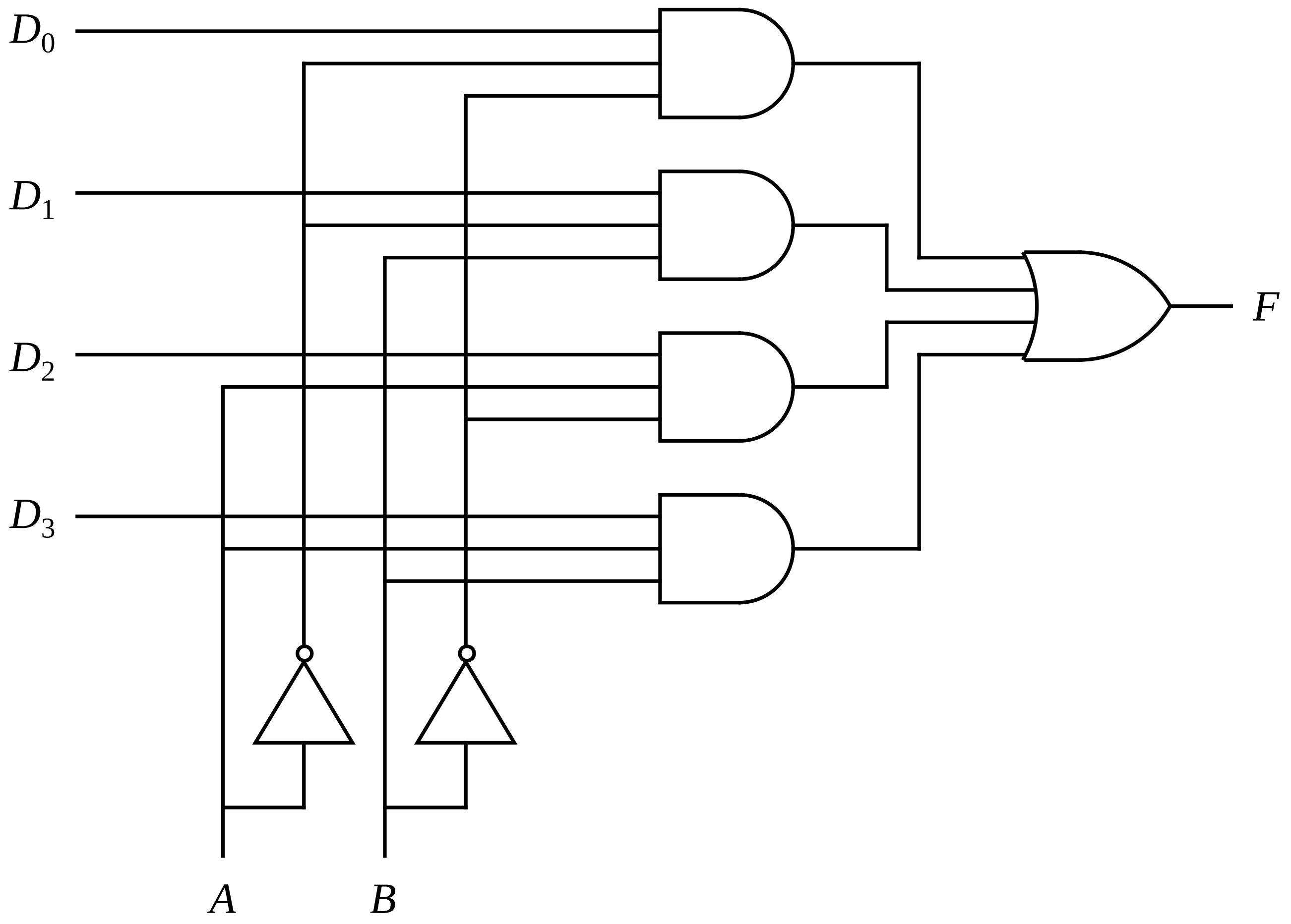
Multiplexer



A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$$F = \overline{A}\,\overline{B}\,D_0 + \overline{A}\,B\,D_1 + A\,\overline{B}\,D_2 + A\,B\,D_3$$

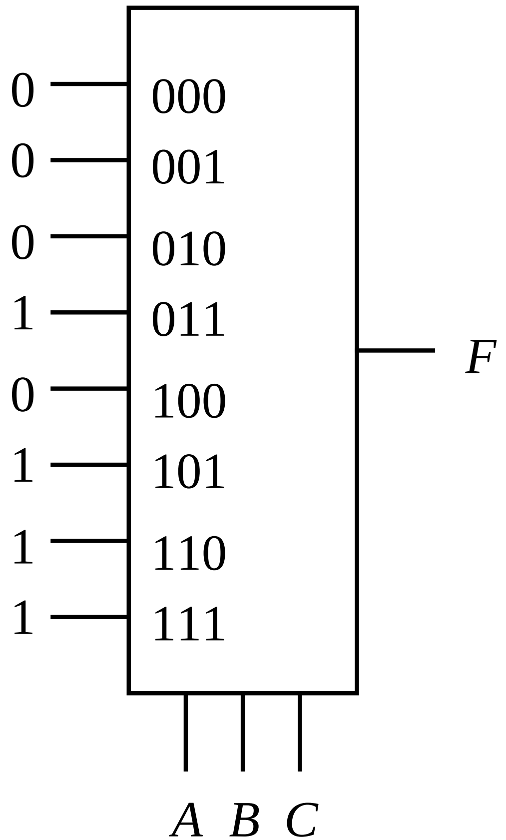
AND-OR Implementation of MUX



MUX Implementation of Majority

- **Principle: Use the 3 MUX control inputs to select (one at a time) the 8 data inputs.**

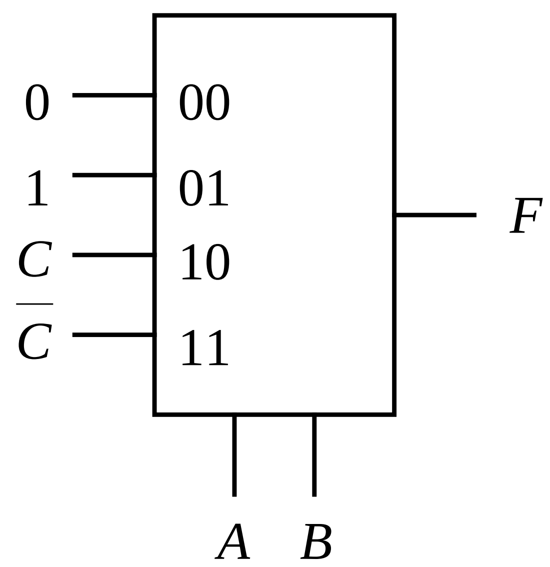
A	*B*	*C*	*M*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



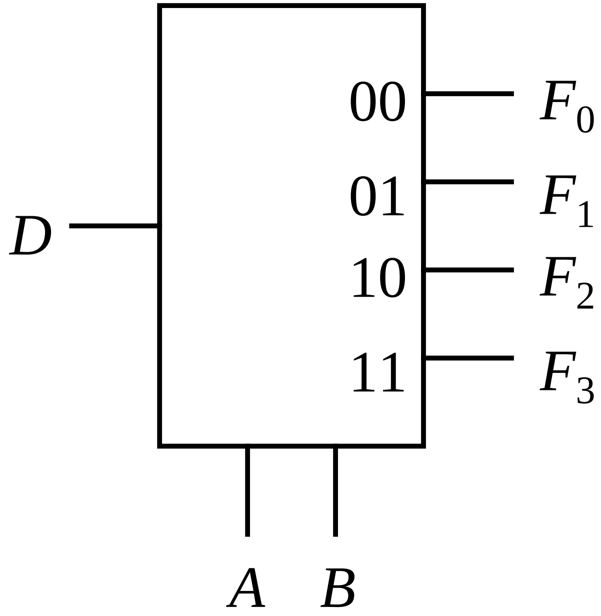
4-to-1 MUX Implements 3-Var Function

- **Principle: Use the A and B inputs to select a pair of minterms. The value applied to the MUX data input is selected from {0, 1, C, $\overline{C}$} to achieve the desired behavior of the minterm pair.**

