Principles of Computer Architecture

Miles Murdocca and Vincent Heuring

Appendix A: Digital Logic

Chapter Contents

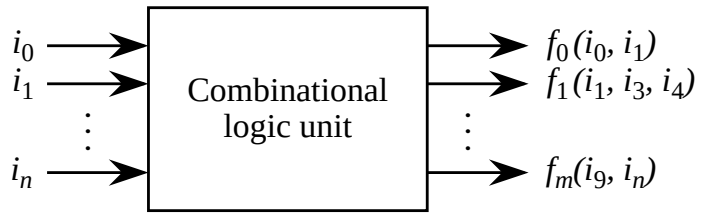
A.1 Introduction
A.2 Combinational Logic
A.3 Truth Tables
A.4 Logic Gates
A.5 Properties of Boolean Algebra
A.6 The Sum-of-Products Form, and Logic Diagrams
A.7 The Product-of-Sums Form
A.8 Positive vs. Negative Logic
A.9 The Data Sheet
A.10 Digital Components
A.11 Sequential Logic
A.12 Design of Finite State Machines
A.13 Mealy vs. Moore Machines
A.14 Registers
A.15 Counters

Some Definitions

- ***Combinational logic:*** **a digital logic circuit in which logical decisions are made based only on combinations of the inputs. *e.g.* an adder.**
- ***Sequential logic:*** **a circuit in which decisions are made based on combinations of the current inputs as well as the past history of inputs. *e.g.* a memory unit.**
- ***Finite state machine:*** **a circuit which has an internal state, and whose outputs are functions of both current inputs and its internal state. *e.g.* a vending machine controller.**

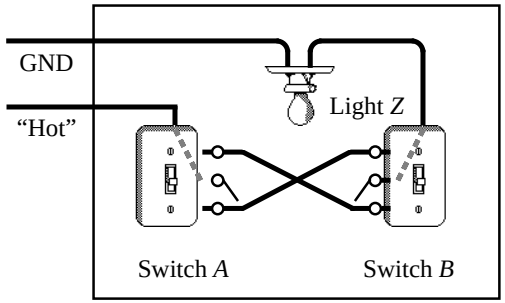
The Combinational Logic Unit

- **Translates a set of inputs into a set of outputs according to one or more mapping functions.**
- **Inputs and outputs for a CLU normally have two distinct (binary) values: high and low, 1 and 0, 0 and 1, or 5 V and 0 V for example.**
- **The outputs of a CLU are strictly functions of the inputs, and the outputs are updated immediately after the inputs change. A set of inputs $i_0 - i_n$ are presented to the CLU, which produces a set of outputs according to mapping functions $f_0 - f_m$.**



A Truth Table

- Developed in 1854 by George Boole.
- Further developed by Claude Shannon (Bell Labs).
- Outputs are computed for all possible input combinations (how many input combinations are there?)
- Consider a room with two light switches. How must they work?



Inputs		Output
A	*B*	*Z*
0	0	0
0	1	1
1	0	1
1	1	0

Alternate Assignment of Outputs to Switch Settings

- We can make the assignment of output values to input combinations any way that we want to achieve the desired input-output behavior.

Inputs		Output
A	*B*	*Z*
0	0	1
0	1	0
1	0	0
1	1	1

Truth Tables Showing All Possible Functions of Two Binary Variables

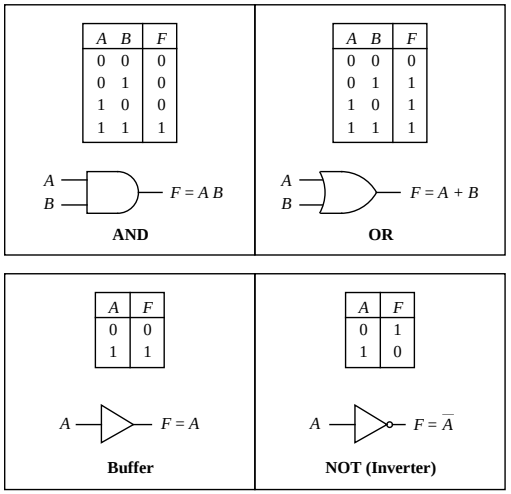
- The more frequently used functions have names: AND, XOR, OR, NOR, XOR, and NAND. (Always use upper case spelling.)

Inputs		Outputs							
A	*B*	*False*	*AND*	$A\overline{B}$	*A*	$\overline{A}B$	*B*	*XOR*	*OR*
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Inputs		Outputs							
A	*B*	*NOR*	*XNOR*	$\overline{B}$	$A + \overline{B}$	$\overline{A}$	$\overline{A} + B$	*NAND*	*True*
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

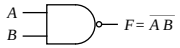
Logic Gates and Their Symbols

- Logic symbols shown for AND, OR, buffer, and NOT Boolean functions.
- Note the use of the "inversion bubble."
- (Be careful about the "nose" of the gate when drawing AND vs. OR.)



