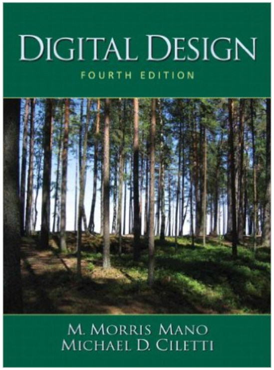
CS221: Digital Design

Waleed A. Yousef, Ph.D.,

Human Computer Interaction Lab.,
Computer Science Department,
Faculty of Computers and Information,
Helwan University,
Egypt.

October 25, 2017

Lectures follow (and some figures are adapted from):



Mano, M. M., Ciletti, M. D., 2007. Digital design, 4th Edition. Prentice-Hall, Upper Saddle River, NJ

We will proceed as follows:

- Fast introduction (few sections from Chapter 1).
- Detailed study of Chapters 2-7; very few sections will be skipped. At the end of each chapter Verilog code for some circuits will be explained.
- If time permits, Chapter 8 will be covered in full or in parts

Course Objectives

This course combines three approaches to teach students Digital Design, which is the fundamental prerequisite to understand computer design and architecture:

1. Theoretical aspects of the subject will be covered in lectures, along with exercises in sections. Students, by the end of the course, should be able to design, analyze, and implement combinational and synchronous digital circuits.

2. A second objective is to teach students the digital design using a Hardware Descriptive Language (HDL). Students by the end of the semester will be able to analyze logic circuits with Verilog (one of the available HDLs).

3. A third objective is to develop the practical sense of the students through lab. experiments. Students will be able to implement logic circuits using breadboards and ICs.

Contents

Chapter 1

Introduction to Digital Systems

1.9 From Binary Logic (Mathematics) to Logic Gates (Circuits)

From what we have studied in Discrete Mathematics, we have the following basic three logic functions:

Truth Tables of Logical Operations

AND

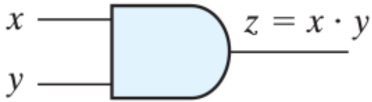
x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

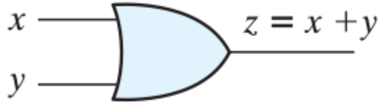
x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

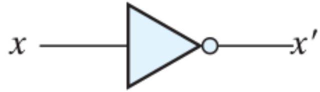
x	x'
0	1
1	0



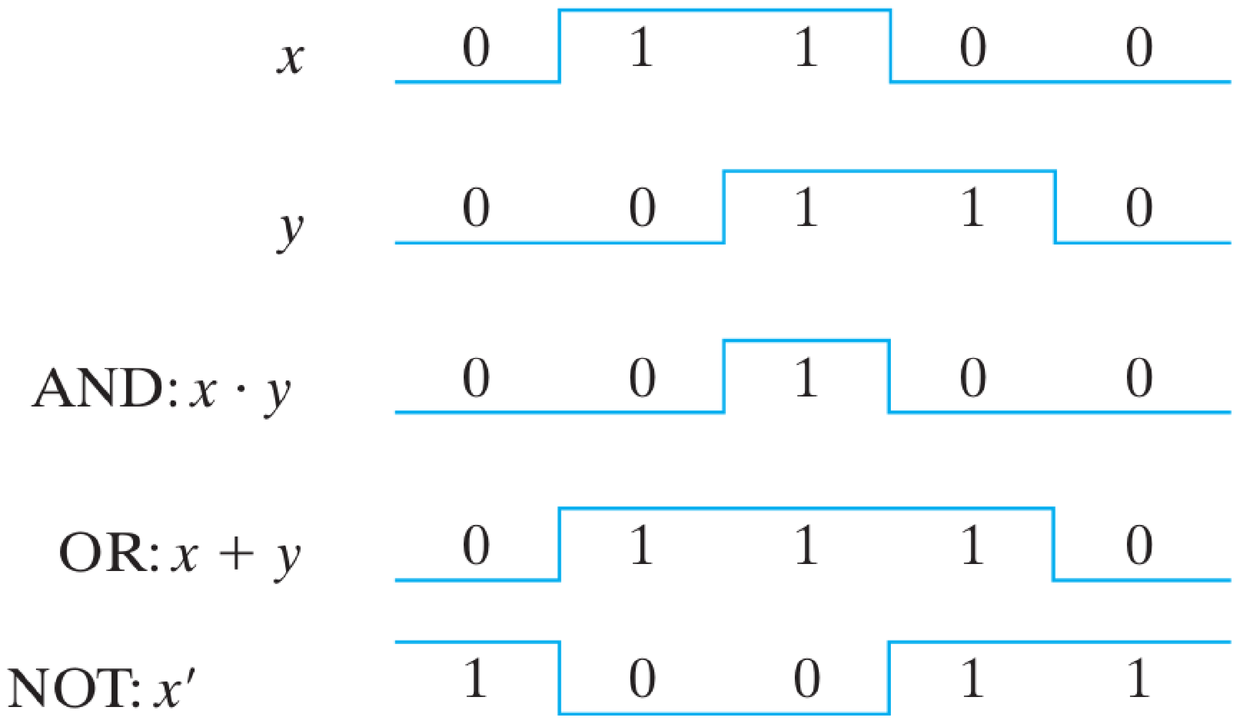
(a) Two-input AND gate



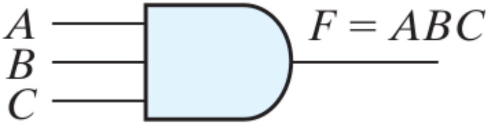
(b) Two-input OR gate



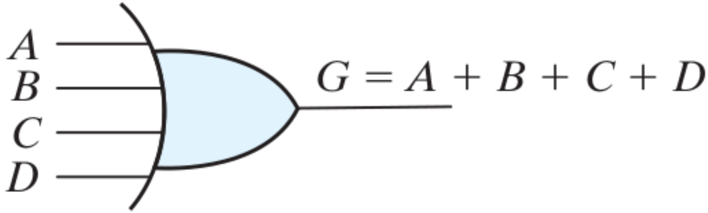
(c) NOT gate or inverter



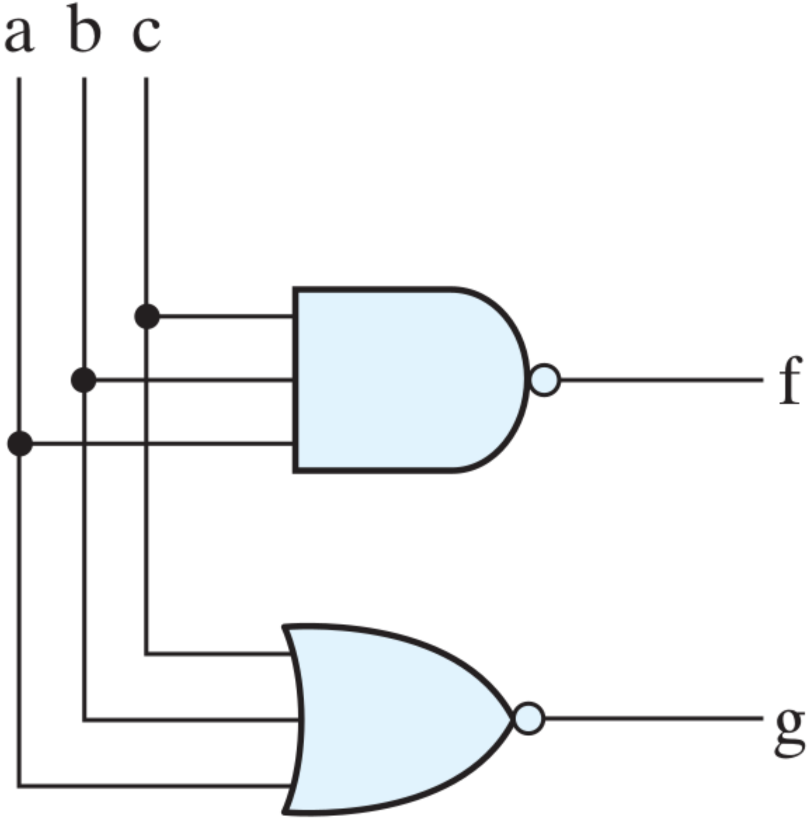
x
0 1 1 0 0
y
0 0 1 1 0
AND: x · y
0 0 1 0 0
OR: x + y
0 1 1 1 0
NOT: x′
1 0 0 1 1



(a) Three-input AND gate

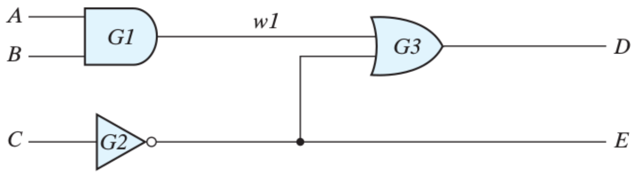


(b) Four-input OR gate


a
b
c
f
g

Revisiting Course Objectives

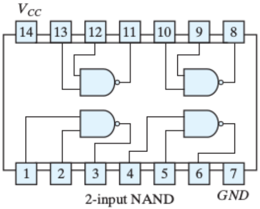
Theory (hands on paper)



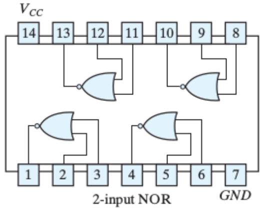
Simulation using Verilog

```
// Verilog model: Simple_Circuit
module Simple_Circuit (A, B, C, D, E);
  output D, E;
  input  A, B, C;
  wire   w1;

  and    G1 (w1, A, B); //Optional gate instance
  not    G2 (E, C);
  or     G3 (D, w1, E);
endmodule
```

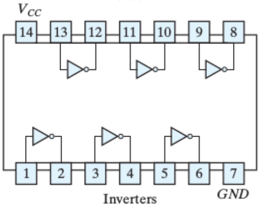
Hardware Implementation



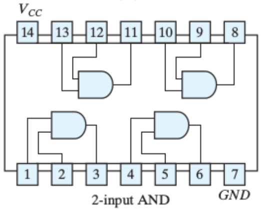
2-input NAND
7400



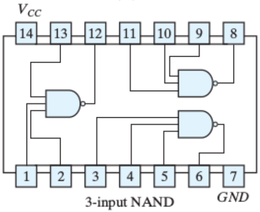
2-input NOR
7402



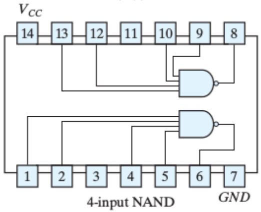
Inverters
7404



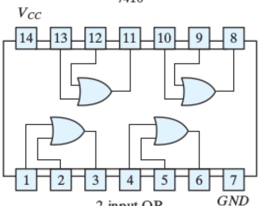
2-input AND
7408



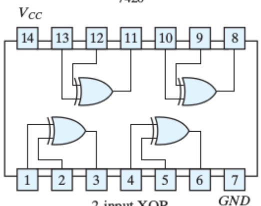
3-input NAND
7410



4-input NAND
7420



2-input OR
7432



2-input XOR
7486

Chapter 2

Boolean Algebra and Logic Gates

2.3 Axiomatic Definition of Boolean Algebra

Definition 1 *:*

- *Denote* `True` *and* `False` *by 1 and 0 that represent* V_{cc} *and 0 voltages.*
- *A* ***bit*** *(**b**inary dig**it**) is a symbol of these two values.*
- *A Boolean Variable is a variable that is either* `True` *or* `False` *(or simply 1 or 0); hence a bit.*
- *Given a set* $B = 0, 1$*, the three operators*

  $\cdot(AND), \quad +(OR), \quad '(NOT)$ *are defined by:*
- *A truth table is the table that represents all the possi- ble combinations of the input to a logical (Boolean) function.*

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

Claim 2 (Boolean Postulates Come "Dual") *:*

1. *Closure: the structure is closed w.r.t.* $+$, $\cdot$
2. *Identity element:*
   a) $0 + x = x$.
   b) $1 \cdot x = x$.
3. *Commutative:*
   a) $x + y = y + x$.
   b) $x \cdot y = y \cdot x$.
4. *Distributive:*
   a) $x \cdot (y + z) = x \cdot y + x \cdot z$.
   b) $x + (y \cdot z) = (x + y)(x + z)$.
5. *Complement:*
   a) $x + x' = 1$.
   b) $x \cdot x' = 0$

Proof. :

1. is clear from truth table above.
2. by simple truth table:
3. is from definition.
4. by constructing the truth table below:
5. from very simple truth table

■

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$y+z$	$x \cdot (y+z)$
0	0
1	0
1	0
1	0
0	0
1	1
1	1
1	1

$x \cdot y$	$x \cdot z$	$(x \cdot y)+(x \cdot z)$
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1
1	1	1

2.4 Basic Theorems and Properties of Boolean Algebra

Theorem 3 (dual postulates and theorems) :

Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

Proof. (algebraically and by truth table. Some algebraic proofs are tedious and truth table is sufficient):

Theorem 1(a) algebraically :

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

Theorem 1(a) by truth table :

Theorem 1(b) algebraically :

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Theorem 1(b) by truth table :

Theorem 2(a-b) : proof is very similar to above.

Theorem 6(a) : algebraically

$$
\begin{aligned}
x + xy &= x \cdot 1 + xy \\
&= x \cdot (1 + y) \\
&= x \cdot 1 \\
&= x
\end{aligned}
$$

Theorem 5(a) DeMorgan : truth table

x	y	$x+y$	$(x+y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x'	y'	$x'y'$
1	1	1
1	0	0
0	1	0
0	0	0

■

Theorem 4 (Generalization) . *It can be easily shown that Theorems 4, 5 in the above table are generalize to more than 2 variables.*

Proof. is straightforward by mathematical induction.