A	B	C	F	
0	0	0	0	0
0	0	1	0	
0	1	0	1	1
0	1	1	1	
1	0	0	0	C
1	0	1	1	
1	1	0	1	$\overline{C}$
1	1	1	0	



Demultiplexer

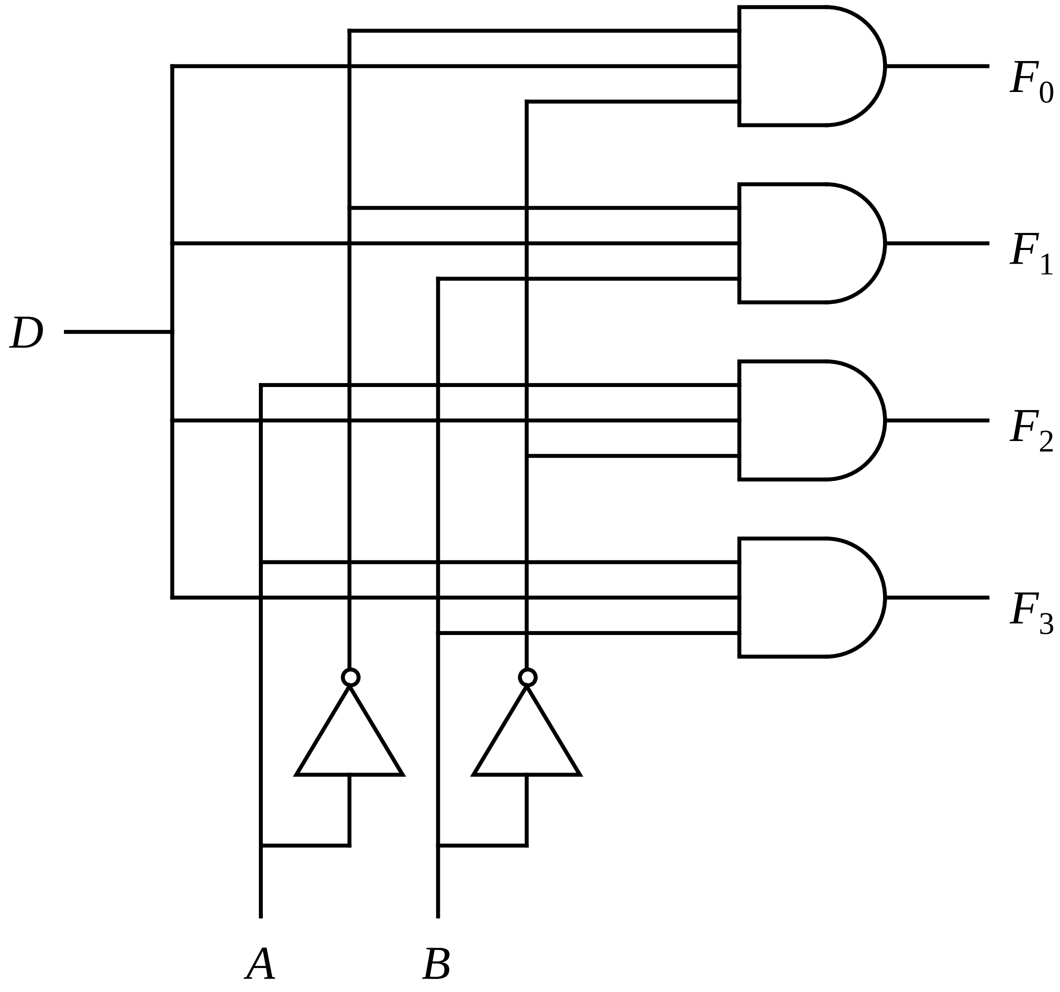


D	A	B	F_0	F_1	F_2	F_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

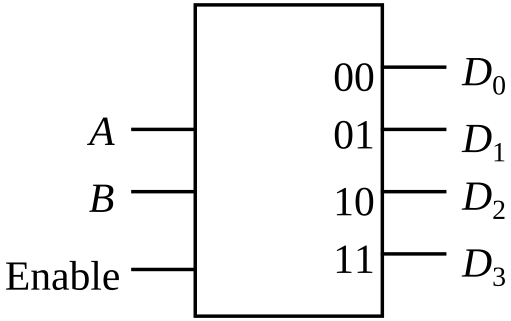
$$F_0 = D\overline{A}\,\overline{B} \qquad F_2 = DA\overline{B}$$