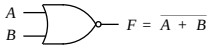
Logic Gates and their Symbols (cont')

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



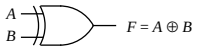
NAND

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



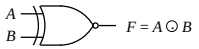
NOR

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



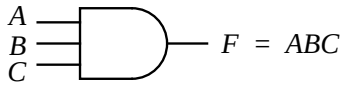
Exclusive-OR (XOR)

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

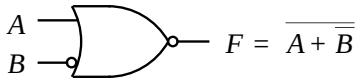


Exclusive-NOR (XNOR)

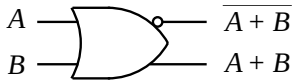
Variations of Logic Gate Symbols



(a)



(b)



(c)

(a) 3 inputs **(b) A Negated input** **(c) Complementary outputs**

Properties of Boolean Algebra

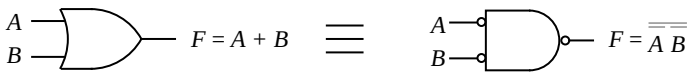
- **Principle of duality: The dual of a Boolean function is obtained by replacing AND with OR and OR with AND, 1s with 0s, and 0s with 1s.**

	Relationship	Dual	Property
Postulates	$A\,B = B\,A$	$A + B = B + A$	Commutative
	$A\,(B + C) = A\,B + A\,C$	$A + B\,C = (A + B)\,(A + C)$	Distributive
	$1\,A = A$	$0 + A = A$	Identity
	$A\,\overline{A} = 0$	$A + \overline{A} = 1$	Complement
Theorems	$0\,A = 0$	$1 + A = 1$	Zero and one theorems
	$A\,A = A$	$A + A = A$	Idempotence
	$A\,(B\,C) = (A\,B)\,C$	$A + (B + C) = (A + B) + C$	Associative
	$\overline{\overline{A}} = A$		Involution
	$\overline{A\,B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$	DeMorgan's Theorem
	$AB + \overline{A}C + BC = AB + \overline{A}C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$	Consensus Theorem
	$A\,(A + B) = A$	$A + A\,B = A$	Absorption Theorem

DeMorgan's Theorem

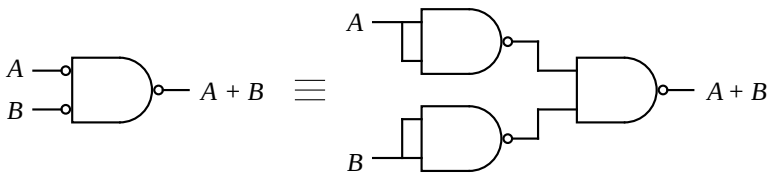
A	B	$\overline{A\,B} = \overline{A} + \overline{B}$		$\overline{A + B} = \overline{A}\,\overline{B}$	
0	0	1	1	1	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	0	0

DeMorgan's theorem: $A + B = \overline{\overline{A + B}} = \overline{\overline{A}\,\overline{B}}$



All-NAND Implementation of OR

- NAND alone implements all other Boolean logic gates.

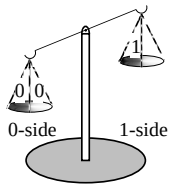


Sum-of-Products Form: The Majority Function

- The SOP form for the 3-input majority function is:

  $M = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = m3 + m5 + m6 + m7 = \Sigma\,(3, 5, 6, 7)$.
- Each of the 2^n terms are called ***minterms***, ranging from 0 to $2^n - 1$.
- Note relationship between minterm number and boolean value.

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



A balance tips to the left or right depending on whether there are more 0's or 1's.

AND-OR Implementation of Majority

- Gate count is 8, gate input count is 19.

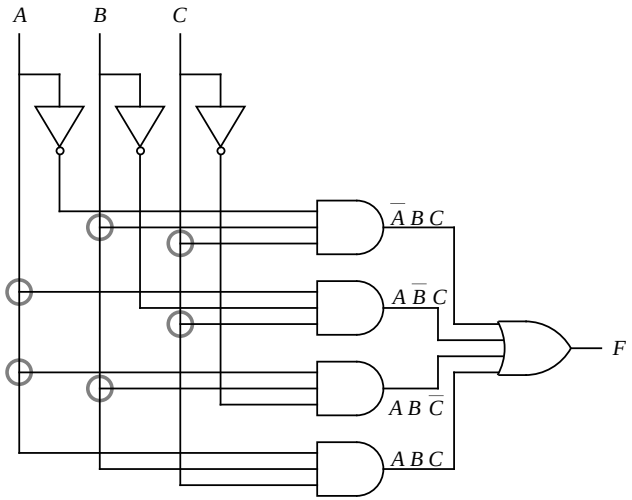
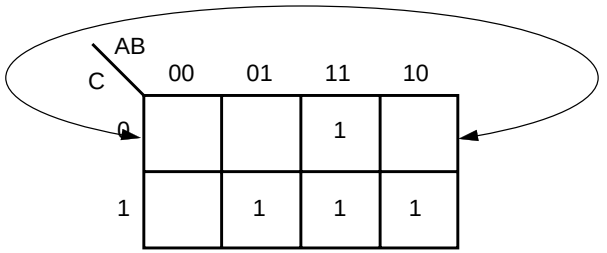