■

Example 5

$$(A_1 + A_2 + \cdots + A_n)' = A_1' A_2' \cdots A_n'$$
$$(A_1 A_2 \cdots A_n)' = A_1' + A_2' + \cdots + A_n'.$$

2.4.1 Operator Precedence

(), `NOT`, `AND`, `OR`:

Example 6

$$x + y \cdot (x+z)'$$

2.5 Boolean Functions and Gate Implementation

Example 7 *Evaluate and implement the functions:*

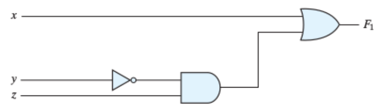
$$F_1 = x + y'z$$
$$F_2 = x'y'z + x'yz + xy'$$

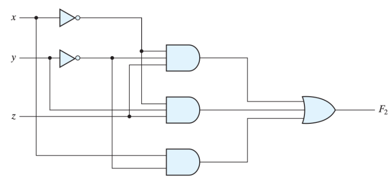
Hint: be smart in filling the truth table:

Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Hint: draw the circuits elegantly:





Simplify the function to simplify the circuit; why?

$$\begin{aligned} F_2 &= x'y'z + x'yz + xy' \\ &= x'z(y' + y) + xy' \\ &= x'z + xy', \end{aligned}$$

which has only 4 literals, *where a literal is a single variable (complemented or un-complemented).*

Less gates (HW) means:

- *low cost.*
- *low power consumption.*
- *simpler implementation.*

Simplifying functions is by:

- *algebraic manipulation (this chapter)*
- *K-map (next chapter)*
- *computer programs for many-input functions*

Example 8 *Simplify the following function to a minimum number of literals:*

$$
\begin{aligned}
F_1 &= x(x' + xy) \\
&= xx' + xxy \\
&= 0 + xy = xy.
\end{aligned}
$$

$$
\begin{aligned}
F_2 &= xy + x'z + yz \\
&= xy + x'z + xyz + x'yz \\
&= xy(1+z) + x'z(1+y) \\
&= xy + x'z.
\end{aligned}
$$

$$
\begin{aligned}
F_3 &= (x(y'z' + yz))' \\
&= x' + (y'z' + yz)' \\
&= x' + (y+z)(y'+z') \\
&= x' + yz' + y'z.
\end{aligned}
$$

Hint: for simplicity apply DeMorgan by duality followed by inverting each literal:

Example 9

$$
\begin{aligned}
F_3 &= x(y'z' + yz) \\
\textit{dual of}\, F_3 &= x + (y' + z')(y + z) \\
F_3' &= x' + (y + z)(y' + z').
\end{aligned}
$$

2.6 Canonical and Standard Forms

2.6.1 Minterms and Maxterms

Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	**y**	**z**	**Term**	**Designation**	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

- a minterm (or maxterm) equals one (or zero) only at one combination.
- the minterm (or maxterm) subscript is the order and value of this combination.
- order of variables is very important!
- $m_i = M_i'$.
- E.g, what is the truth table of m_0 and M_0?
- Each function, then, can be expressed as either: **sum of minterms or product of maxterms!**

Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	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

From Truth Table:

ORing (sum of) minterms

$$\begin{aligned} f_1 &= \sum_{1\ of\ f_1} = \sum(1,4,7) = m_1 + m_4 + m_7 \\ &= x'y'z + xy'z' + xyz, \\ f_1' &= \sum_{1\ of\ f_1'} = \sum_{0\ of\ f_1} = \sum(0,2,3,5,6) = m_0 + m_2 + m_3 + m_5 + m_6. \end{aligned}$$

ANDing maxterms

$$\begin{aligned} f_1 &= \prod_{0\ of\ f_1} = \prod(0,2,3,5,6) = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x+y+z)(x+y'+z)(x+y'+z')(x'+y+z')(x'+y'+z), \\ f_1' &= \prod_{0\ of\ f_1'} = \prod_{1\ of\ f_1} = \prod(1,4,7) = M_1 M_4 M_7. \end{aligned}$$

Conclusion: For any function f equals 1 on the set *1* and 0 on the set *0*:

$$\sum_{\mathit{1\ of\ f}} = \prod_{\mathit{0\ of\ f}} \Longrightarrow f,$$
$$\sum_{\mathit{0\ of\ f}} = \prod_{\mathit{1\ of\ f}} \Longrightarrow f'.$$

This is obtainable, as well, from DeMorgan:

$$\begin{aligned} f &= \sum_{\mathit{1}} \\ &= (m_i + m_j + \cdots) \\ f' &= (m_i + m_j + \cdots)' \\ &= m_i' \cdot m_j' \cdots \\ &= M_i \cdot M_j \cdots \\ &= \prod_{\mathit{1}} \end{aligned}$$

Gate Implementation

$$f_1 = \sum(1,4,7) = m_1 + m_4 + m_7 = x'y'z + xy'z' + xyz$$

Converting algebraic expression to: sum of minterms or product of maxterms

Example 10 (two variables)

$$
\begin{aligned}
A &= A(B' + B) = AB' + AB \\
&= m_2 + m_3.
\end{aligned}
$$

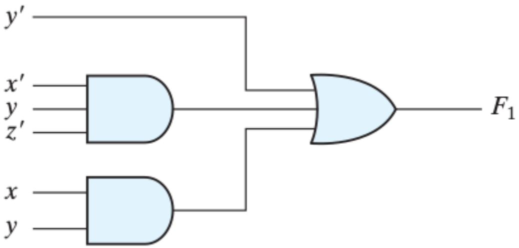
Example 11

$$
\begin{aligned}
A &= A(B'C' + B'C + BC' + BC) \\
&= AB'C' + AB'C + ABC' + ABC \\
&= m_4 + m_5 + m_6 + m_7.
\end{aligned}
$$

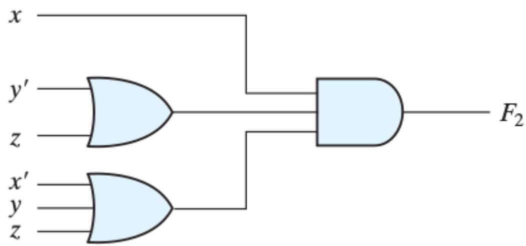
2.6.2 Standard Forms

- Sum of minterms and product of maxterms are almost not minimized (because each term contains by construction all variables).
- We should minimize and keep it as sum of products (SOP) or product of sums (POS), because it is only two-level implementation; e.g.,

$$F_1 = y' + xy + x'yz',$$
$$F_2 = x(y' + z)(x' + y + z').$$



(a) Sum of Products



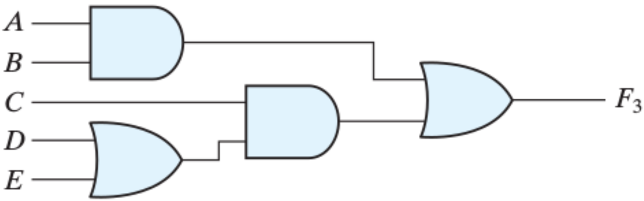
(b) Product of Sums

- Observe that, we do not consider inversion a level.
- This provides minimal signal delay and simpler implementation.
- So, implement any function as SOM (POM), then simplify to SOP (POS).

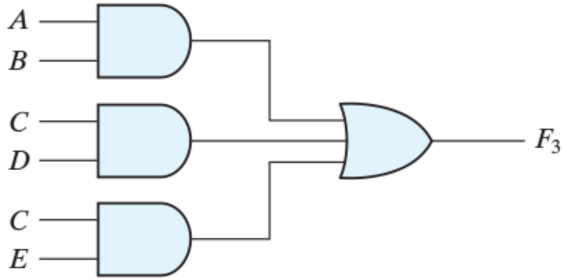
Example 12 (2- vs. 3-level implementation)

$$F_3 = AB + C(D + E)$$

$$F_3 = AB + CD + CE.$$



(a) $AB + C(D + E)$



(b) $AB + CD + CE$

2.7 Other Logic Operations

Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

For n binary variables:

$$\begin{aligned}\text{number of functions} &= 2^{(\text{number of rows in truth table})} \\ &= 2^{2^n}.\end{aligned}$$

$$\begin{aligned}
F_0 &= 0, && && \\
F_1 &= m_3 &&= xy, && \\
F_2 &= m_2 &&= xy', && \\
F_3 &= m_2 + m_3 &&= xy' + xy &&= x, \\
F_4 &= m_1 &&= x'y, && \\
F_5 &= m_1 + m_3 &&= x'y + xy &&= y, \\
F_6 &= m_1 + m_2 &&= x'y + xy' &&= x \oplus y, \\
F_7 &= M_0 &&= x + y. &&
\end{aligned}$$

The other functions can be obtained, of course, by complementing the previous functions (Why?)

Boolean Expressions for the 16 Functions of Two Variables

Boolean Functions	**Operator Symbol**	**Name**	**Comments**
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Special offer for Mathematics lovers: from our study in Discrete Mathematics we have already learned that (Ch1. Rosen, 2007, P. 6):

Consider the sets $\mathcal{X}, \mathcal{Y}$, and any element e, and the indicators

$$x = I_{\mathcal{X}} = e \in \mathcal{X},$$
$$y = I_{\mathcal{Y}} = e \in \mathcal{Y},$$

then all the following are equivalent:

- $\mathcal{X} \subset \mathcal{Y}$ (**typo in book**),

- If x then y, y if x, y when x,

- x implies y, y follows from x,

- x is sufficient for y, y is necessary for x,

- $F = M_2 = x' + y$.

HW: what is the logic function F that represent the case that $\mathcal{X} \cap \mathcal{Y} = \phi$ and $\mathcal{X} \cup \mathcal{Y} \neq \Omega$.

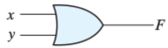
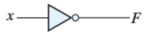
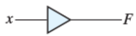
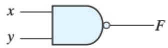
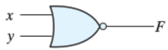
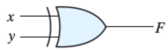
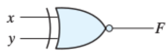
2.8 Other Digital Logic Gates

- From purely mathematical point of view: do we need more gates to implement other functions?

  Problem 13 *(Sec. 1.2 Rosen, 2007, Prob.: 43–45): Show that `NOT`, `OR`, `AND` form a functionally complete collection of logical operators.*

- **Motivation:** why, then, introducing new logic gates: `NAND`, `NOR` (next figure)? As we will see in Sec. 3.7:
  - `NAND` can represent the main three logic functions!
  - Similarly `NOR` (HW).
  - $\sum$ can be implemented **only** with `NAND`.
  - $\prod$ can be implemented **only** with `NOR`.
  - Therefore, only `NAND` or `NOR` suffice.
  - Great to have uniform gate delay.

- We already know `AND`, `OR`, `NOT` gates, with direct hardware implementation. `NAND`, `NOR` have their hardware implementation as well.

<table>
<tr><th>Name</th><th>Graphic symbol</th><th>Algebraic function</th><th>Truth table</th></tr>
<tr><td>AND</td><td></td><td>$F = x \cdot y$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>OR</td><td></td><td>$F = x + y$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>Inverter</td><td></td><td>$F = x'$</td><td><table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>Buffer</td><td></td><td>$F = x$</td><td><table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>NAND</td><td></td><td>$F = (xy)'$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>NOR</td><td></td><td>$F = (x + y)'$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>Exclusive-OR
(XOR)</td><td></td><td>$F = xy' + x'y$
$= x \oplus y$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>Exclusive-NOR
or
equivalence</td><td></td><td>$F = xy + x'y'$
$= (x \oplus y)'$</td><td><table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
</table>

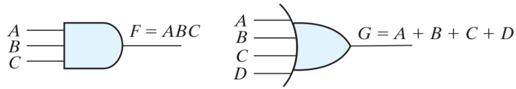
2.8.1 Extention to Multiple Inputs

`AND`, `OR` are directly extensible to multiple input from their commutative and associative laws:

$$(x+y)+z = x+(y+z),$$
$$(xy)z = x(yz).$$

Therefore, we **DEFINE:**

$$x+y+z = (x+y)+z = x+(y+z),$$
$$xyz = (xy)z = x(yz).$$

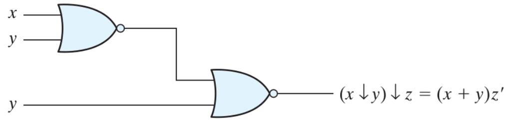


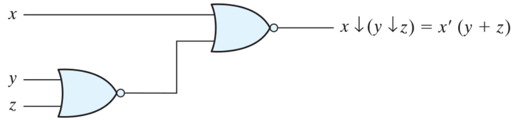
(a) Three-input AND gate (b) Four-input OR gate

Unfortunately: ↓ (`NOR`), ↑ (`NAND`) are not associative:

$$(x \downarrow y) \downarrow z = ((x+y)'+z)' = (x+y)z' = xz' + yz',$$
$$x \downarrow (y \downarrow z) = (x+(y+z)')' = x'(y+z) = x'y + x'z.$$

HW: prove the same for NAND ↑.

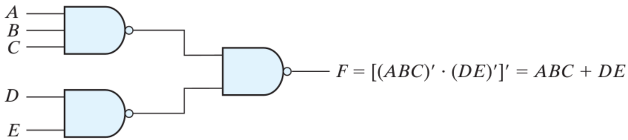




Hence, we have to **DEFINE:**

$$x \downarrow y \downarrow z = (x + y + z)',$$
$$x \uparrow y \uparrow z = (xyz)'.$$

(a) 3-input NOR gate (b) 3-input NAND gate



(c) Cascaded NAND gates

XOR Function

- Prove (by truth table) that it is commutative and associative; i.e.,

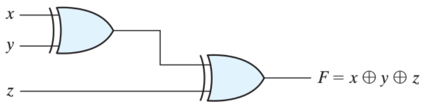
$$x \oplus y = y \oplus x,$$
$$x \oplus (y \oplus z) = (x \oplus y) \oplus z.$$

- Hence, we **DEFINE**:

$$x \oplus y \oplus z = (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

- From its truth table, all the following are correct
  - $x \oplus y = 1$ when only x or only y is 1.
  - $x \oplus y = 1$ when x and y are different.
  - $x \oplus y = 1$ when x and y have odd # of 1s.