$$F_1 = D\overline{A}B \qquad F_3 = DAB$$

Gate-Level Implementation of DEMUX



Decoder

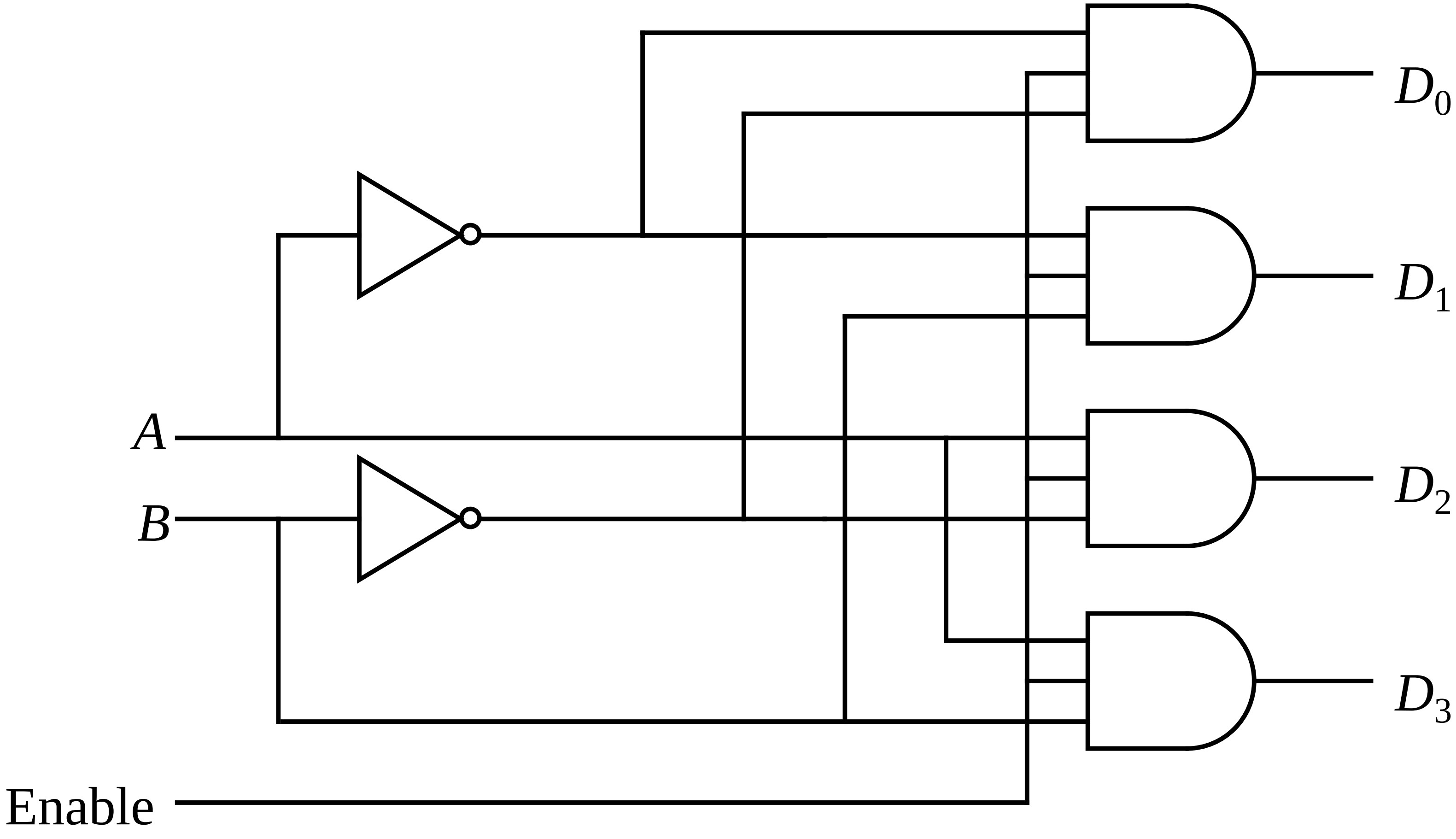


Enable = 1					
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Enable = 0					
A	B	D_0	D_1	D_2	D_3
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0

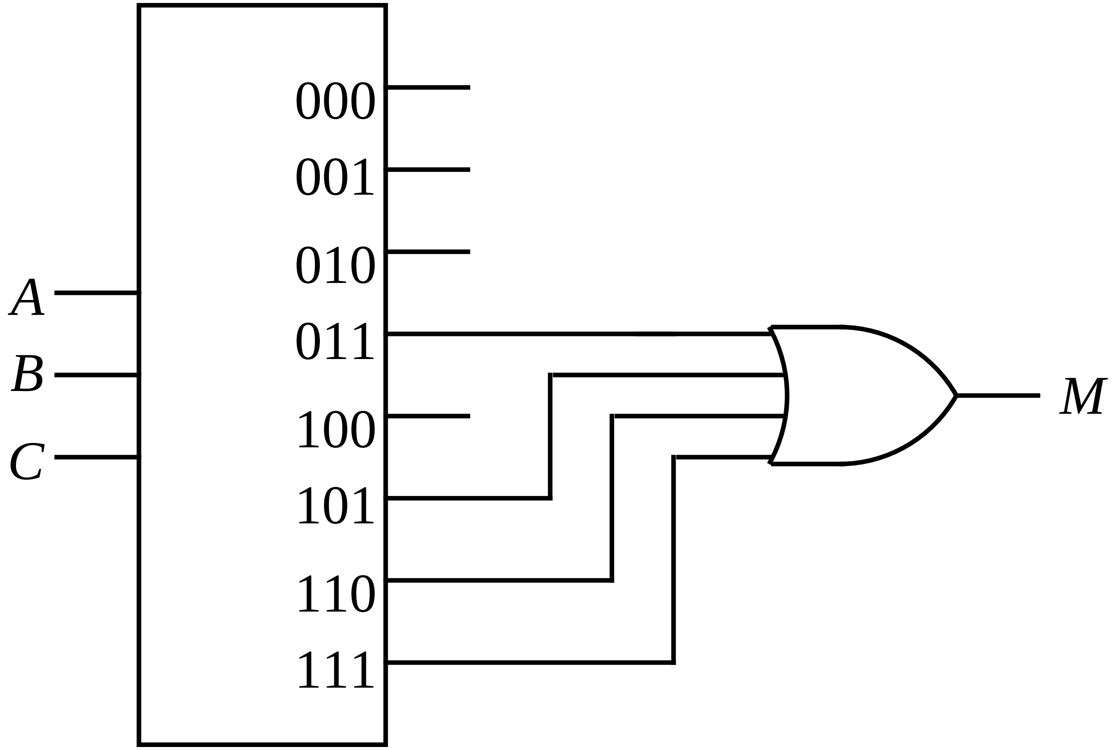
$D_0 = \overline{A}\,\overline{B}$ $D_1 = \overline{A}\,B$ $D_2 = A\,\overline{B}$ $D_3 = A\,B$

Gate-Level Implementation of Decoder



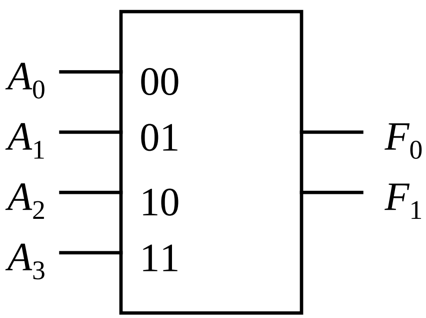
Decoder Implementation of Majority Function

- **Note that the enable input is not always present. We use it when discussing decoders for memory.**



Priority Encoder

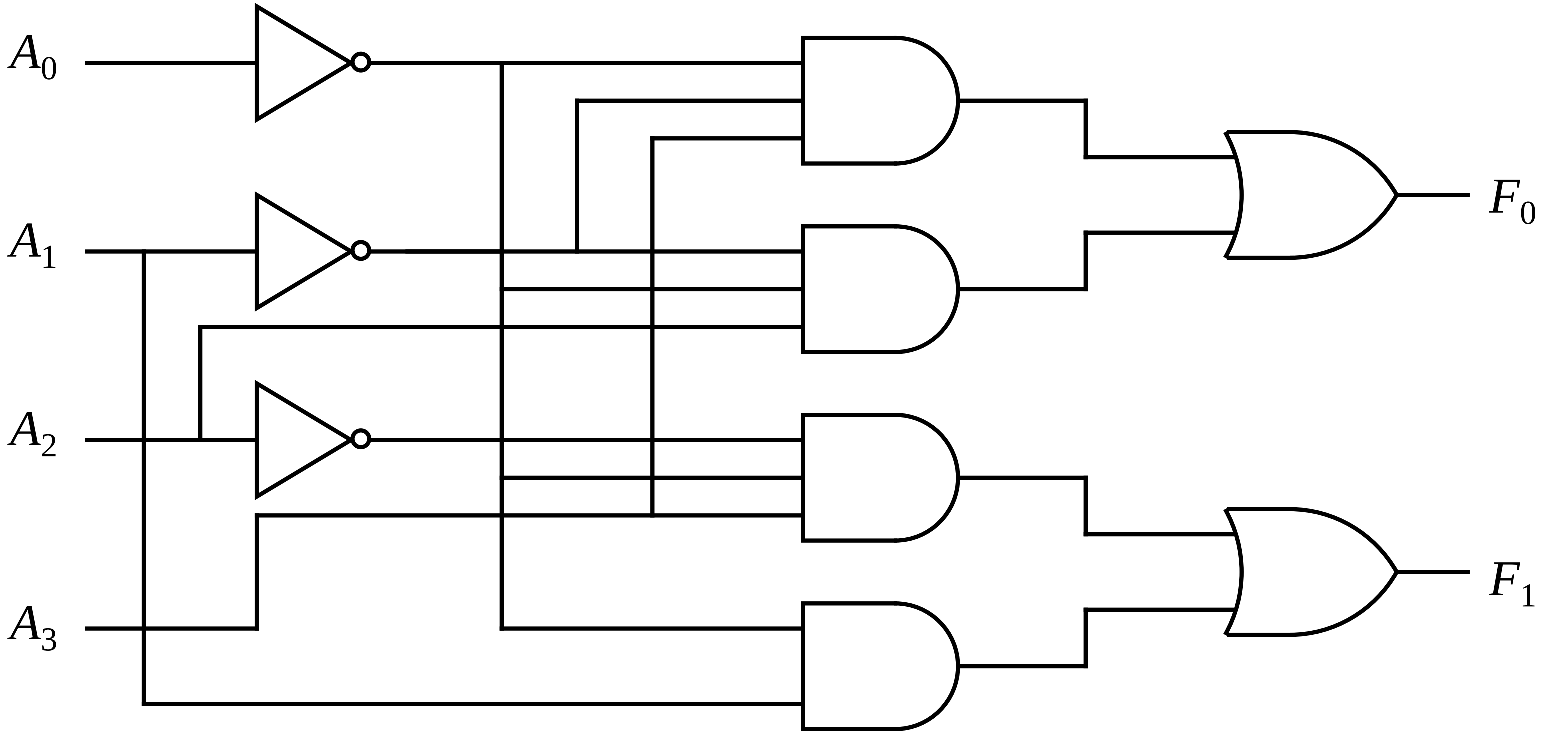
- An encoder translates a set of inputs into a binary encoding.
- Can be thought of as the converse of a decoder.
- A priority encoder imposes an order on the inputs.
- A_i has a higher priority than A_{i+1}