Fig A.41 A K-Map of the Majority Function

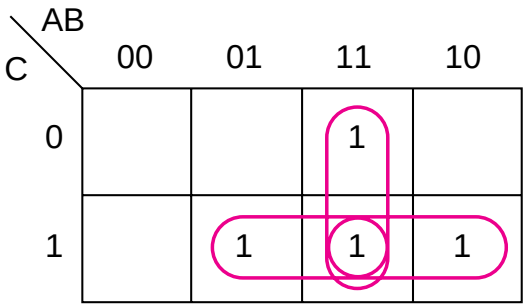
Place a "1" in each cell that has a that minterm.
Cells on the outer edge of the map "wrap around"

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



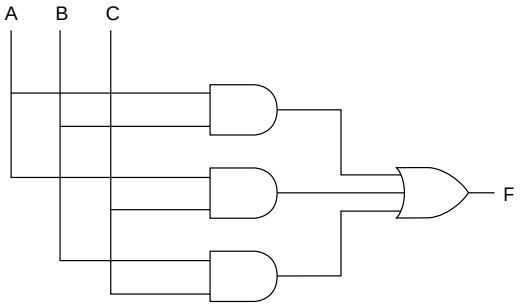
The map contains all the minterms. Adjacent 1's in the K-map satisfy the complement property of Boolean algebra.

Fig A.42 Adjacency Groupings for the Majority Function



M= BC + AC + AB

A.43 Minimized AND-OR Circuit for the Majority Function



M= BC + AC + AB

Fig A.44 Minimal and Not-Minimal K-Map Groupings

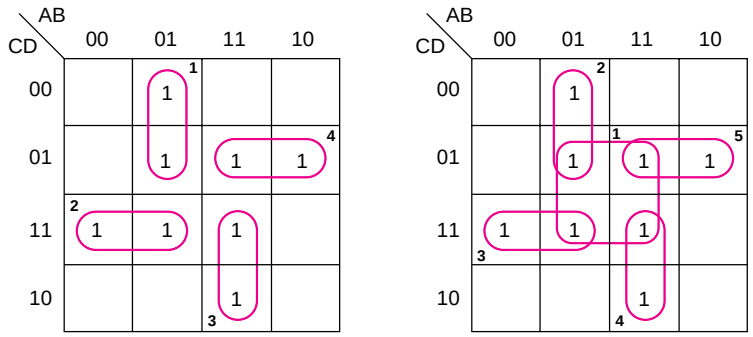
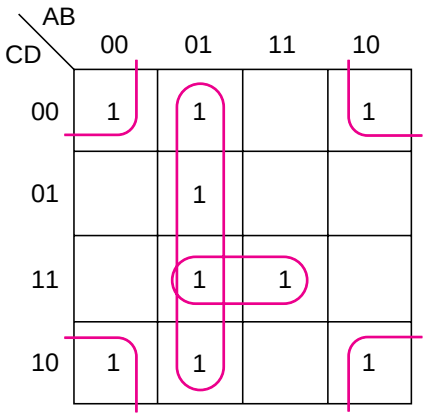
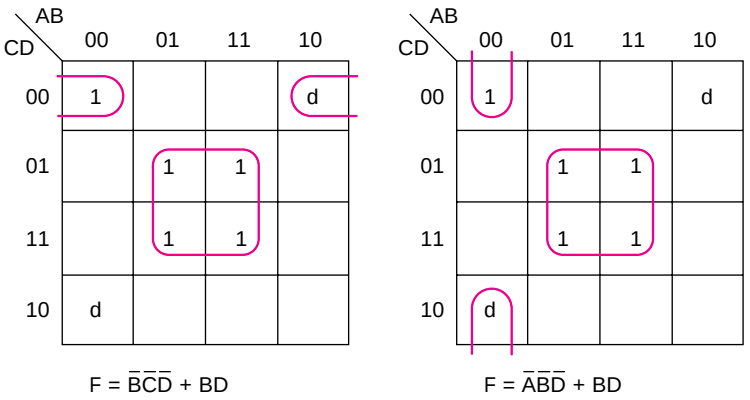


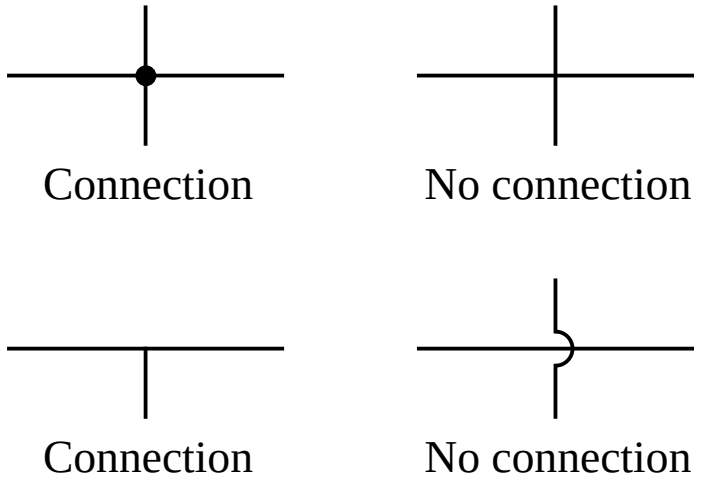
Fig A.45 The Corners of a K-Map Are Logically Adjacent