- Get the truth table of $x \oplus y \oplus z$ as:



(a) Using 2-input gates

(b) 3-input gate

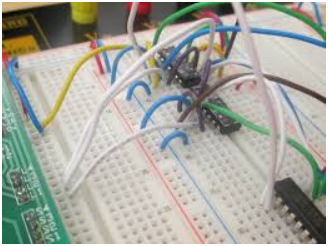
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(c) Truth table

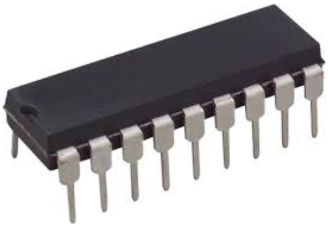
Therefore, the third meaning is emphasized:
$x \oplus y \oplus z = 1$ when they have odd number of ones; (more coming on parity check)

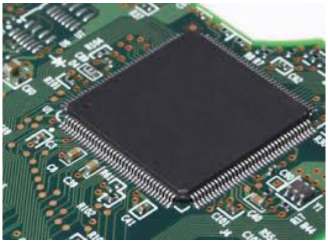
2.9 Integrated Circuits (ICs)

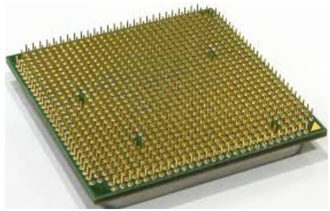
- We design a digital circuit using theory, HDL, and HW implementation in lab. (breadboards, ...) using data-sheets on page (8)



- After settling on particular design, this circuit can be fabricated on an IC using sophisticated engineering tools and industry







2.9.1 Levels of Integration

Small-Scale Integration (SSI) ~ 10 gates;

(Ch. 3)

Medium-Scale Integration (MSI) $\sim 10 \sim 1000$ gates;

(Ch. 4 and 6)

Lare-Scale Integration (LSI) $\sim 10,000$ gates;

(Ch. 7 and Computer Organization course.)

Very-Larage-Scale Integration $\sim 100,000$ gates;

(complex processors,...)

Chapter 3

Gate-Level Minimization

3.1 Introduction

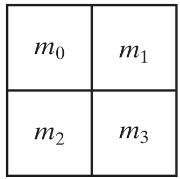
Let's remember why minimization (page 17).

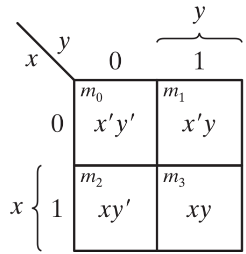
3.2 The Map Method

- K-map (named after Karnaugh) is a visual method, utilizing **visual power**.
- Difficult for more than 5 variables.
- Produces always SOP or POS expressions.

3.2.1 Two-Variable Map

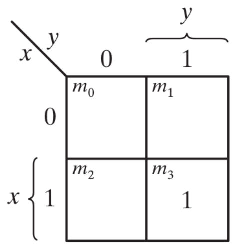
0	0
0	1
1	0
1	1

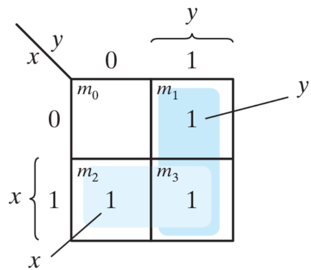




Simplify the following functions both algebraically and with K-Map and observe the visual power:

$$F_1 = m_1 + m_2 + m_3 \qquad = x'y + xy' + xy \qquad = xy + x'y + xy' + xy = y + x.$$
$$F_2 = m_3 \qquad = xy.$$
$$F_3 = m_0 + m_3 \qquad = x'y' + xy.$$

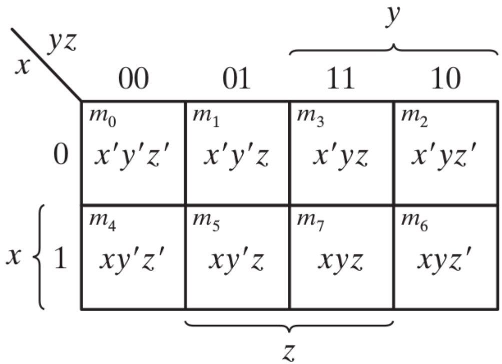




- Only one-bit (variable) change for any two adjacent squares!

3.2.2 Three-Variable Map

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

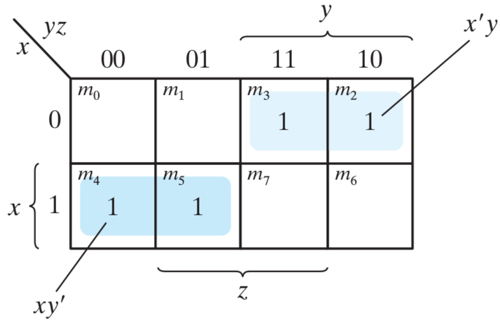


- Only one-bit (variable) change for any two adjacent squares! Therefore, e.g.,

$$F = m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

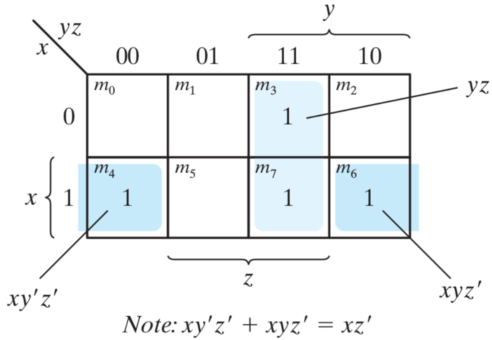
- Make sure of number of **variables** (e.g., $m_2 + m_3$ for 2 or 3 variables) and their **order**

Example 14 : *Simplify the function* $F(x, y, z) = \sum(2, 3, 4, 5)$.



$$F = x'y + xy' = x \oplus y.$$

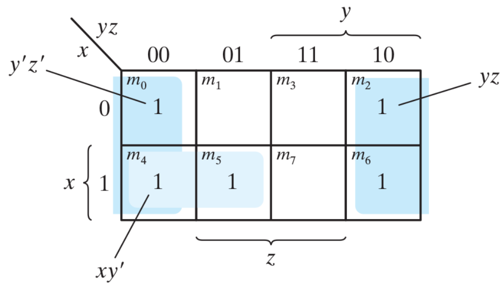
Example 15 *Simplify* $F(x, y, z) = \sum(3, 4, 6, 7)$. *(**Hint:** look at the most isolated 1s first.)*



$$F = xz' + yz.$$

Example 16 : *Consider the function* $F(x, y, z) = \sum(0, 2, 4, 5, 6)$.

1. *Simplify* F. (**Hint:** *a group should have* (2^m) *ones and its resulting SOP has* $(n - m)$ *literals.*
2. *Implement the function using* `AND`, `OR`, `NOT`.
3. *Observe the number of* `AND` *and* `OR` *gates (ignore inverters for now).*



Note: $y'z' + yz' = z'$

$$m_0 + m_2 + m_4 + m_6 = x'y'z' + x'yz' + xy'z' + xyz' \qquad (z' \text{ with the 4 combinations of } x, y)$$

$$= (x'y' + x'y + xy' + xy)z' \qquad (z' \cdot \textstyle\sum_{\text{all Minterms of } x,\ y} = z')$$

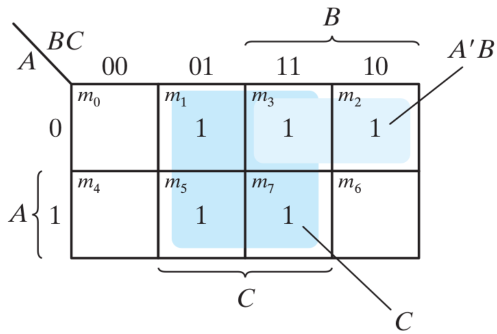
$$= (x'(y' + y) + x(y' + y))z' = (x' + x)z' = z'.$$

$$F = xy' + z'.$$

Example 17 *Consider the function $F = A'C + A'B + AB'C + BC$.*

1. *Express the function as a sum of Minterms.*
2. *Find the minimal SOP expression.*

Hint: *each SOP term missing m literals will be expanded by 2^m Minterms.*



$$F(A, B, C) = \sum(1, 2, 3, 5, 7)$$
$$F = C + A'B.$$

Golden Rules to Remember:

- Only one-bit (variable) change for any two adjacent squares!
- The boundaries are adjacent as well.
- A group of ones should be 2^m, where m is the number of removed variables in this group (SOP), and the SOP will have $n - m$ literals.
- Therefore, maximize the number of 1s in each group to minimize the number of literals in the SOP.
- Minimize the number of groups (the SOP terms).
- Therefore, start with the most isolated 1s.
- Make sure of number of **variables** and their **order**
- The number of literals in each group (SOP) is the number of inputs to its `AND` gate.
- The number of groups is the number of SOP terms is the number of `AND` gates is the number of inputs to the `OR` gate.
- All Minterms are covered.

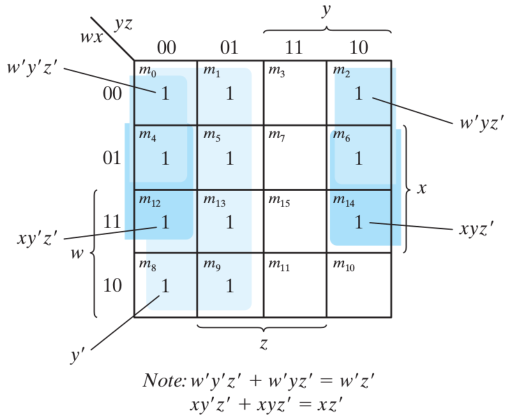
3.3 Four-Variable Map

$wx \backslash yz$	00	01	11	10
00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$
01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$
11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$
10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$

- Adjacency from top-bottom, right-left, and corners.
- **Corners:** w and y took their 4 combinations at $x = 0,\ z = 0$: $\Longrightarrow$

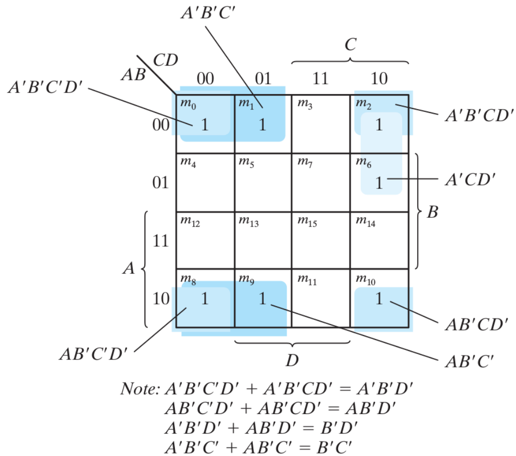
$$
\begin{aligned}
m_0 + m_2 + m_8 + m_{10} &= w'x'y'z' + w'x'yz' + wx'y'z' + wx'yz' \\
&= (w'y' + w'y + wy' + wy)x'z' \\
&= x'z'.
\end{aligned}
$$

Example 18 *Simplify the function* $F(w,x,y,z) = \sum(0,1,2,4,5,6,8,9,12,13,14)$



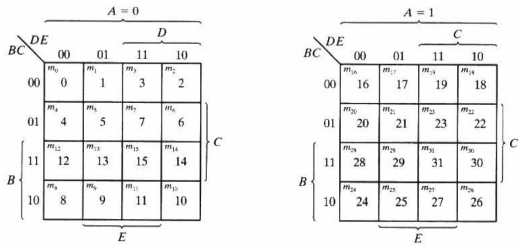
$$F = y' + w'z' + xz'.$$

Example 19 *Simplify the function* $F = A'B'C' + B'CD' + A'BCD' + AB'C'$.



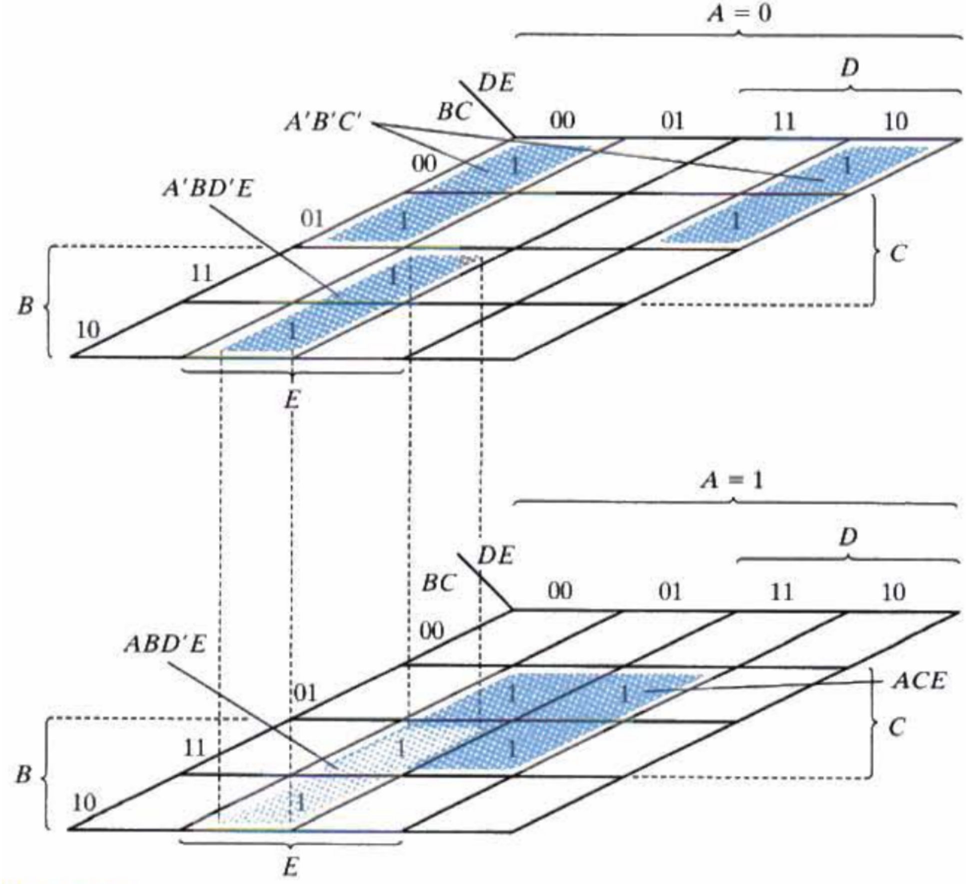
$$F = B'D' + B'C' + A'CD'.$$

3.4 Five-Variable Map



Example 20 *Simplify the function* $F(A,B,C,D,E) = \sum(0,2,4,6,9,13,21,23,25,29,31)$.