$$F_0 = \overline{A_0}\,\overline{A_1}\,A_3 + \overline{A_0}\,\overline{A_1}\,A_2$$
$$F_1 = \overline{A_0}\,\overline{A_2}\,A_3 + \overline{A_0}\,A_1$$

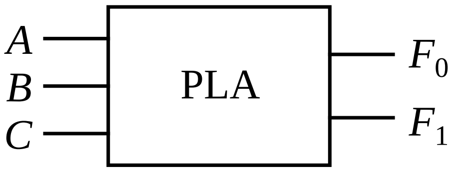
A_0	A_1	A_2	A_3	F_0	F_1
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

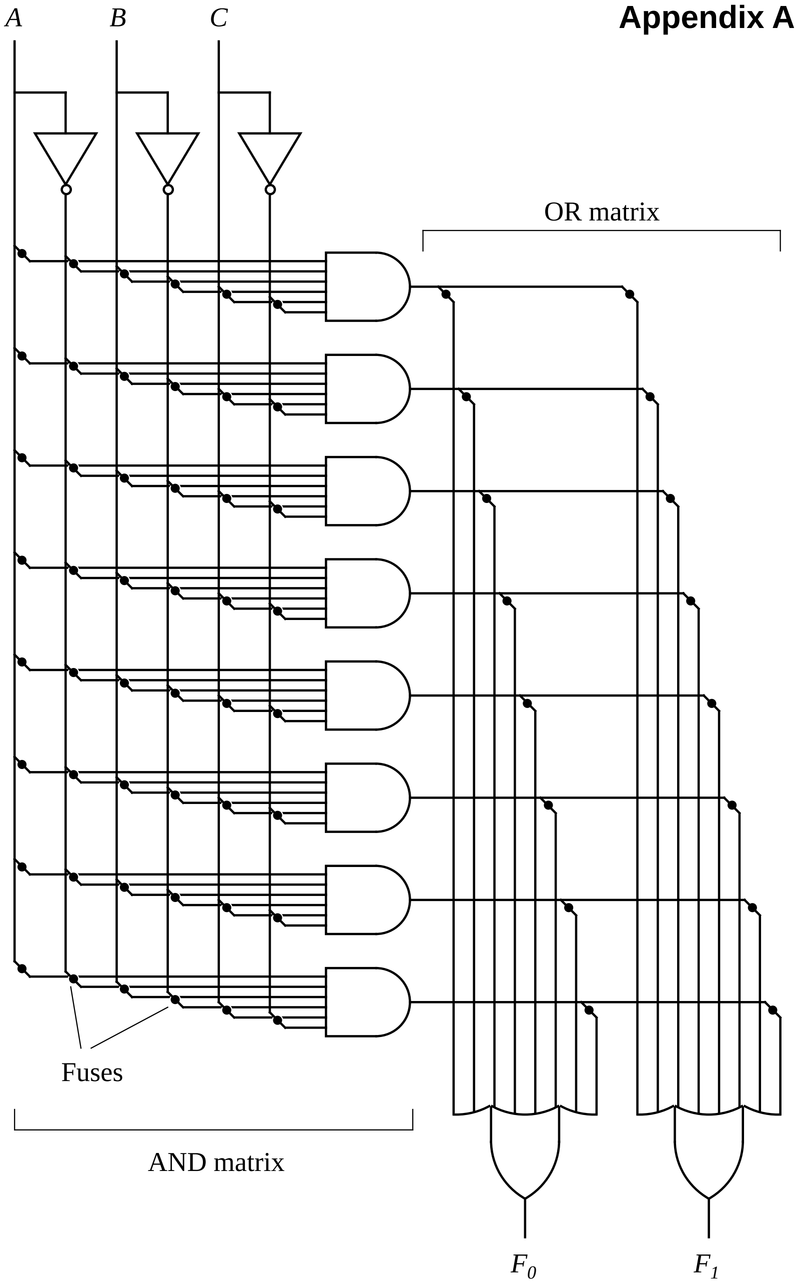
AND-OR Implementation of Priority Encoder