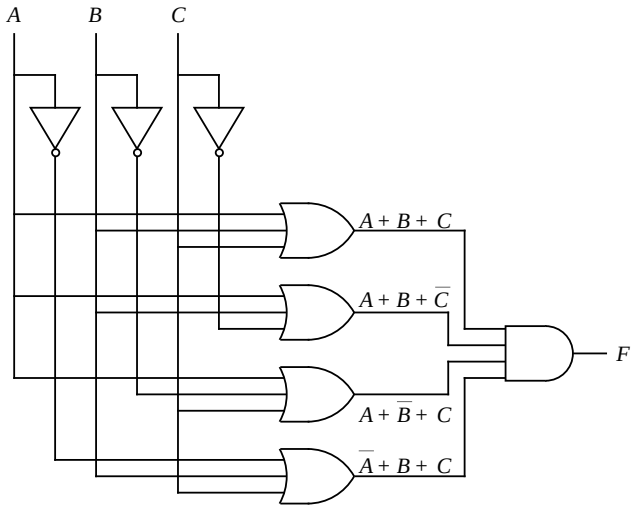
A.46 Two Different Minimized Equations Are Produced from the Same K-Map



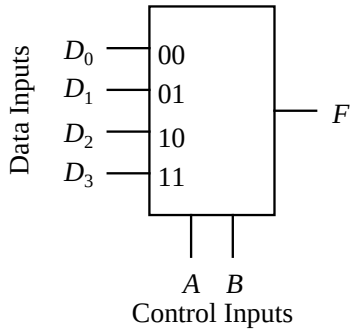
Notation Used at Circuit Intersections



OR-AND Implementation of Majority



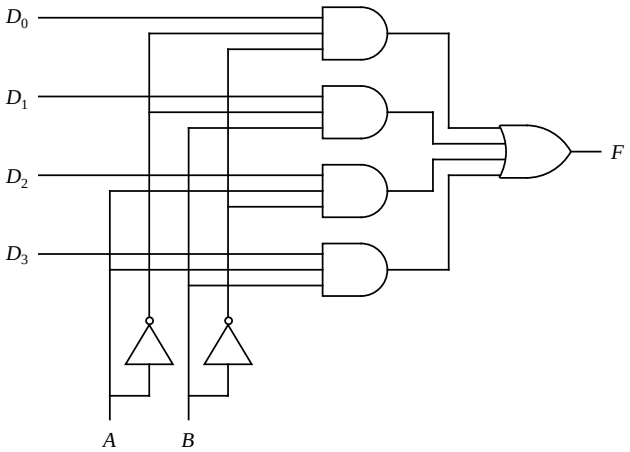
Multiplexer



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

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

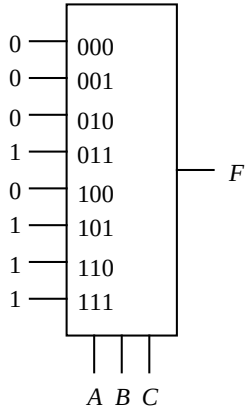
AND-OR Implementation of MUX



MUX Implementation of Majority

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

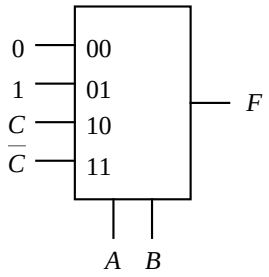
A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



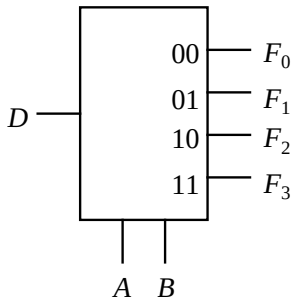
4-to-1 MUX Implements 3-Var Function

- **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

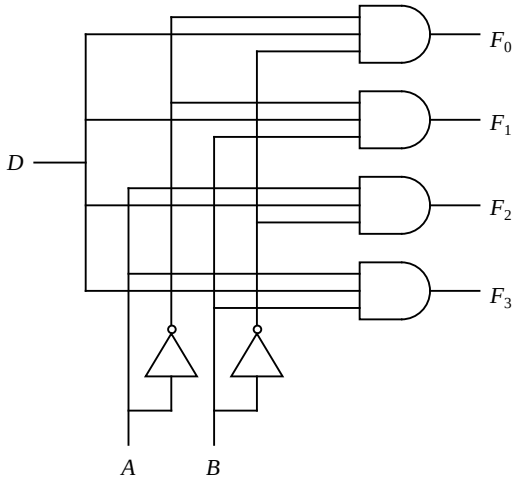


$F_0 = D\overline{A}\,\overline{B}$ $\quad F_2 = DA\overline{B}$

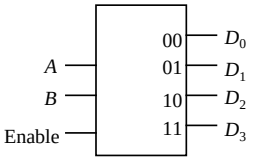
$F_1 = D\overline{A}B$ $\quad F_3 = DAB$