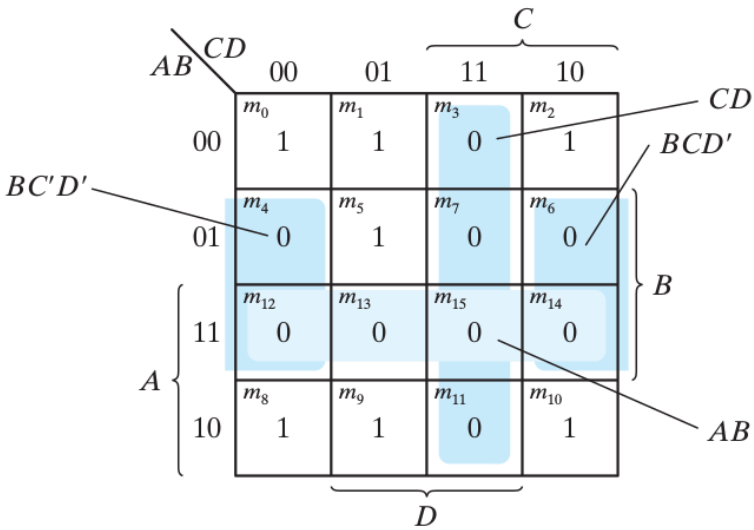
$$F = A'B'E' + BD'E + ACE.$$


A = 0
D
DE
BC
A'B'C'
00
01
11
10
A'BD'E
00
01
11
10
1
1
1
1
1
1
C
B
E
A = 1
D
DE
BC
00
01
11
10
ABD'E
00
01
11
10
1
1
1
1
1
ACE
C
B
E

3.5 Product-of-Sums Simplifications

Example 21 *Simplify the function* $F(A,B,C,D) = \sum(0,1,2,5,8,9,10)$. *(**Hint:** Remember page (22)):*

$$\sum_{1\ of\ f} = \prod_{0\ of\ f} \Longrightarrow f, \quad \sum_{0\ of\ f} = \prod_{1\ of\ f} \Longrightarrow f'.$$

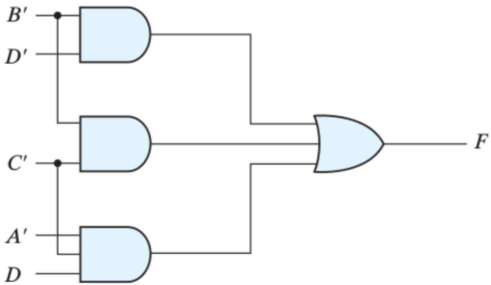


Note: $BC'D' + BCD' = BD'$

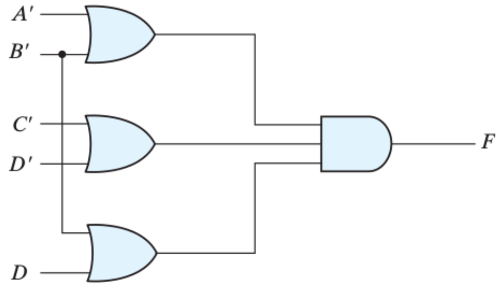
$$F = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D).$$

Two-Level Implementation

Remember: Section 2.6.2



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



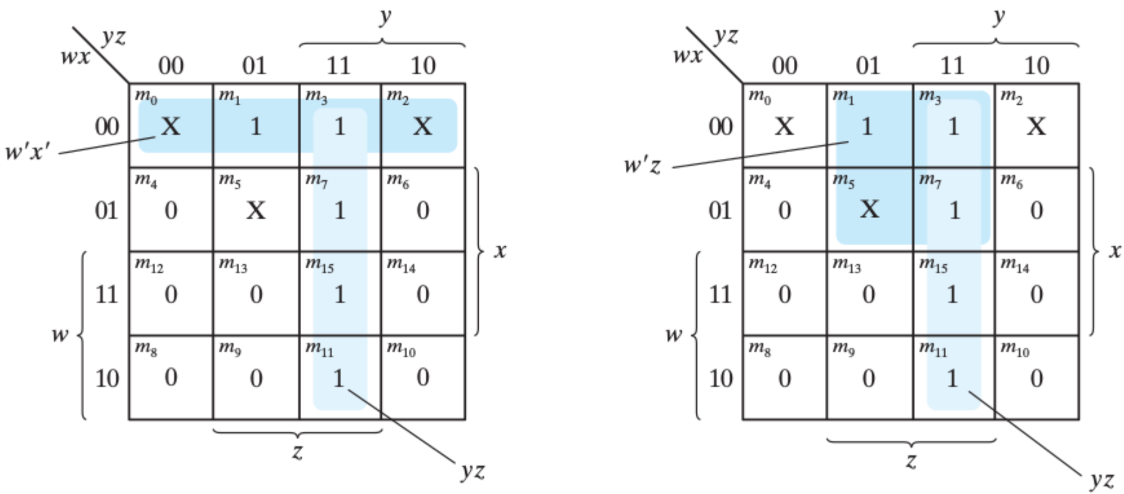
(b) $F = (A' + B')(C' + D')(B' + D)$

3.6 Don't-Care Conditions

- A combination of input that will not happen; so don't care about it. E.g., BCD.
- A combination of input at which we don't care about the output.

Example 22 *Minimize F and find its expression in terms of Minterms:*

$$F(w,x,y,z) = \sum(1,3,7,11,15); \quad d(w,x,y,z) = \sum(0,2,5).$$



(a) $F = yz + w'x'$ (b) $F = yz + w'z$

3.6.1 More Examples on K-Map Simplifications

1			1
1			1

1			
			1

1			1
x			1

1	1		
	1	1	1
		1	1

1	1	1	1
	1	1	1
		1	1
			1

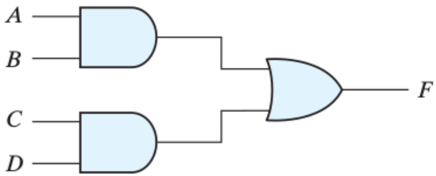
1	1	1	1
	1	1	1
		1	1
x			1

1	1	1	1
1		1	1
1			1
1	1		1

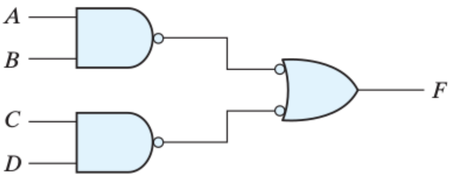
3.7 Two-Level Implementation in NAND and NOR

- Recall that AND, OR, NOT are complete; they can implement any function.
- NAND, NOR can implement them as well.

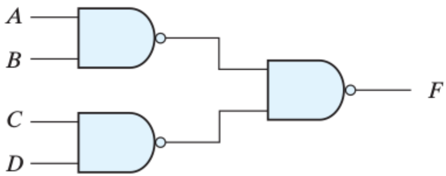
Example: $F = AB + CD$.



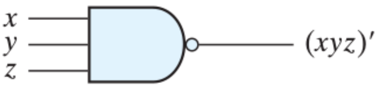
(a)



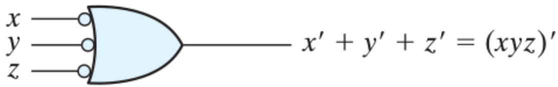
(b)



(c)



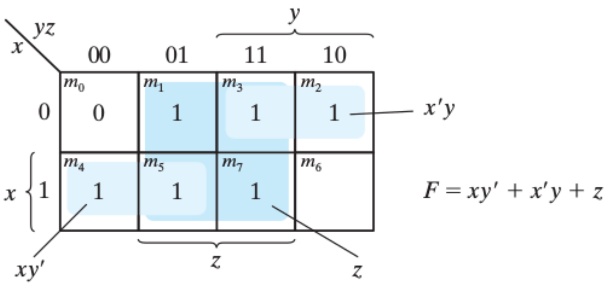
(a) AND-invert



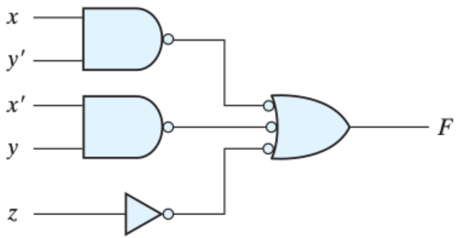
(b) Invert-OR

Example 23 *Minimize and implement, using only* `NAND` *gates the function.*
Hint: remember to use a `NAND` ***for inverter to keep two-level implementations.***

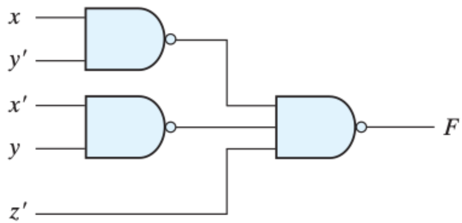
$$F(x,y,z) = \sum(1,2,3,4,5,7)$$



(a)



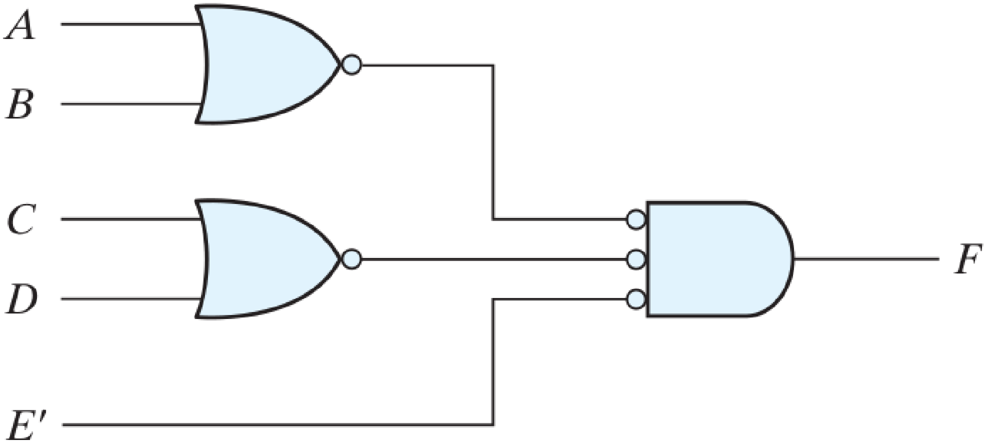
(b)



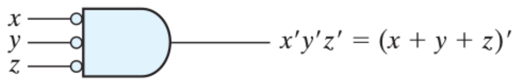
(c)

3.7.1 NOR Implementation

$$F = (A + B)(C + D)E$$



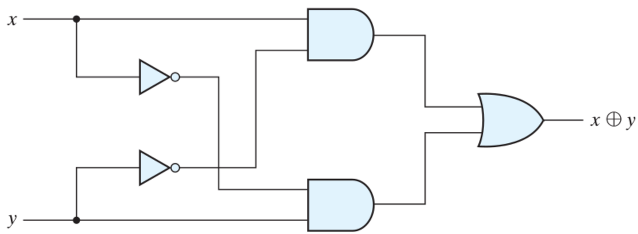
(a) OR-invert



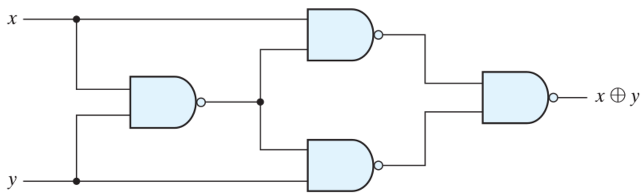
(b) Invert-AND

3.9 Exclusive-OR (XOR) Function: revisit

$$
\begin{aligned}
x \oplus y &= xy' + x'y, \\
x \oplus 0 &= x, \\
x \oplus 1 &= x', \\
x \oplus x &= 0, \\
x \oplus x' &= 1.
\end{aligned}
$$



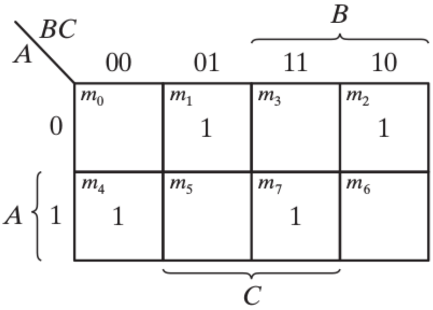
(a) Exclusive-OR with AND–OR–NOT gates



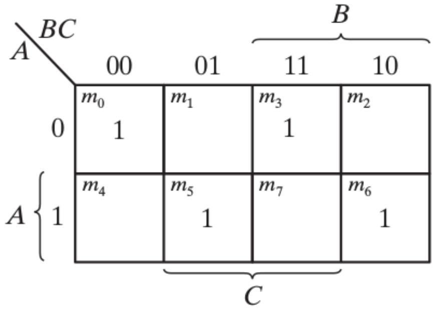
(b) Exclusive-OR with NAND gates

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

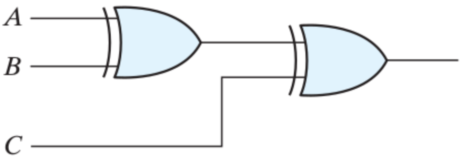
Three-Input XOR: (revise page 33)



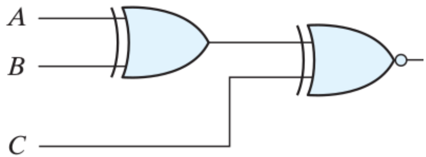
(a) Odd function $F = A \oplus B \oplus C$



(b) Even function $F = (A \oplus B \oplus C)'$



(a) 3-input odd function



(b) 3-input even function

- Of course, could be implemented with two-level ordinary implementation.