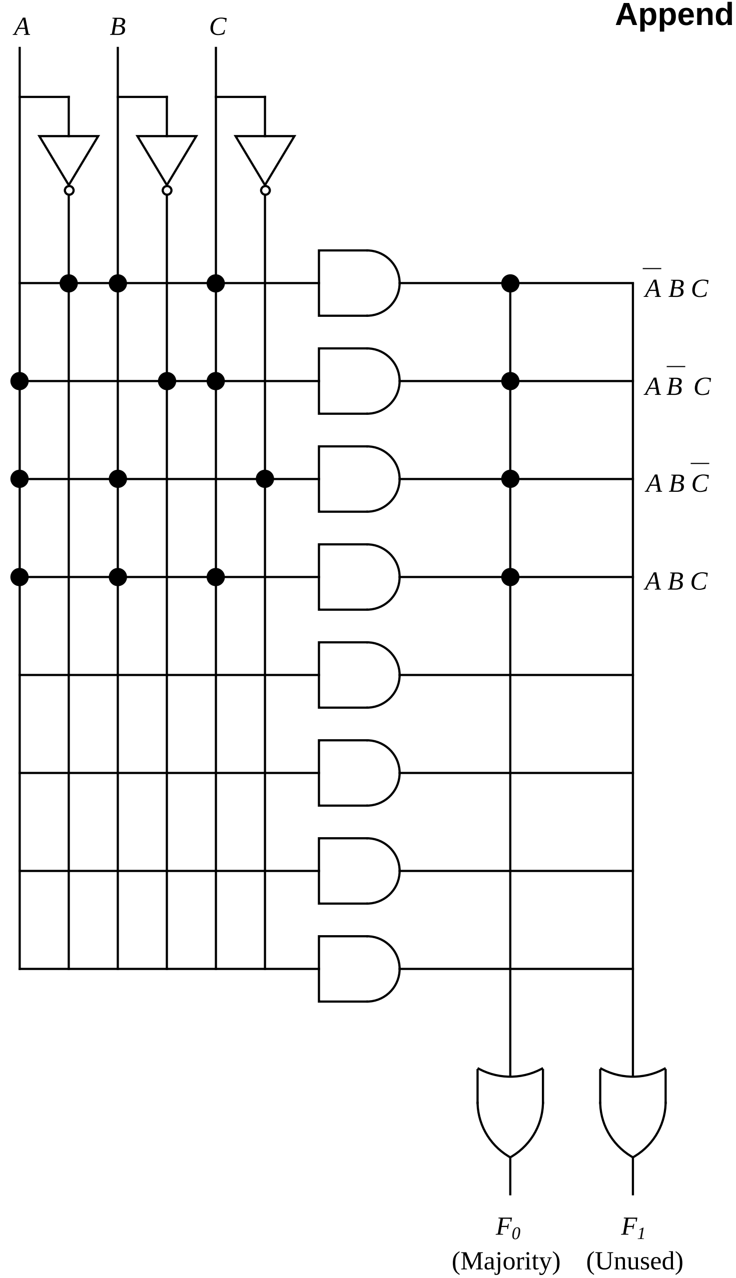
Programmable Logic Array

- A PLA is a customizable AND matrix followed by a customizable OR matrix.
- Black box view of PLA:



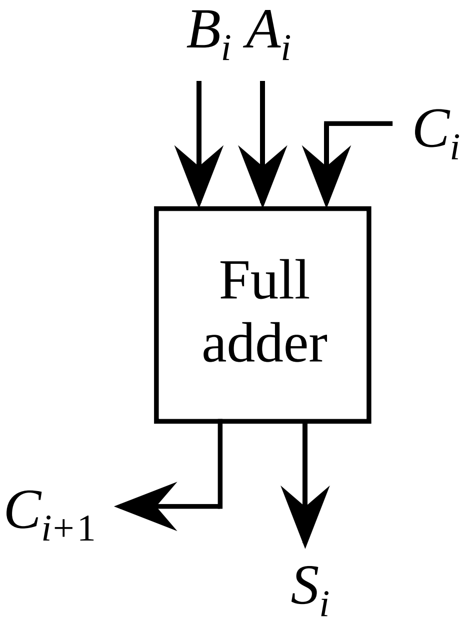


Simplified Representation of PLA Implementation of Majority Function

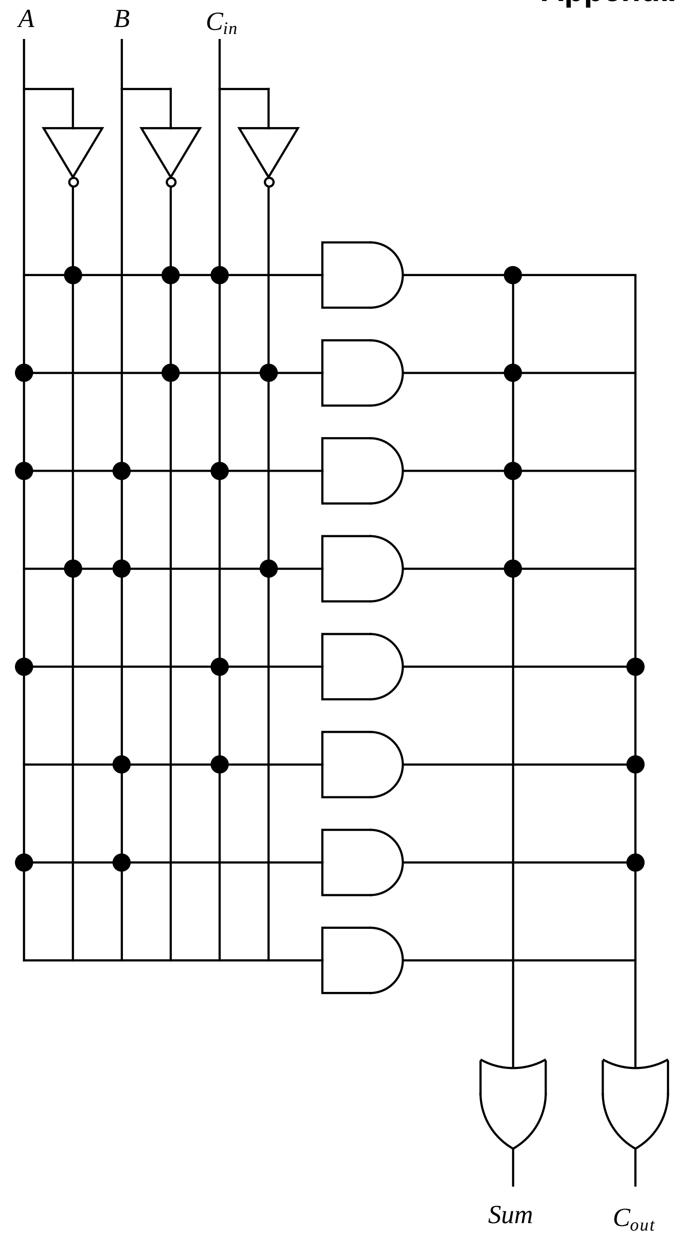


Full Adder

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



PLA Realization of Full Adder

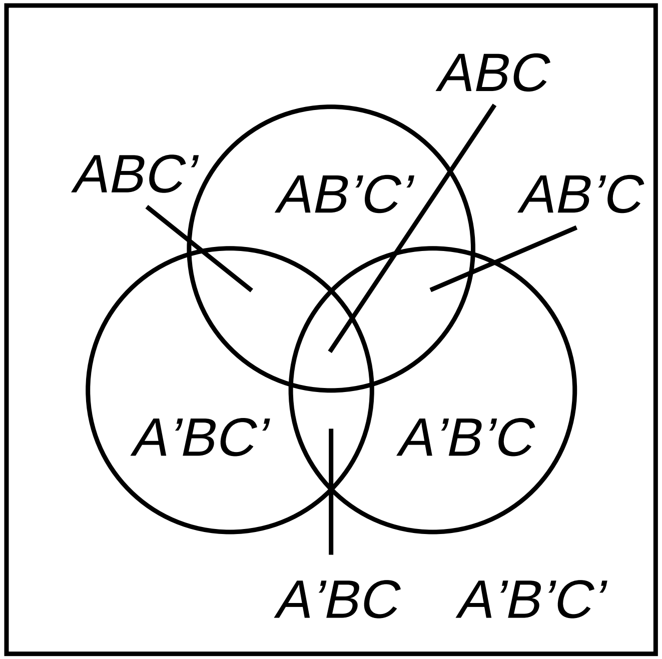


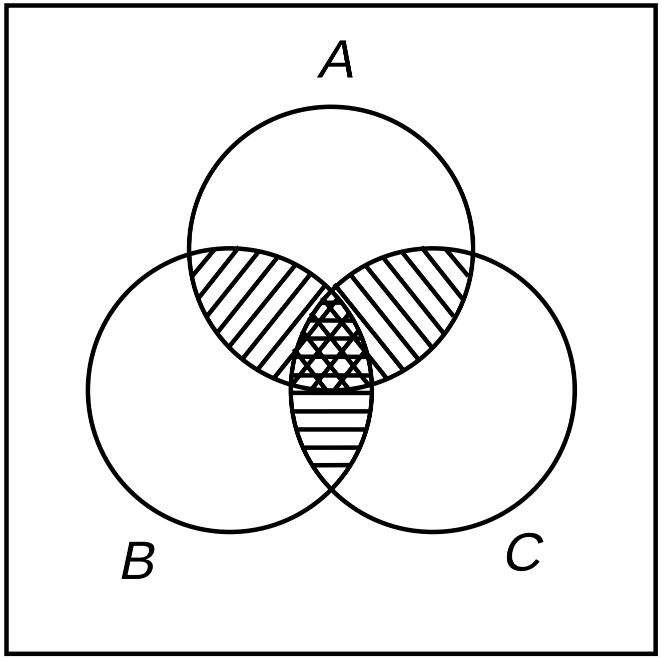
Reduction (Simplification) of Boolean Expressions