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

Gate-Level Implementation of DEMUX



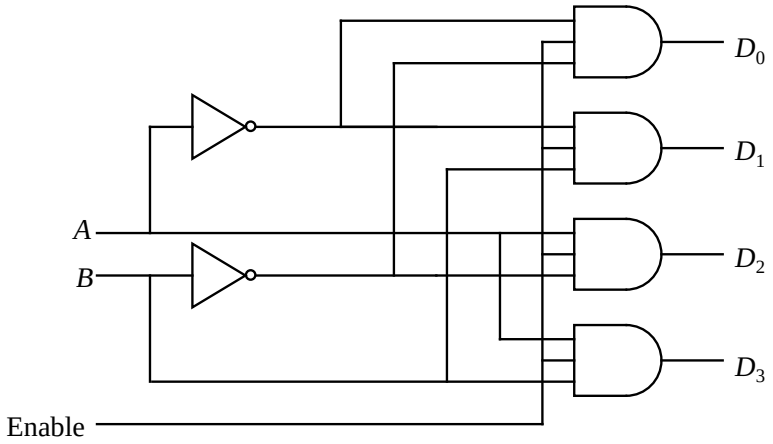
Decoder



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

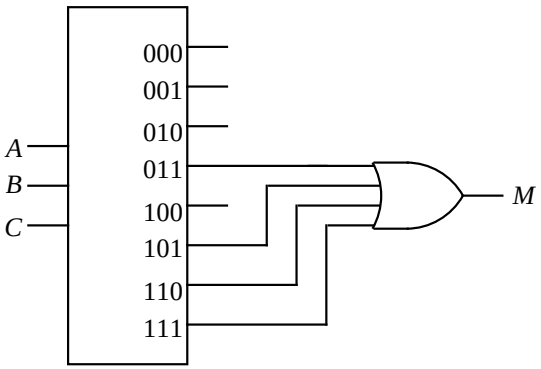
$D_0 = \overline{A}\,\overline{B}$ $D_1 = \overline{A}\,B$ $D_2 = A\,\overline{B}$ $D_3 = A\,B$

Gate-Level Implementation of Decoder



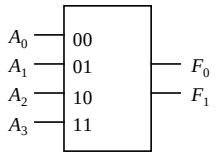
Decoder Implementation of Majority Function

- **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}

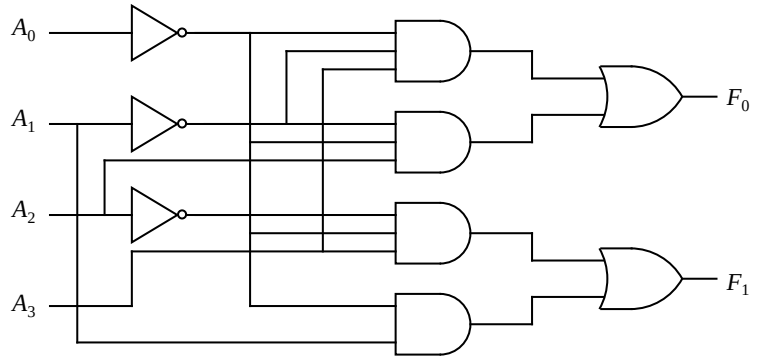


$$F_0 = \overline{A_0}\,\overline{A_1}A_3 + \overline{A_0}\,\overline{A_1}A_2$$

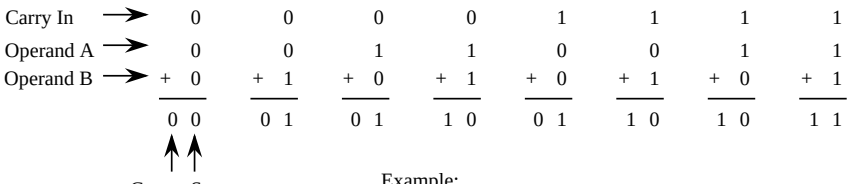
$$F_1 = \overline{A_0}A_2A_3 + \overline{A_0}A_1$$

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



Example: Ripple-Carry Addition

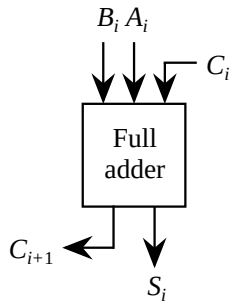


Example:

Carry		1	0	0	0
Operand A		0	1	0	0
Operand B	+	0	1	1	0
Sum		1	0	1	0

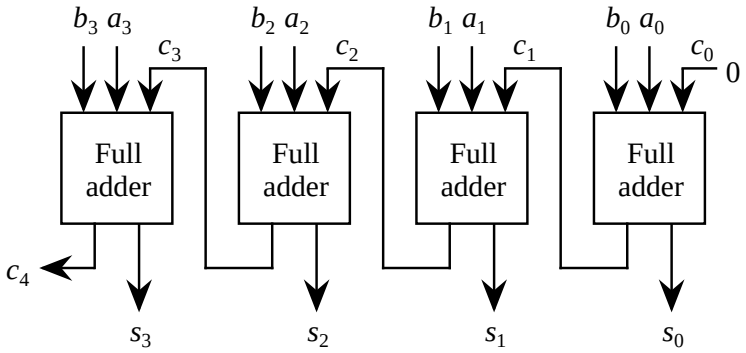
Full Adder

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



Four-Bit Ripple-Carry Adder

- **Four full adders connected in a ripple-carry chain form a four-bit ripple-carry adder.**