Multi-Input XOR: (special offer for Mathematics lovers)

Lemma 24 *The* XOR *function is an odd function for any arbitrary number of bits; i.e.,* $F = A_0 \oplus A_1 \cdots A_n$ *is 1 when* $(A_0 \cdots A_n)$ *have odd number of ones and 0 otherwise.*

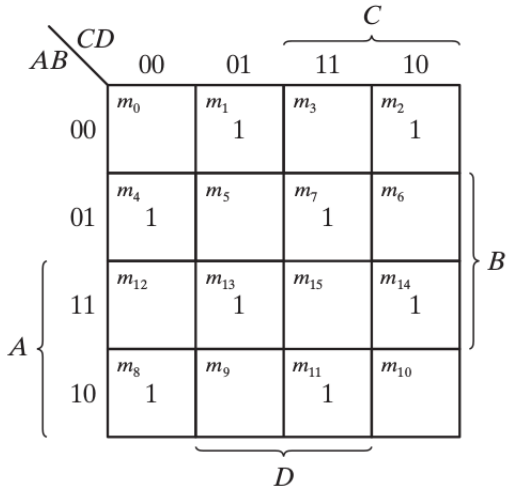
Proof. We prove it by induction. For $n = 1$, it is true from the definition of XOR. Next, suppose the statement is true for some n. Then, for $n+1$:

$$F_{n+1} = A_0 \oplus A_1 \cdots A_{n+1}$$
$$F_{n+1} = F_n \oplus A_{n+1}.$$

F_n	A_{n+1}	F_{n+1}	$(A_0 \cdots A_n)$	$(A_0 \cdots A_{n+1})$
0	0	0	even	even
0	1	1	even	odd
1	0	1	odd	odd
1	1	0	odd	even

■

Implementation of 4-input XOR using NANDs or only XORs

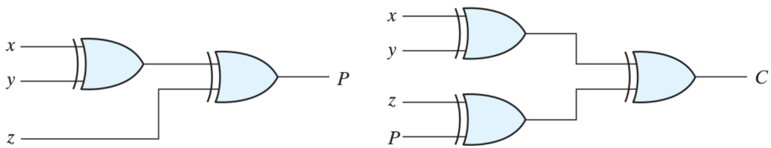


(a) Odd function $F = A \oplus B \oplus C \oplus D$

3.9.1 Parity Generation and Checking

Even-Parity-Generator Truth Table

Three-Bit Message			Parity Bit
x	**y**	**z**	**P**
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



(a) 3-bit even parity generator (b) 4-bit even parity checker

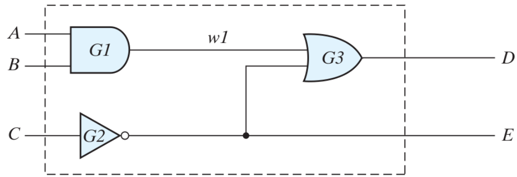
- With this implementation, checker works as generator if $P = 0$.
- **HW:** How many 2-input `XOR`s are needed to implement an even parity generator?

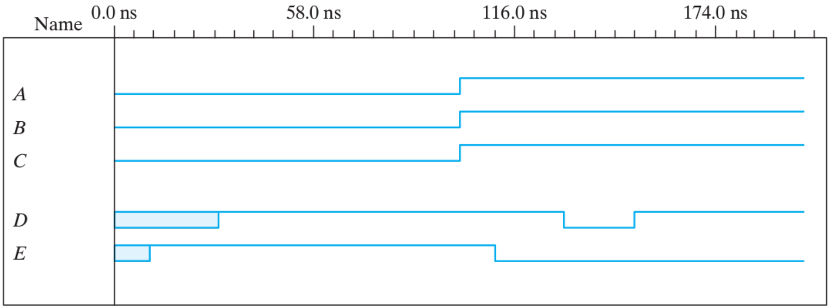
3.10 Hardware Descriptive Language (HDL)

The objective of the following few pages is to MOTIVATE you to self-start NOT to fully teach you

- When circuits become complicated and hard to analyze either on paper or by HW implementation.
- The basic two languages supported by IEEE are VHDL and Verilog
- Verilog CD comes with the book, or you can download the SW and a free 6-months license from: `http://www.syncad.com/`

A Basic Example






```
module Simple_Circuit (A, B, C, D, E);
  output          D, E;
  input           A, B, C;
  wire            w1;

  and             G1 (w1, A, B);
  not             G2 (E, C);
  or              G3 (D, w1, E);
endmodule
```

```
module Simple_Circuit_prop_delay (A, B, C, D, E);
  output          D, E;
  input           A, B, C;
  wire            w1;

  and             #(30) G1 (w1, A, B);
  not             #(10) G2 (E, C);
  or              #(20) G3 (D, w1, E);
endmodule
```

```verilog
// Testbench for Simple_Circuit_prop_delay
`timescale 1 ns / 100 ps
module t_Simple_Circuit_prop_delay;
  wire  D, E;
  reg   A, B, C;

  Simple_Circuit_prop_delay M1 (A, B, C, D, E);  //Instance name required

  initial
    begin
      A = 1'b0; B = 1'b0; C = 1'b0;
      #100 A = 1'b1; B = 1'b1; C = 1'b1;
      #100 $finish;
    end

  initial $monitor($time,,"A = 5b  B= %b  C = %b  w1 = %b D = %b  E = %b", A, B, C, D, M1.w1
      , E);
endmodule
```

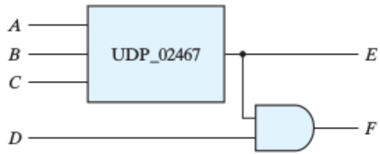
Boolean Expressions

$$E = A + BC + B'D,$$
$$F = B'C + BC'D'.$$

```
// Verilog model: Circuit_Boolean_CA
module Circuit_Boolean_CA(E, F, A, B, C, D);
  output        E, F;
  input         A, B, C, D;

  assign E = A | (B & C) | (~B & D);
  assign F = (~B & C) | (B & ~C & ~D);
endmodule
```

User-Defined Primitives

```
primitive UDP_02467 (D, A, B, C);
  output D;
  input A, B, C;
// Truth table for D = f (A, B, C) = Sum (0, 2, 4, 6, 7);
  table
//      A       B       C       :       D   // Column header
        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


// Verilog model: Circuit instantiation of Circuit_UDP_02467
module Circuit_with_UDP_02467 (e, f, a, b, c, d);
  output        e, f;
  input         a, b, c, d;

  UDP_02467     M0 (e, a, b, c);
  and               (f, e, d); //instance name omitted
endmodule
```



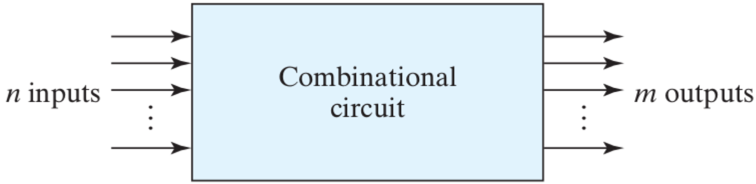
Chapter 4

Combinational Logic

4.1 Introduction: types of logic circuits

- **combinational**: output is function only of input (Ch. 4)
- **sequential**: output is function of input and previous output => MEMORY (Ch. 5–9)

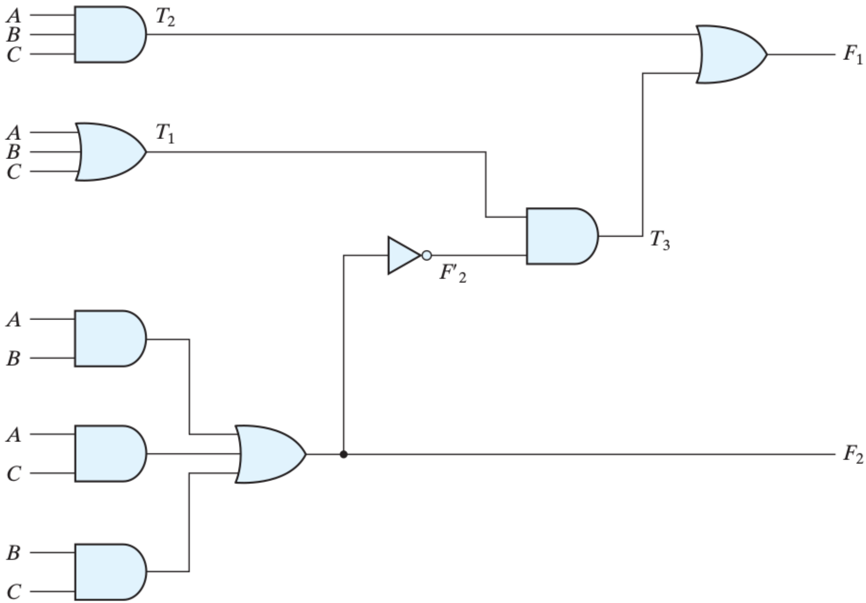
4.2 Combinational Circuits



- Output is a function only of inputs.
- Therefore, outputs can be specified by only truth table or Boolean function.
- Of course, $m \leq 2^{2^n}$.
- In Ch. 4 we will employ the previous chapters to analyze, design, and simplify these circuits.
  - **Analysis**: given a circuit find the output as a function of input.
  - **Design**: given a certain functionality, design the circuit.

4.3 Analysis Procedure

- Make sure that the circuit has no feedback => combinational.
- Label outputs with meaningful names; then start propagating until you reach the output.



$$
\begin{aligned}
F_1 &= T_3 + T_2 \\
&= F_2' T_1 + ABC \\
&= ABC + \\
&\quad (AB + AC + BC)'(A + B + C) \\
&= \vdots \\
&= A'B'C + A'BC' + AB'C' + ABC \\
&= \sum(1,2,4,7).
\end{aligned}
$$

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

4.4 Design Procedure

1. Determine the number of inputs and outputs from the circuit functionality; then draw a block diagram without the internal details. This is exactly similar to function prototype in programming.

2. Derive a truth table.

3. Simplify the expression.

4. Implement the circuit.

4.4.1 Code Conversion Example

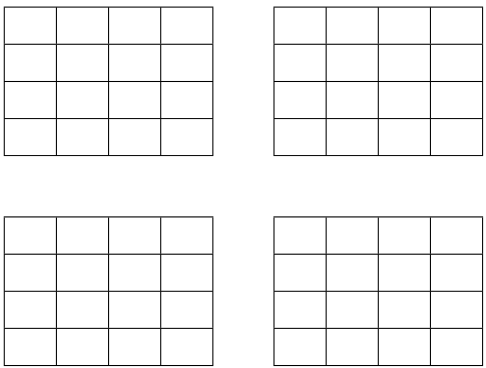
Design a circuit that converts from a BCD code to excess-3 code.
Motivation:

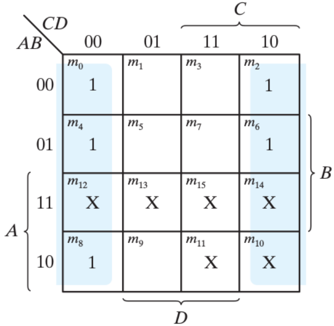
- Some circuits use excess-3 code and we need to feed this circuit with the right input.

- very easy to complement: $9 - x$ is obtained by inverting the digits!

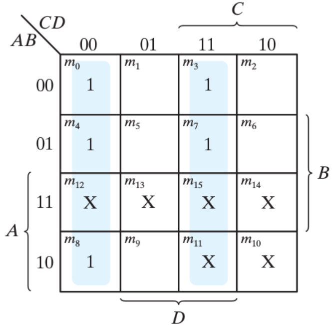
Let's draft it on a clean page, then see the complete solution to save time.

Input BCD				Output Excess-3 Code			
A	***B***	***C***	***D***	***w***	***x***	***y***	***z***
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

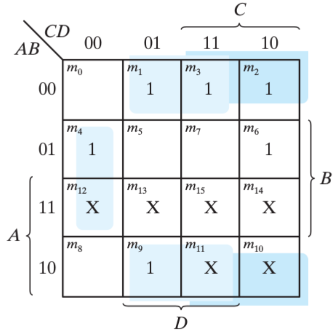




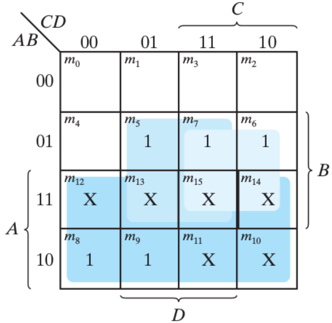
$z = D'$



$y = CD + C'D'$



$x = B'C + B'D + BC'D'$



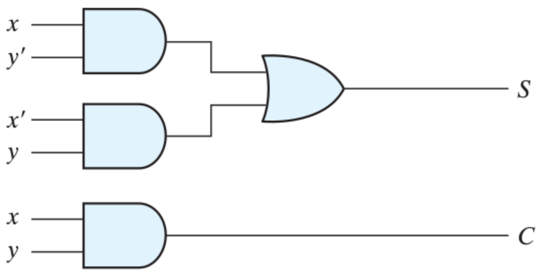
$w = A + BC + BD$

4.5 Binary Adder-Subtractor

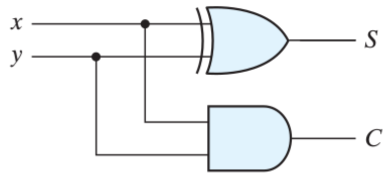
4.5.1 Half Adder

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



(a) $S = xy' + x'y$
$C = xy$

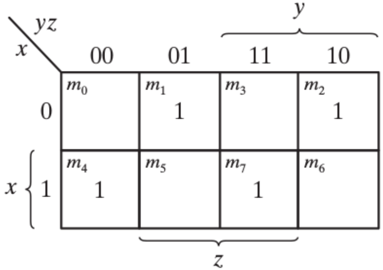


(b) $S = x \oplus y$
$C = xy$

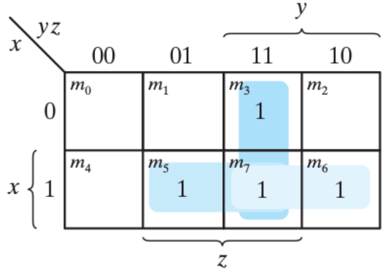
4.5.2 Full Adder

Full Adder

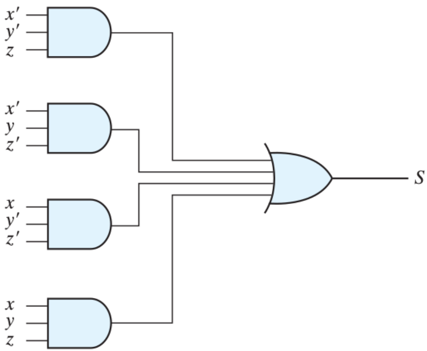
x	*y*	*z*	*C*	*S*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