- It is usually possible to simplify the canonical SOP (or POS) forms.
- A smaller Boolean equation generally translates to a lower gate count in the target circuit.
- We cover three methods: algebraic reduction, Karnaugh map reduction, and tabular (Quine-McCluskey) reduction.

Karnaugh Maps: Venn Diagram Representation of Majority Function

- **Each distinct region in the “Universe” represents a minterm.**
- **This diagram can be transformed into a *Karnaugh Map*.**





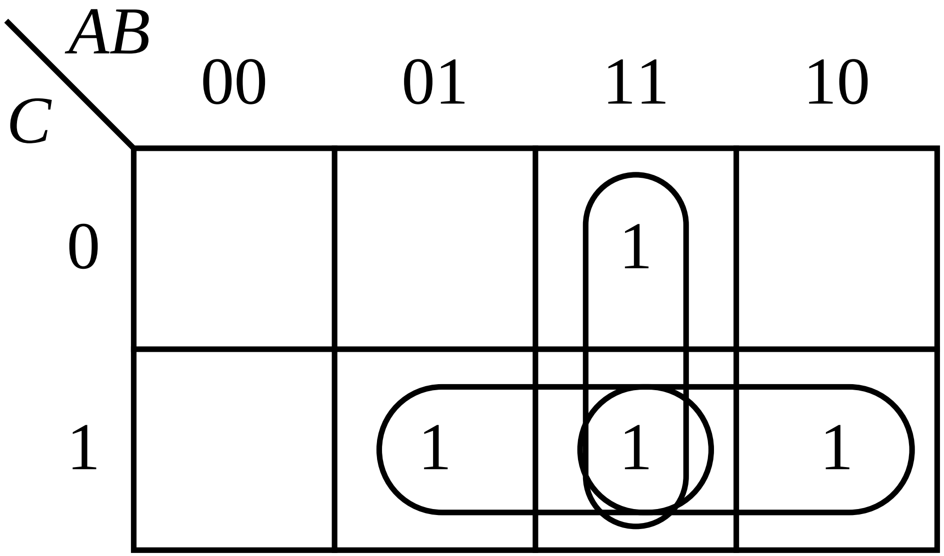
K-Map for Majority Function

- Place a “1” in each cell that corresponds to that minterm.
- Cells on the outer edge of the map “wrap around”

Minterm Index	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

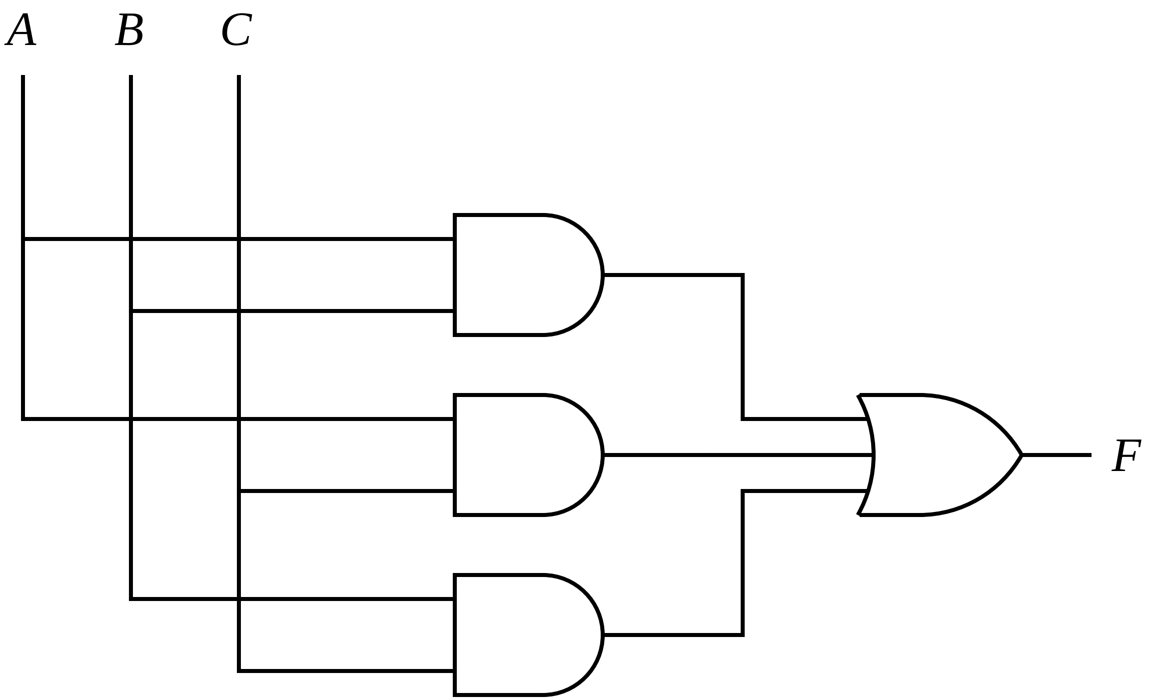
C \ AB	00	01	11	10
0			1	
1		1	1	1

Adjacency Groupings for Majority Function



- **F = BC + AC + AB**

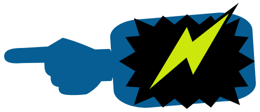
Minimized AND-OR Majority Circuit

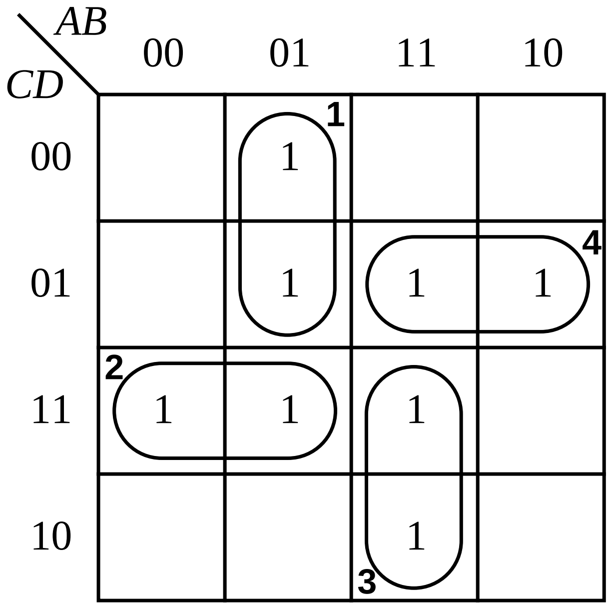


- F = BC + AC + AB
- The K-map approach yields the same minimal two-level form as the algebraic approach.