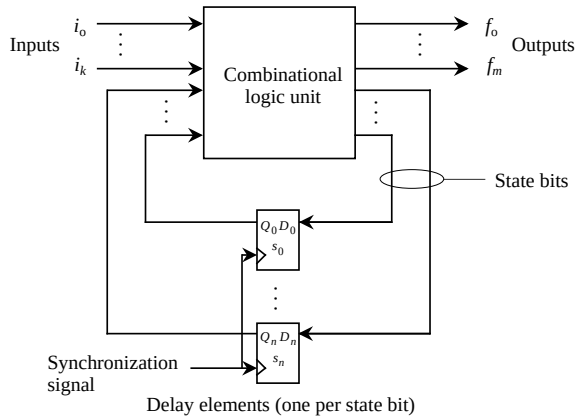


Sequential Logic

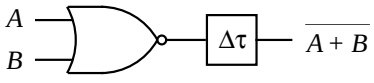
- **The combinational logic circuits we have been studying so far have no memory. The outputs always follow the inputs.**
- **There is a need for circuits with memory, which behave differently depending upon their previous state.**
- **An example is a vending machine, which must remember how many and what kinds of coins have been inserted. The machine should behave according to not only the current coin inserted, but also upon how many and what kinds of coins have been inserted previously.**
- **These are referred to as *finite state machines*, because they can have at most a finite number of states.**

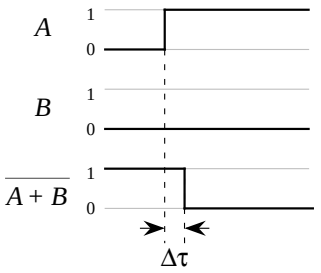
Classical Model of a Finite State Machine

- **An FSM is composed of a combinational logic unit and delay elements (called *flip-flops*) in a feedback path, which maintains state information.**



NOR Gate with Lumped Delay



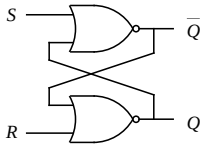


Timing Behavior

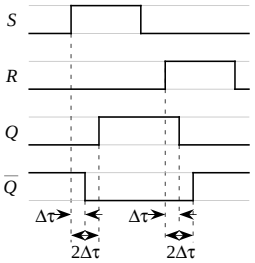
- **The delay between input and output (which is lumped at the output for the purpose of analysis) is at the basis of the functioning of an important memory element, the *flip-flop*.**

S-R Flip-Flop

- **The S-R flip-flop is an active high (positive logic) device.**

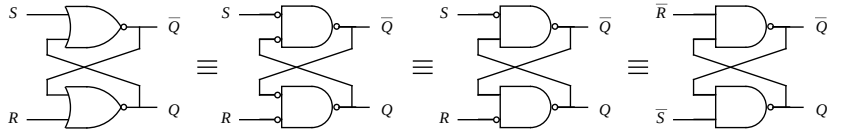


Q_t	S_t	R_t	Q_{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	(disallowed)
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	(disallowed)



Timing Behavior

NAND Implementation of S-R Flip-Flop

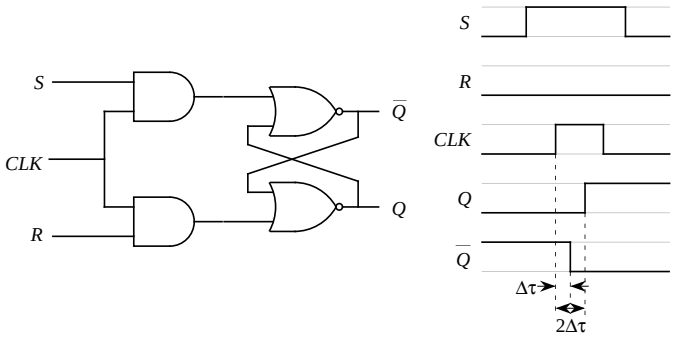


Scientific Prefixes

- **For computer memory, 1K = 2^{10} = 1024. For everything else, like clock speeds, 1K = 1000, and likewise for 1M, 1G, *etc.***

Prefix	Abbrev.	Quantity	Prefix	Abbrev.	Quantity
milli	m	10^{-3}	Kilo	K	10^{3}
micro	μ	10^{-6}	Mega	M	10^{6}
nano	n	10^{-9}	Giga	G	10^{9}
pico	p	10^{-12}	Tera	T	10^{12}
femto	f	10^{-15}	Peta	P	10^{15}
atto	a	10^{-18}	Exa	E	10^{18}