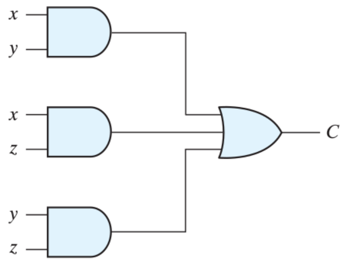


(a) $S = x'y'z + x'yz' + xy'z' + xyz$

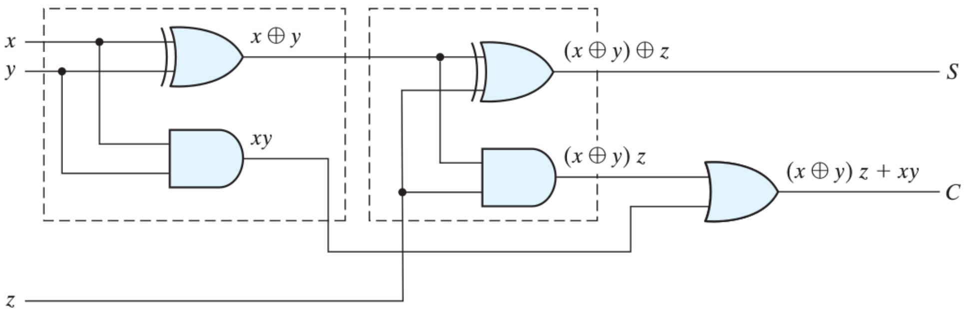


(b) $C = xy + xz + yz$





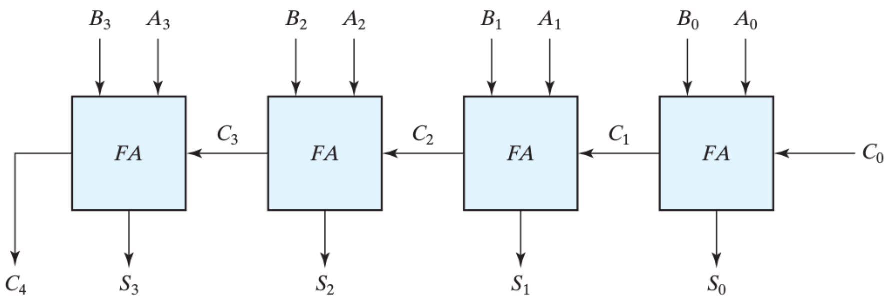
4.5.3 Implementing `FA` using ONLY `HA` => modular design => very interesting



$$
\begin{aligned}
C_3 &= 0 \\
S &= S_2 = S_1 \oplus z \\
&= x \oplus y \oplus z \\
C &= S_3 = C_1 + C_2 \\
&= xy + S_1 z \\
&= xy + (x \oplus y) z \\
&= \vdots \\
&= \sum(3,5,6,7)
\end{aligned}
$$

4.5.4 Binary Adder: => more modular design => wonderful!

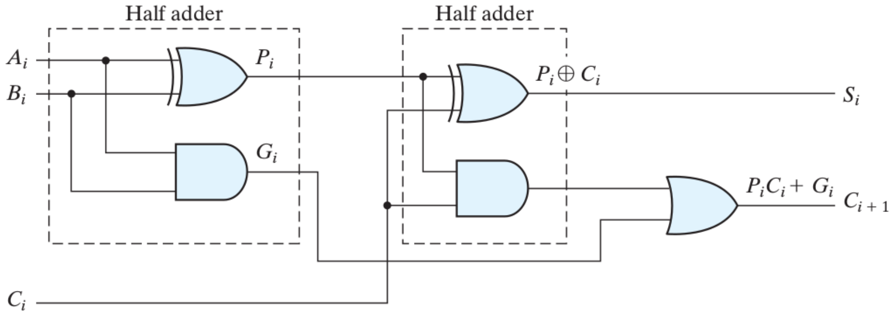
Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}



Hint:

- without observing this modularity, it would be extremely difficult to design, e.g., 8-bit binary adder!
- carry subscript here is different from our previous example.

4.5.5 Carry Propagation: complexity-speed trade off!



$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

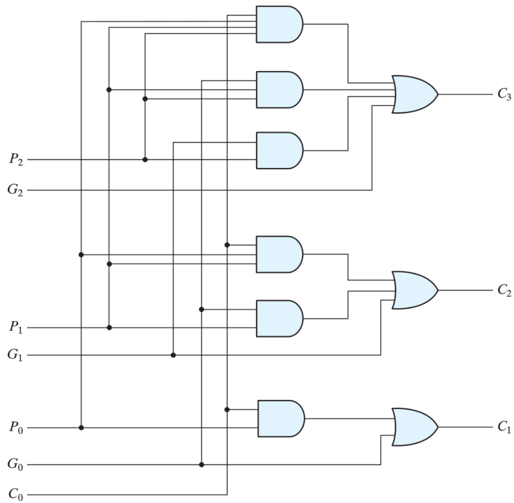
$$S_i = P_i \oplus C_i$$

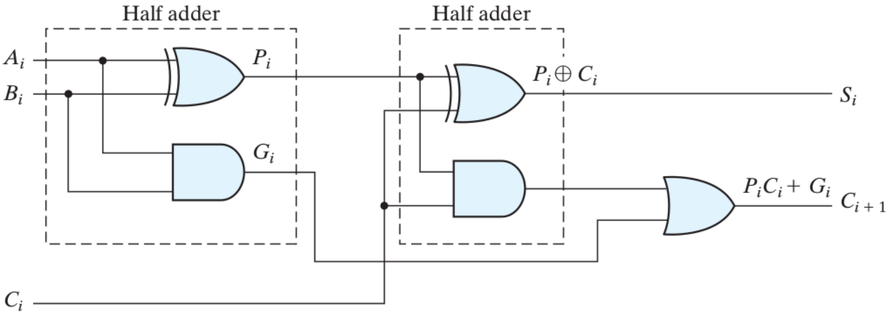
$$C_{i+1} = G_i + P_i C_i$$

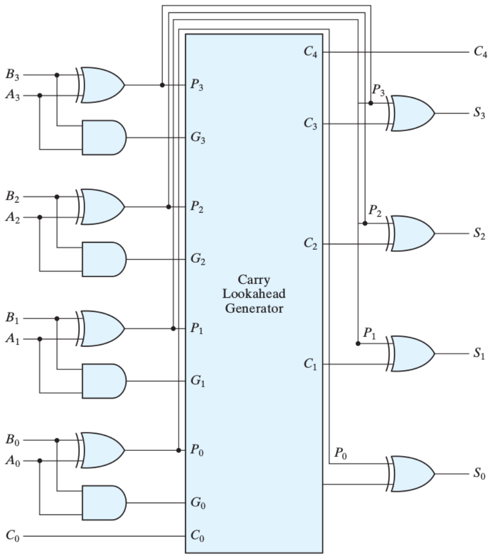
$$C_1 = G_0 + P_0 C_0$$

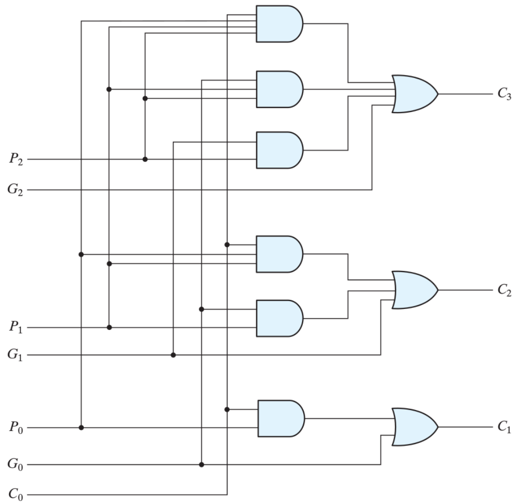
$$\begin{aligned} C_2 &= G_1 + P_1 C_1 \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$

$$\begin{aligned} C_3 &= G_2 + P_2 C_2 \\ &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_o C_o. \end{aligned}$$




Half adder
Half adder
A_i
B_i
P_i
G_i
$P_i \oplus C_i$
S_i
$P_iC_i + G_i$
C_{i+1}
C_i


B_3
A_3
B_2
A_2
B_1
A_1
B_0
A_0
C_0
P_3
G_3
P_2
G_2
P_1
G_1
P_0
G_0
C_0
Carry Lookahead Generator
C_4
C_3
C_2
C_1
C_4
P_3
S_3
P_2
S_2
P_1
S_1
P_0
S_0


P_2
G_2
P_1
G_1
P_0
G_0
C_0
C_3
C_2
C_1

4.5.6 Binary Subtractor

From Appendix A (Binary Number System) we already know that for any n-bit numbers:

$$
\begin{aligned}
A - B &= A - B + 2^n - 2^n \\
&= \big(A + (2^n - 1 - B) + 1\big) - 2^n \\
&= \big(A + (1\text{'s comp of } B) + 1\big) - 2^n \\
&= (A + B' + 1) - 2^n.
\end{aligned}
$$

6	0110	0110
2	0010	1101
		1
		10100

- For signed numbers, we use 2's Comp.
- Over-flow: when last two carries are different.

Block Diagram level:

FAs level

4.5.7 Binary Adder-Subtractor

Back to truth table and K-Map:

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

x	y	f
0	0	0
0	1	1
1	0	A
1	1	B

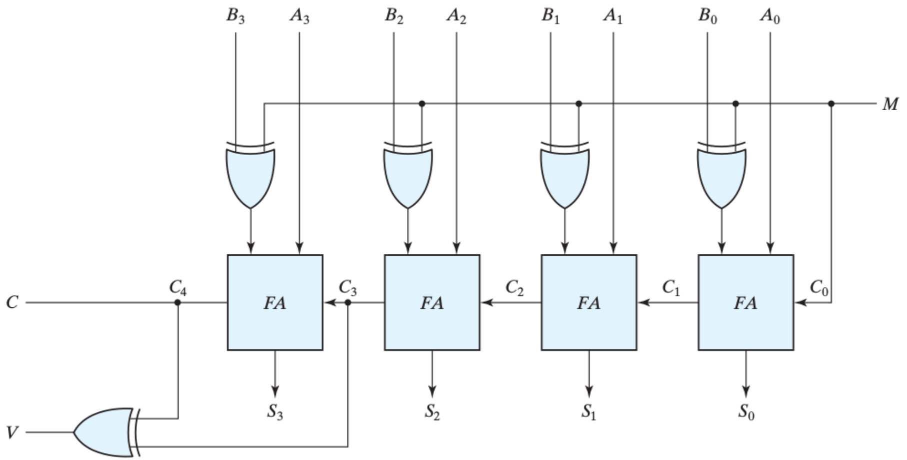
0	1
A	B

0	1
A	x

0	0	1	1
x	A	x	x

M	Op	x_i	y_i	C_0
0	$A+B$	A_i	B_i	0
1	$A+B'+1$	A_i	B_i'	1

$$x_i = A_i$$
$$y_i = M'B_i + MB_i' = M \oplus B_i$$
$$C_o = M.$$

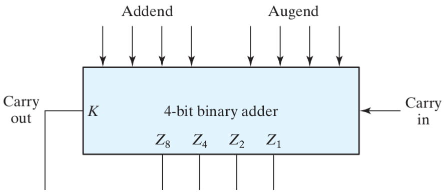


4.6 Decimal Adder

Design 1: very goofy
5 functions, each:
$2^9 = 512$ rows!

cin	b3	b2	b1	b0	a3	a2	a1	a0	cout	s8	s4	s2	s1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
				⋮							⋮		
				X							X		

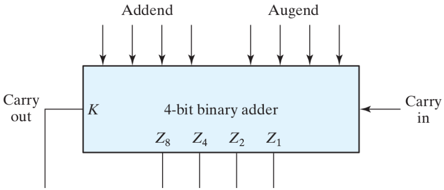
Design 2: smarter but not yet



Derivation of BCD Adder

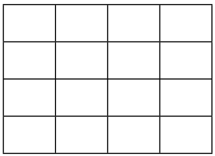
Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

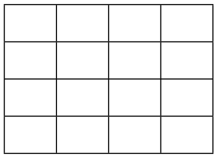
Design 3: smartest!



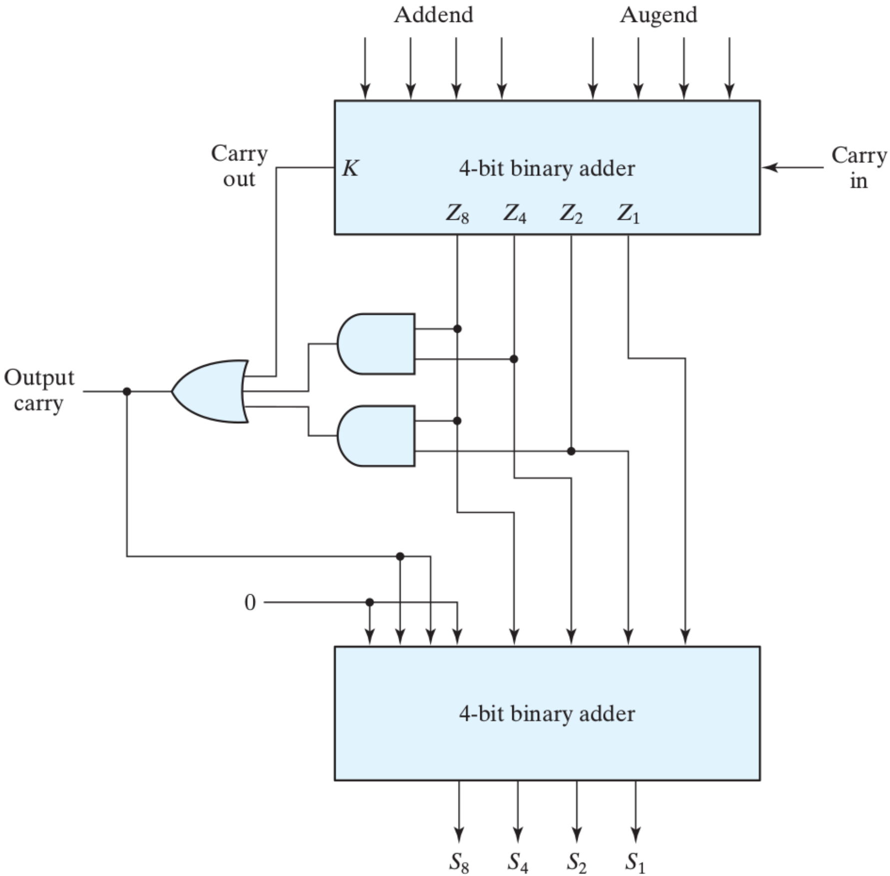
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19