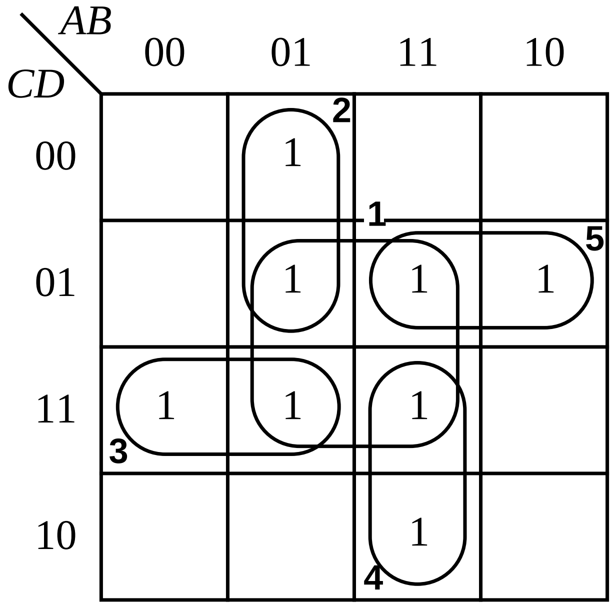
K-Map Groupings

- **Minimal grouping is on the left, non-minimal (but logically equivalent) grouping is on the right.**
- **To obtain minimal grouping, create *smallest* groups first.**