Clocked S-R Flip-Flop

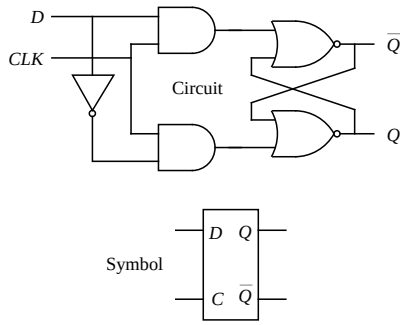


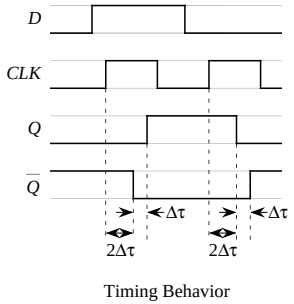
Timing Behavior

- **The clock signal, CLK, enables the S and R inputs to the flip-flop.**

Clocked D Flip-Flop

- **The clocked D flip-flop, sometimes called a *latch*, has a potential problem: If D changes while the clock is high, the output will also change. The *Master-Slave* flip-flop (next slide) addresses this problem.**

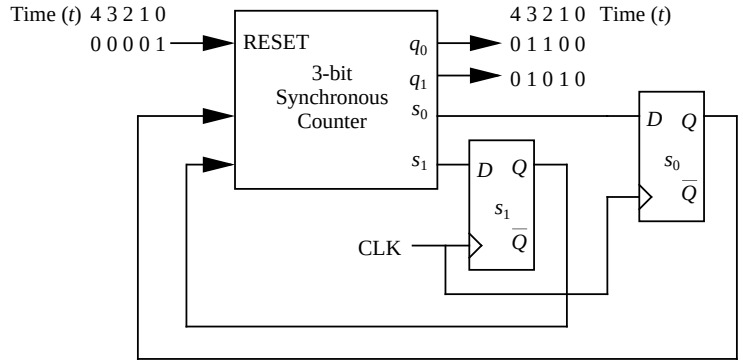




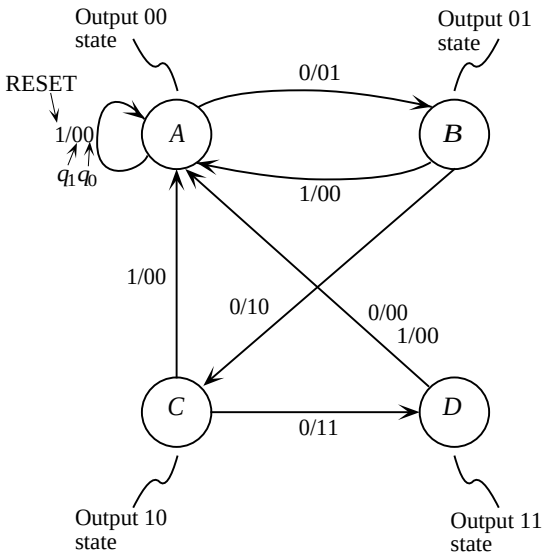
Timing Behavior

Example: Modulo-4 Counter

- **Counter has a clock input (CLK) and a RESET input.**
- **Counter has two output lines, which take on values of 00, 01, 10, and 11 on subsequent clock cycles.**



State Transition Diagram for Mod-4 Counter



State Table for Mod-4 Counter

Present state \ Input	*RESET* 0	*RESET* 1
A	*B*/01	*A*/00
B	*C*/10	*A*/00
C	*D*/11	*A*/00
D	*A*/00	*A*/00

Next state Output

State Assignment for Mod-4 Counter

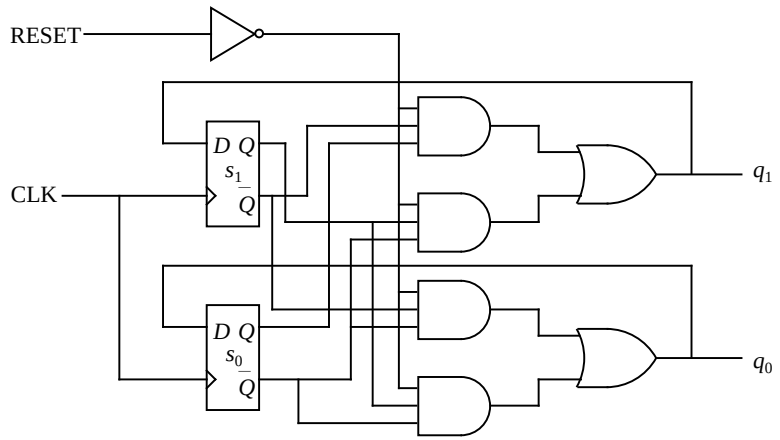
Present state (S_t) \ Input	*RESET* 0	*RESET* 1
A:00	01/01	00/00
B:01	10/10	00/00
C:10	11/11	00/00
D:11	00/00	00/00