C	a b c d
0	0 0 0 0
1	0 1 1 0


Addend
Augend
Carry out
K
4-bit binary adder
Carry in
Z_8 Z_4 Z_2 Z_1
Output carry
0
4-bit binary adder
S_8 S_4 S_2 S_1

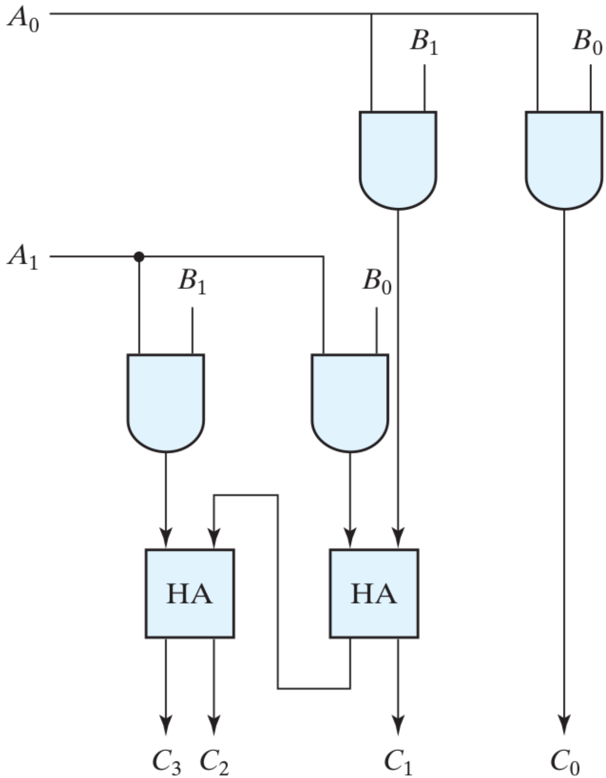
4.7 Binary Multiplier

4.7.1 2-bit x 2-bit Multiplier

Design 1: truth table

Design 2: Let's think algorithmic

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0

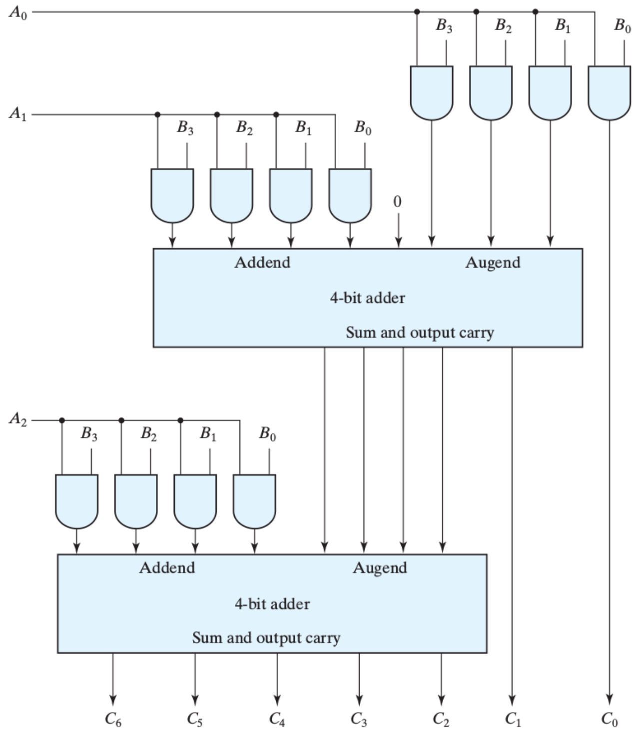


4.7.2 3-bit x 4-bit Multiplier: How many bits?

Design 1: truth table (very goofy)

Design 2: smart

			B_3	B_2	B_1	B_0
				A_2	A_1	A_0
0	0	0	A_0B_3	A_0B_2	A_0B_1	A_0B_0
0	0	A_1B_3	A_1B_2	A_1B_1	A_1B_0	0
0	A_2B_3	A_2B_2	A_2B_1	A_2B_0	0	0
C_6	C_5	C_4	C_3	C_2	C_1	C_0



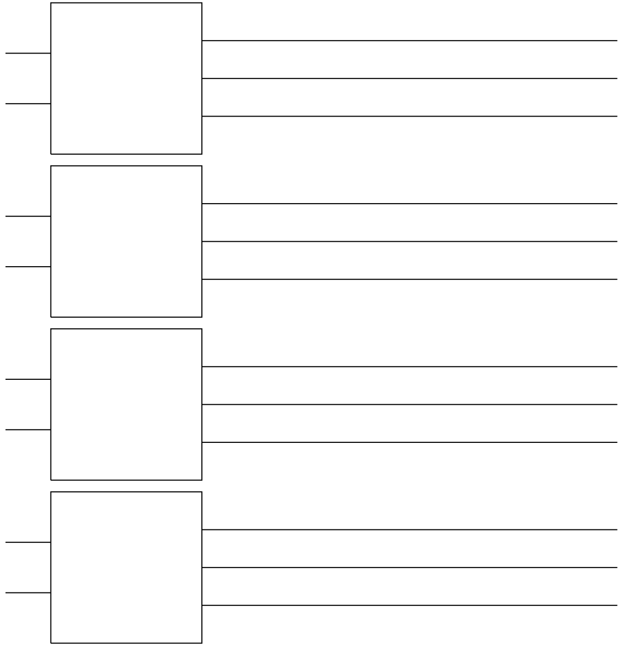
4.8 4-bit x 4-bit Magnitude Comparator

Design 1: truth table (very goofy).

Design 2: Let's go modular. design a 1-bit x 1-bit.

A_0	B_0	L_0	S_0	E_0
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

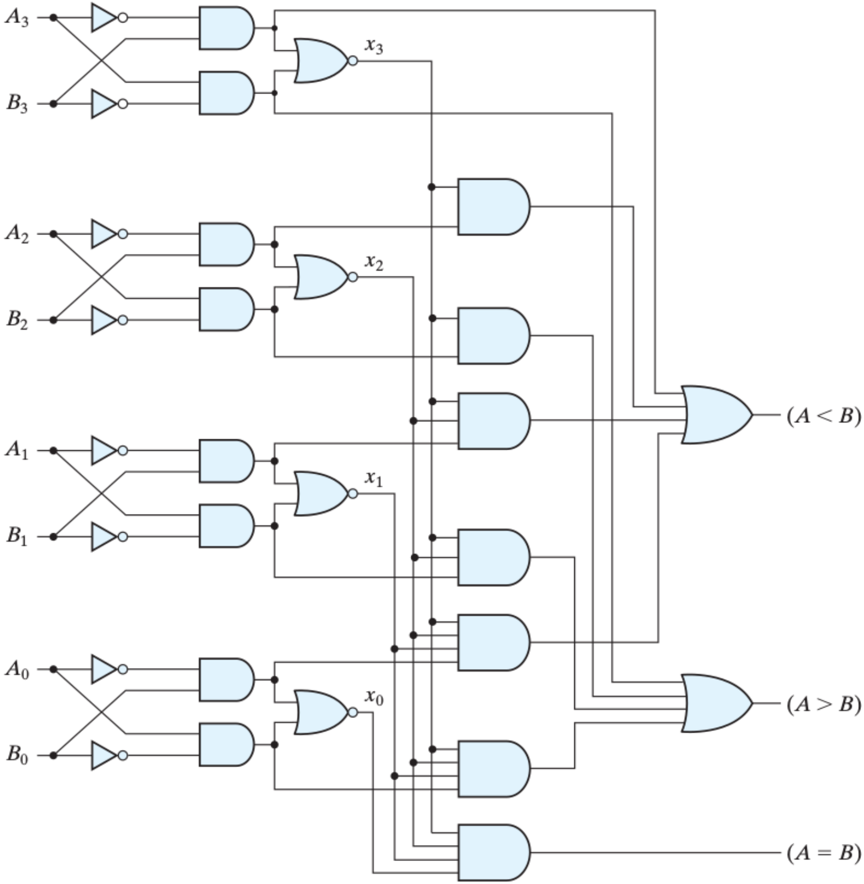
$$L_0 = A_0 B_0',$$
$$S_0 = A_0' B_0,$$
$$E_0 = A_0' B_0' + A_0 B_0$$
$$= (L_0 + S_0)'.$$



For 4-bit comparator:

```
/* Algorithm for the logic: L = A > B
   A3 A2 A1 A0
   B3 B2 B1 B0
*/
L=(A3>B3)||
  (A3==B3)&&(A2>B2)||
  (A3==B3)&&(A2==B2)&&(A1>B1)||
  (A3==B3)&&(A2==B2)&&(A1==B1)&&(A0>B0)
```

$$L = L_3 + E_3 L_2 + E_3 E_2 L_1 + E_3 E_2 E_1 L_0$$
$$S = S_3 + E_3 S_2 + E_3 E_2 S_1 + E_3 E_2 E_1 S_0$$
$$E = E_3 E_2 E_1 E_0$$


A_3
B_3
x_3
A_2
B_2
x_2
A_1
B_1
x_1
A_0
B_0
x_0
$(A < B)$
$(A > B)$
$(A = B)$

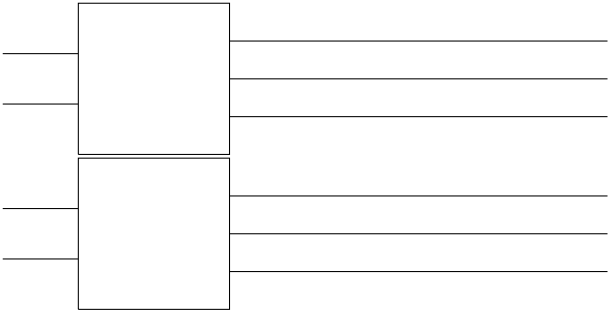
Example 25 *Design an 8-bit comparator.*

Design 1: *using the 1-bit comparator very similar to what we have done.*

Design 2: *using the 4-bit comparator.*

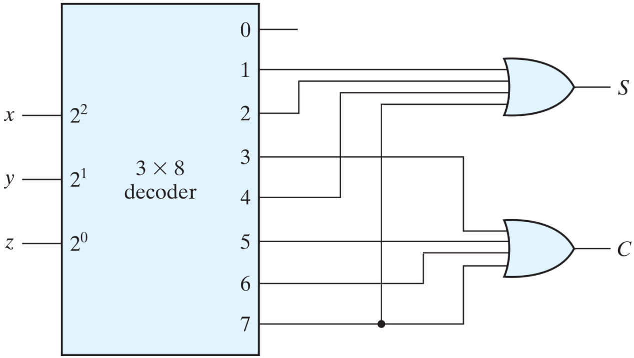
$$\begin{array}{cccc|cccc} A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 \\ B_7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 \end{array}$$

$$L = L_1 + E_1 L_0,$$
$$S = S_1 + E_1 S_0,$$
$$E = E_1 E_0.$$



Is design 2 slower (propagation delay)? HW

4.9 $n \times 2^n$ Decoder: $D_i = m_i$



$$S = \sum(1,2,4,7)$$

$$C = \sum(3,5,6,7)$$

$$K = \sum(0,1,2,3,4,5) \qquad (\#\ \text{minterms} > 2^n/2)$$

$$K' = \sum(6,7)$$

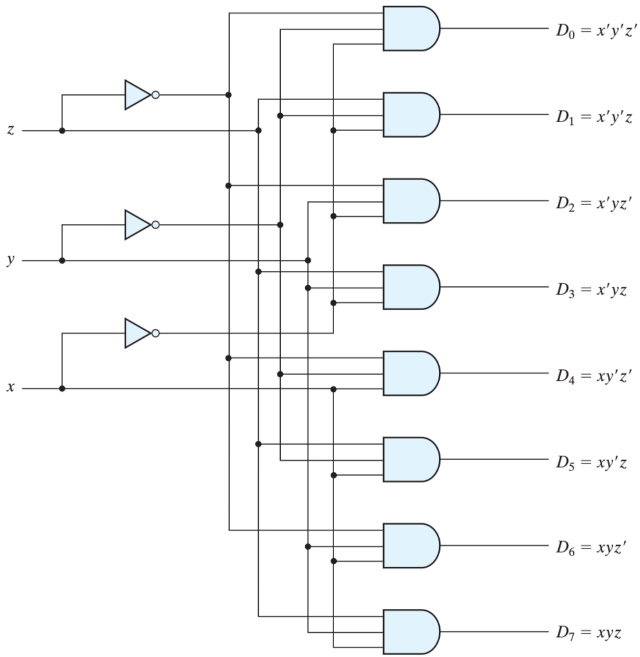
inverted output and enabled decoders:

E	D_i
0	0
1	m_i

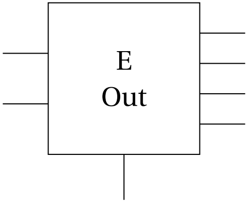
$$D_i = Em_i$$

Truth Table of a Three-to-Eight-Line Decoder

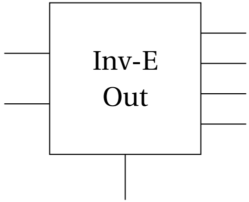
Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



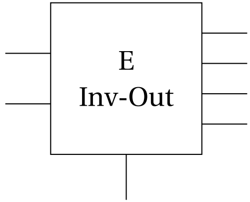
Therefore, we have:



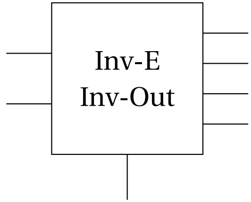
E	Op.	D_i
0	Dsb	0
1	Enb	m_i



E	Op.	D_i
0	Enb	m_i
1	Dsb	0



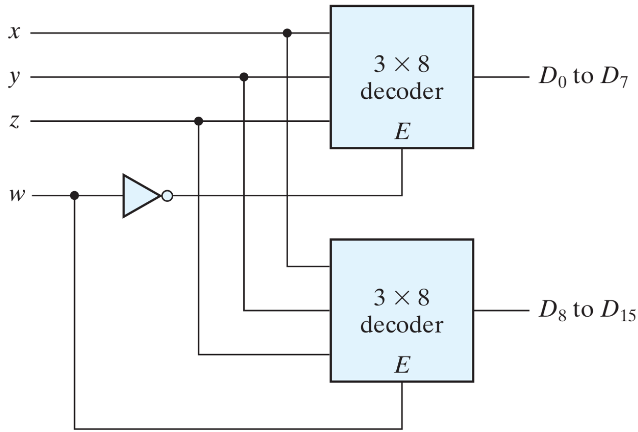
E	Op.	D_i
0	Dsb	1
1	Enb	M_i



E	Op.	D_i
0	Enb	M_i
1	Dsb	1

Example 26 *Design a 4x16 decoder using only 3x8 decoders.* ***Hint:*** *take care of LSB and MSB.*

w	x	y	z
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1



we will see after studying multiplexer the connection to decoders!