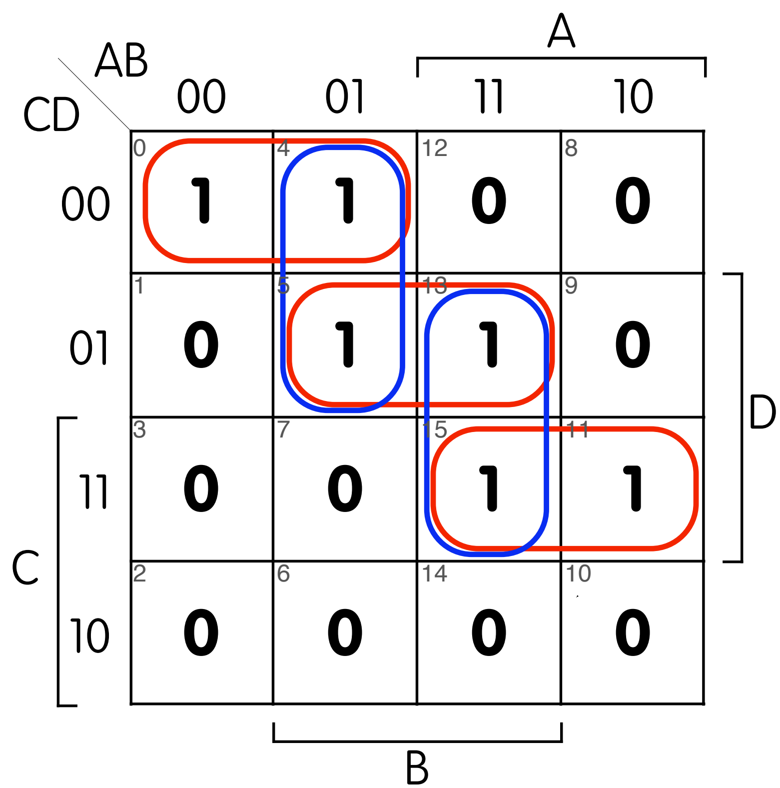


$$F = \overline{A}B\overline{C} + \overline{A}CD + ABC + A\overline{C}D$$

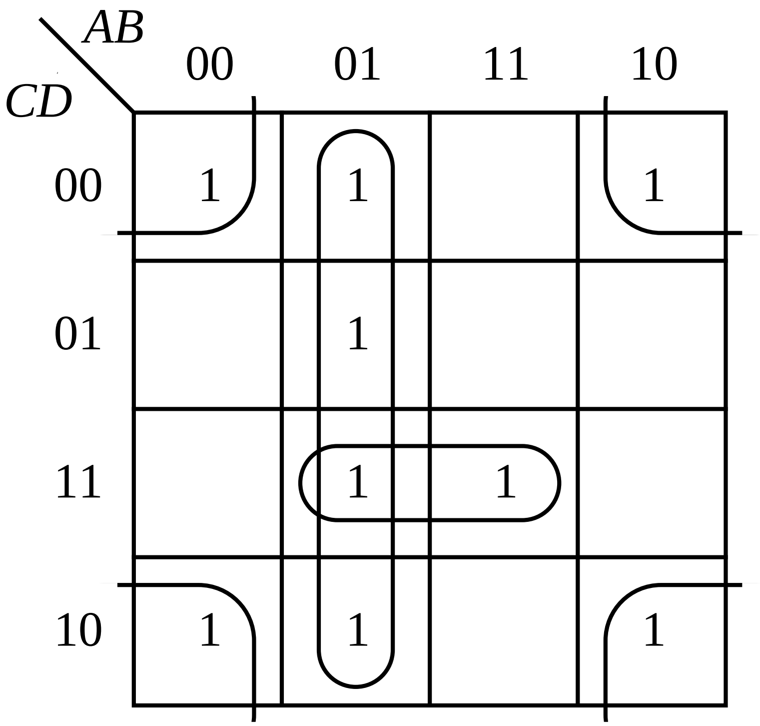


$$F = BD + \overline{A}B\overline{C} + \overline{A}CD + ABC + A\overline{C}D$$

Example Requiring More Rules



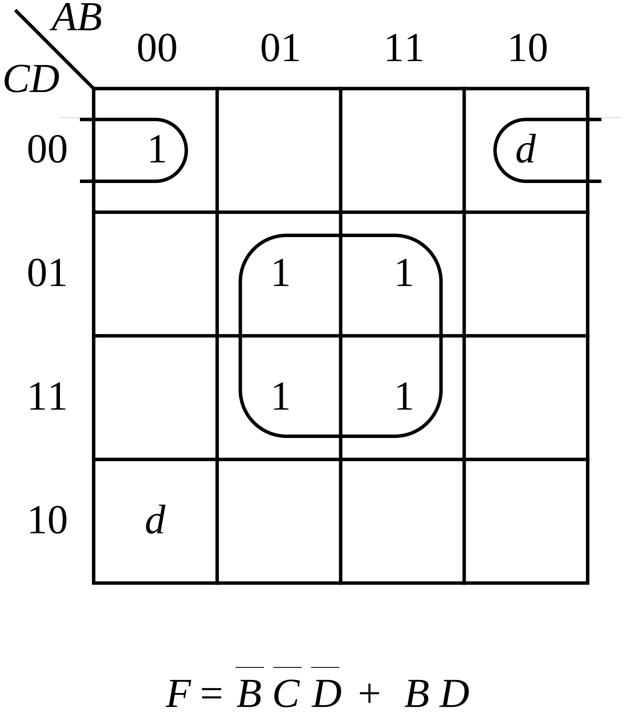
K-Map Corners are Logically Adjacent

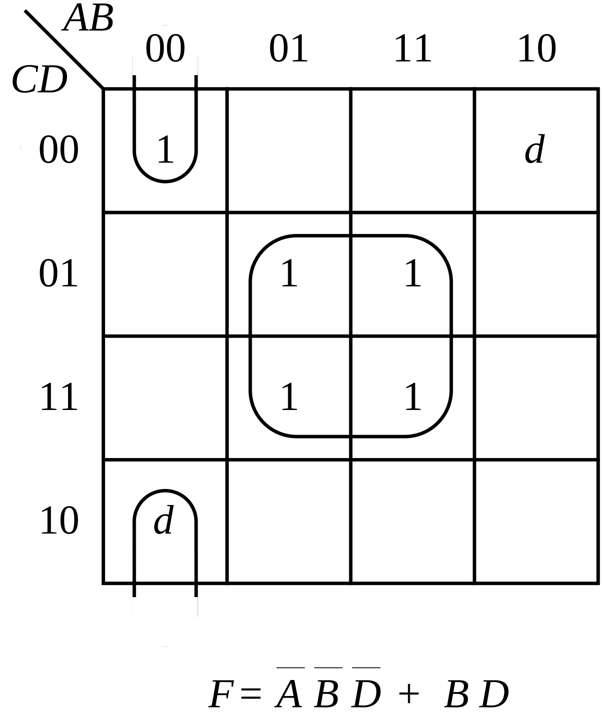


$$F = B\,C\,D + \overline{B}\,\overline{D} + \overline{A}\,B$$

K-Maps and Don’t Cares

- **There can be more than one minimal grouping, as a result of don’t cares.**

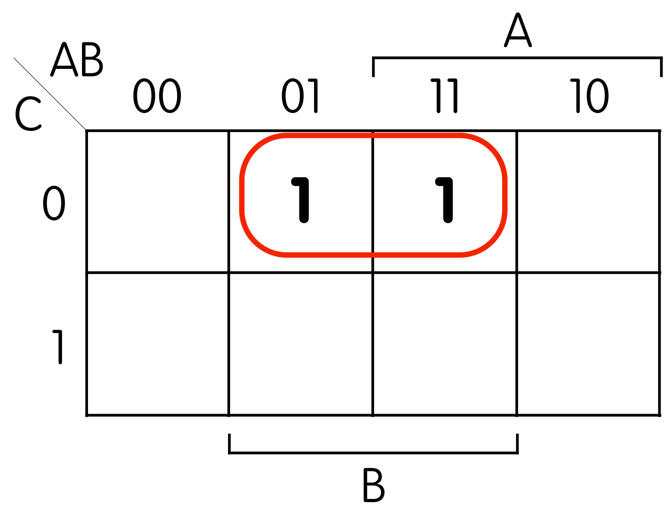




Gray Code

- Two bits: 00, 01, 11, 10
- Three bits: 000, 001, 011, 010, 110, 111, 101, 100
- Successive bit patterns only differ at 1 position
- For Karnaugh maps, adjacent 1's represent minterms that can be simplified using the rule:

$$ABC' + A'BC' = (A + A')BC' = 1\,BC' = BC'$$



Karnaugh Maps

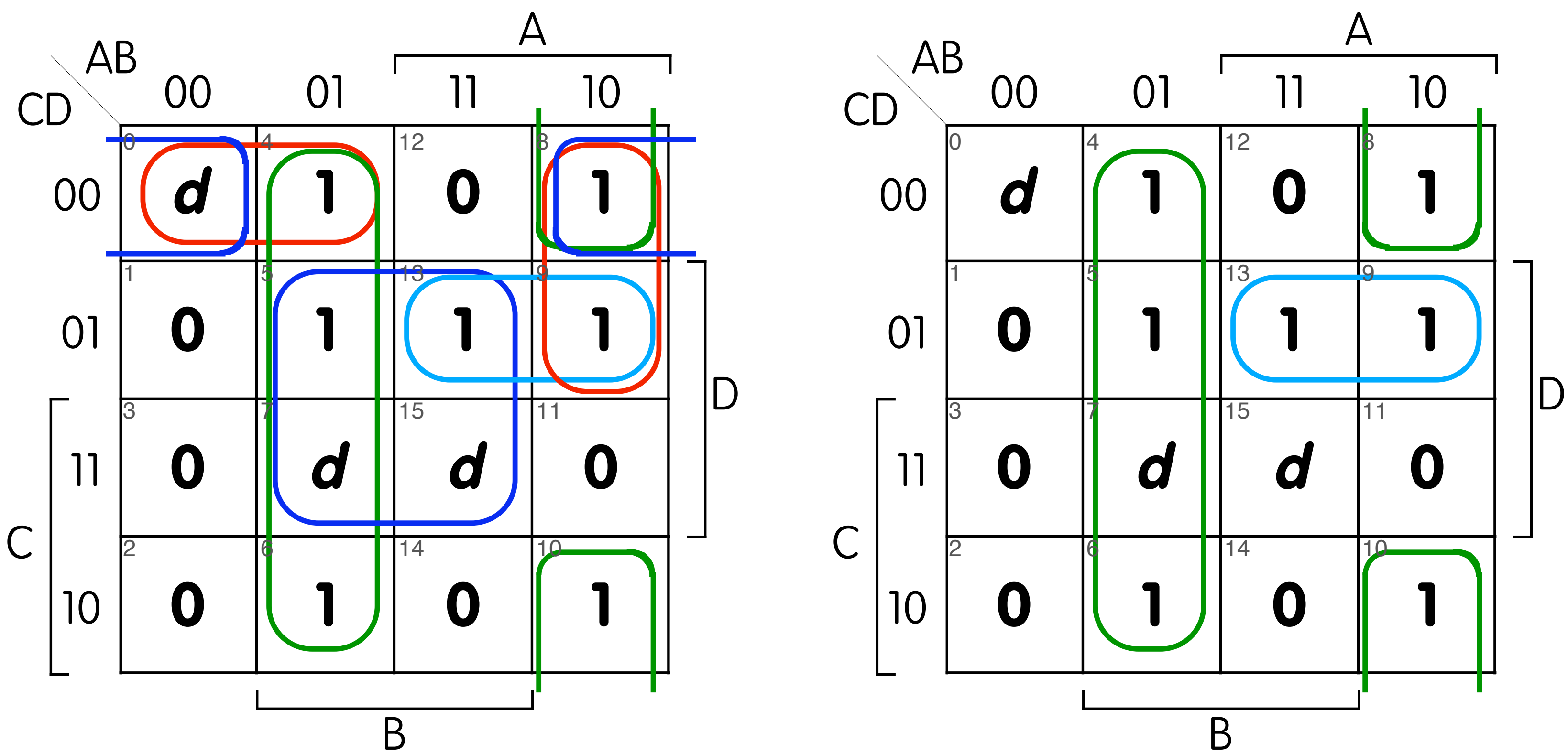
- **Implicant:** rectangle with 1, 2, 4, 8, 16 ... 1's
- **Prime Implicant:** an implicant that cannot be extended into a larger implicant
- **Essential Prime Implicant:** the only prime implicant that covers some 1
- **K-map Algorithm (not from M&H):**

  1. Find ALL the prime implicants. Be sure to check every 1 and to use don't cares.

  2. Include all essential prime implicants.

  3. Try all possibilities to find the minimum cover for the remaining 1's.

K-map Example



A′B + AC′D + AB′D′

UMBC, CMSC313, Richard Chang <chang@umbc.edu>

Notes on K-maps

- Also works for POS
- Takes 2^n time for formulas with n variables
- Only optimizes two-level logic
  - Reduces number of terms, then number of literals in each term
- Assumes inverters are free
- Does not consider minimizations across functions
- Circuit minimization is generally a hard problem
- Quine-McCluskey can be used with more variables
- CAD tools are available if you are serious

Next Time

- Continue Circuit Simplification
- Homework 4 due
- Homework 5 assigned