Truth Table for Mod-4 Counter

RESET $r(t)$	$s_1(t)$	$s_0(t)$	$s_1s_0(t+1)$	$q_1q_0(t+1)$
0	0	0	01	01
0	0	1	10	10
0	1	0	11	11
0	1	1	00	00
1	0	0	00	00
1	0	1	00	00
1	1	0	00	00
1	1	1	00	00

$$s_0(t+1) = \overline{r}(t)\overline{s_1}(t)\overline{s_0}(t) + \overline{r}(t)s_1(t)\overline{s_0}(t)$$
$$s_1(t+1) = \overline{r}(t)\overline{s_1}(t)s_0(t) + \overline{r}(t)s_1(t)\overline{s_0}(t)$$
$$q_0(t+1) = \overline{r}(t)\overline{s_1}(t)\overline{s_0}(t) + \overline{r}(t)s_1(t)\overline{s_0}(t)$$
$$q_1(t+1) = \overline{r}(t)\overline{s_1}(t)s_0(t) + \overline{r}(t)s_1(t)\overline{s_0}(t)$$

Logic Design for Mod-4 Counter



Sequence Detector State Table

Present state \ Input	*X* 0	*X* 1
A	*B*/0	*C*/0
B	*D*/0	*E*/0
C	*F*/0	*G*/0
D	*D*/0	*E*/0
E	*F*/0	*G*/1
F	*D*/0	*E*/1
G	*F*/1	*G*/0

Sequence Detector State Assignment

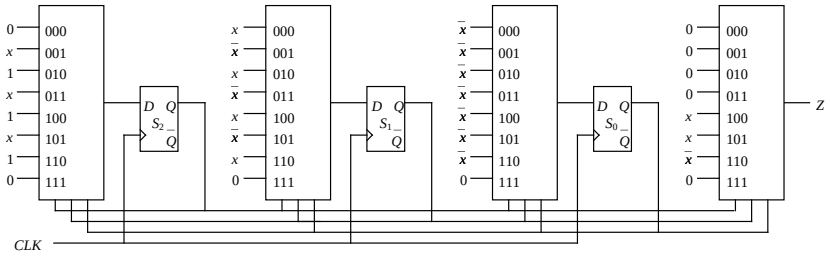
Present state \ Input	X = 0	X = 1
	$S_2S_1S_0$	$S_2S_1S_0Z$	$S_2S_1S_0Z$
A: 000	001/0	010/0
B: 001	011/0	100/0
C: 010	101/0	110/0
D: 011	011/0	100/0
E: 100	101/0	110/1
F: 101	011/0	100/1
G: 110	101/1	110/0

(a)

Input and state at time t				Next state and output at time t+1			
s_2	s_1	s_0	x	s_2	s_1	s_0	z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	0
0	1	0	0	1	0	1	0
0	1	0	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	1	1	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	1	1	0	1
1	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	0
1	1	1	0	d	d	d	d
1	1	1	1	d	d	d	d

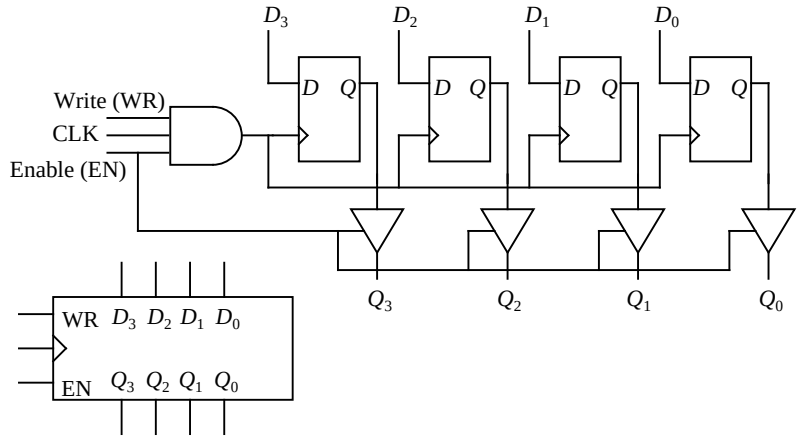
(b)

Sequence Detector Logic Diagram



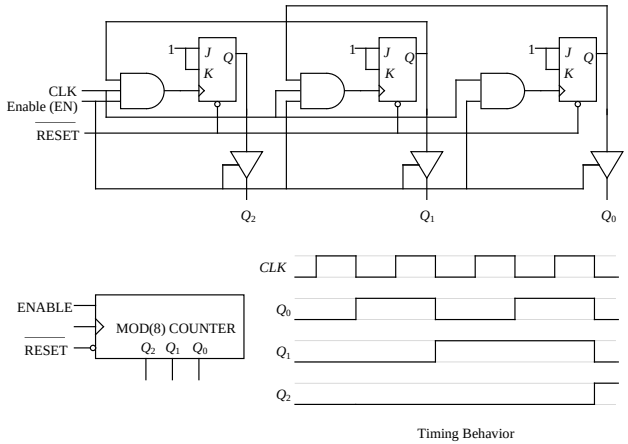
Four-Bit Register

- **Makes use of tri-state buffers so that multiple registers can gang their outputs to common output lines.**



Modulo-8 Counter

- **Note the use of the T flip-flops, implemented as J-K's. They are used to toggle the input of the next flip-flop when its output is 1.**



Timing Behavior