4.10 2^n to n Encoders

Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

- the opposite operation of decoders.
- we rely on no $D_i = D_j = 1,\ i \neq j$.
- no handling to $D_i = 0 \forall i$

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7.$$

4.10.1 Priority Encoder: 4 x 2

#	D_3	D_2	D_1	D_0	x	y	v
0	0	0	0	0	X	X	0
1	0	0	0	1	0	0	1
2-3	0	0	1	X	0	1	1
4-7	0	1	X	X	1	0	1
8-15	1	X	X	X	1	1	1

$$v = D_3 + D_2 + D_1 + D_0$$

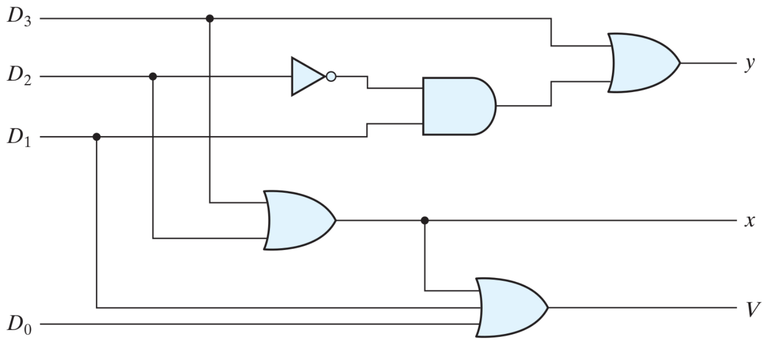
X	0	0	0
1	1	1	1
1	1	1	1
1	1	1	1

$$x = D_3 + D_2$$

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

$$y = D_3 + D_1 D_2'$$

- D_3 is more prior than D_0.
- Xs in inputs are NOT don't care.
- V is 1 to indicate "Valid".



Example 27 *Encode the comparator outputs L, S, E to X, Y such that:*

cond.	*X*	*Y*
$A > B$	1	0
$A = B$	1	1
$A < B$	0	1

4.11 Multiplexers: $2^n \times 1$

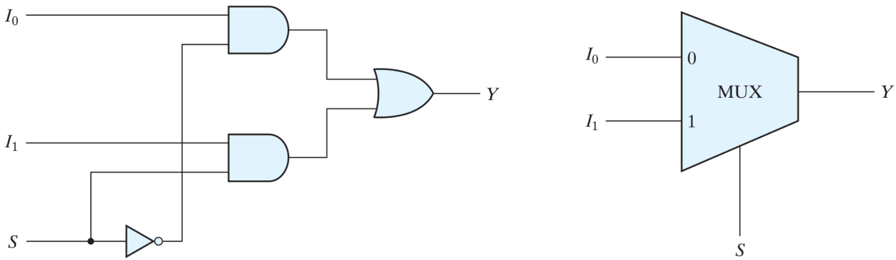
How to use n selectors to select only one among 2^n inputs?

4.11.1 Two-to-one-line Mux

Motivation: How to select between two variables (or more generally: two things)?

S	Y
0	I_0
1	I_1

$$Y = S' I_0 + S I_1$$

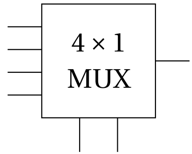


(a) Logic diagram (b) Block diagram

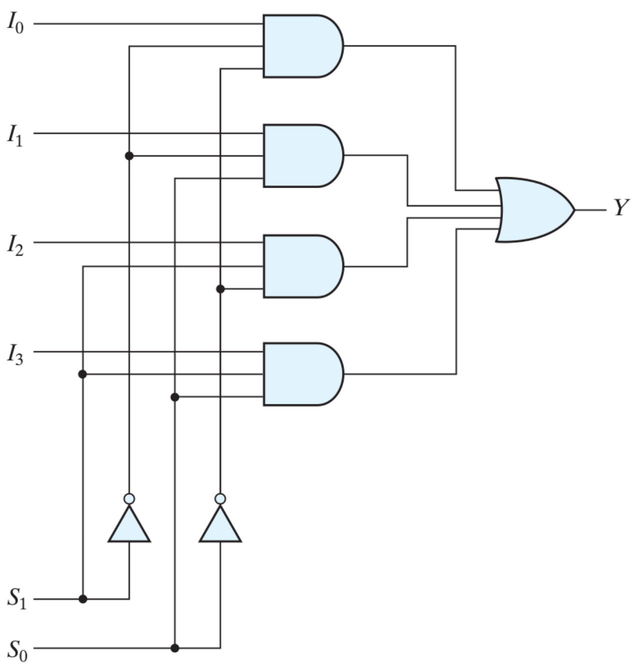
- Of course, we can make it Enable or Inv-Enable MUX; how?

4.11.2 Four-to-one-line Mux

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



$$
\begin{aligned}
Y &= I_0 S_1' S_0' + I_1 S_1' S_0 + I_2 S_1 S_0' + I_3 S_1 S_0 \\
&= I_0 m_0 + I_1 m_1 + I_2 m_2 + I_3 m_3 \\
&= \sum_{i=0}^{2^n-1} I_i m_i .
\end{aligned}
$$



- $2^n \times 1$ MUX is an extension to $n \times 2^n$ DEC (check Section 4.9)
- In Dec. :
  - All I_i where just the enable E.
  - No `OR` Gate.

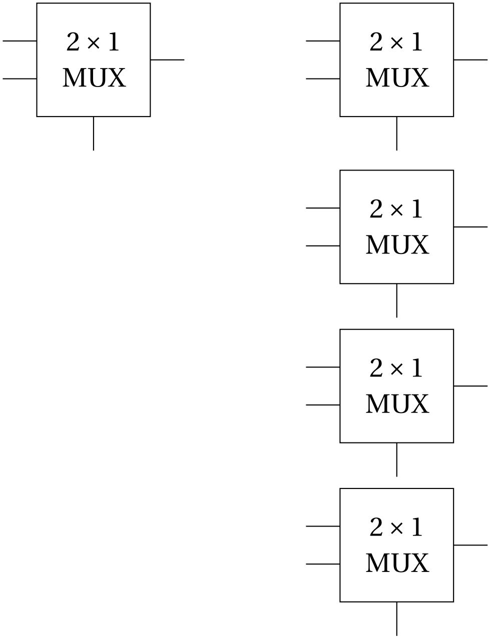
4.11.3 Quadrable two-to-one-line Mux (block-reuse design)

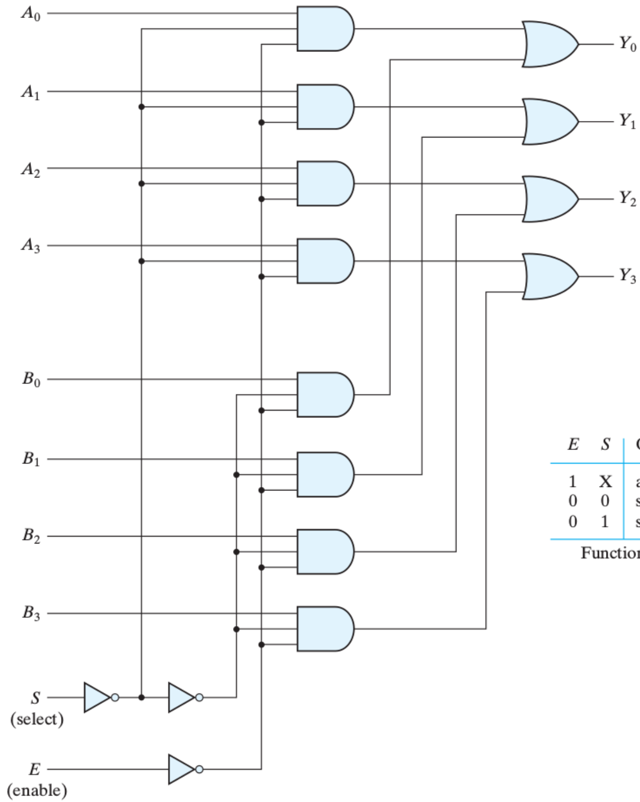
Design 1:

$Y_i = S' A_i + S B_i$

S	Y_3	Y_2	Y_1	Y_0
0	A_3	A_2	A_1	A_0
1	B_3	B_2	B_1	B_0

Design 2: use four 2×1 MUXs





E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

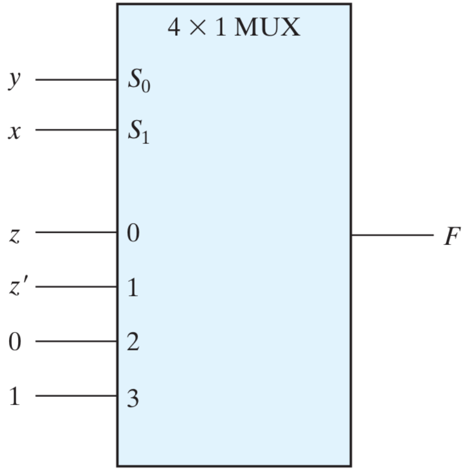
4.11.4 Boolean Function Implementation

- Given: $F(x, y, z) = \sum(1, 2, 6, 7)$
- Implement it using 4×1 MUX.

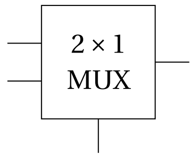
- Could we use 2×1 MUX?
  - For $x = 0$
    $F = I_0 = y'z + yz'$,
  - For $x = 1$
    $F = I_1 = yz' + yz = y$.

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

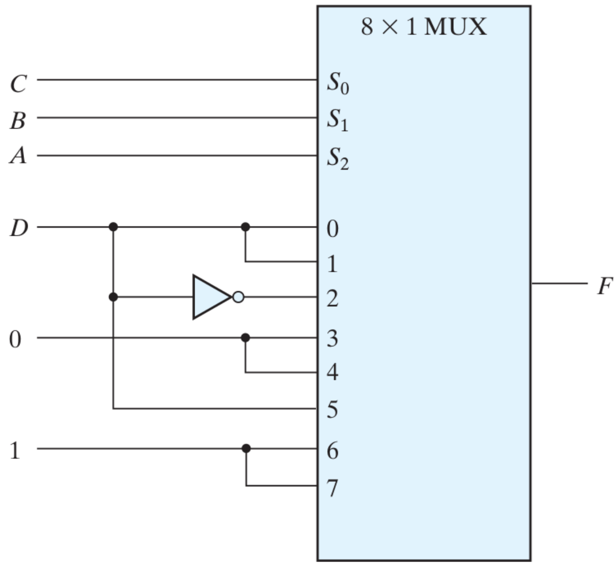


(b) Multiplexer implementation



Example 28 : *Implement the function* $F(A,B,C,D)=\sum(1,3,4,11,12,13,14,15)$ *using* 8×1 *MUX.*

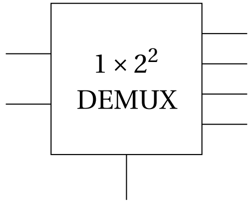
A	B	C	D	F	
0	0	0	0	0	$F=D$
0	0	0	1	1	
0	0	1	0	0	$F=D$
0	0	1	1	1	
0	1	0	0	1	$F=D'$
0	1	0	1	0	
0	1	1	0	0	$F=0$
0	1	1	1	0	
1	0	0	0	0	$F=0$
1	0	0	1	0	
1	0	1	0	0	$F=D$
1	0	1	1	1	
1	1	0	0	1	$F=1$
1	1	0	1	1	
1	1	1	0	1	$F=1$
1	1	1	1	1	



4.12 Demultiplexers (DEMUX)

- Analogous to DEC-ENC could we have MUX-DEMUX?
- $2^n \times 1$ MUX => 1×2^n DEMUX?
- Only the output $F_i = I$ when selectors have a minterm value of i

$$F_i = I m_i$$



- **HENCE**: 1×2^n DEMUX is nothing but Enabled $n \times 2^n$ DEC, where the “Enable” is the input! (see Section 4.9)

Homework

Chapter 5

Synchronous Sequential Logic

5.1 Introduction

- Remember the introduction of Section 4.1.
- In “Sequential” the new output is a function of its current value (“state”) and the input.

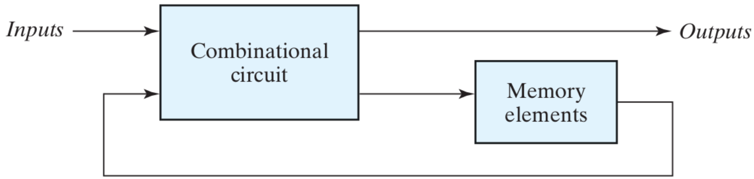
5.2 Sequential Circuits

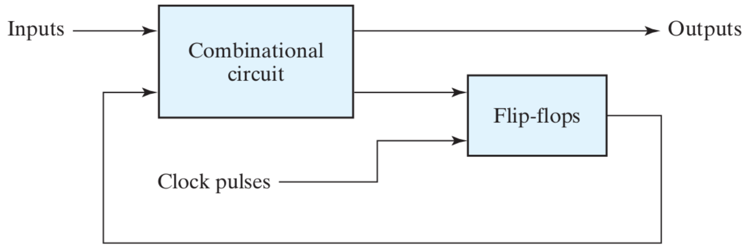
Asynchronous : Ch. 9

- Element delay determines speed.
- Basic elements are "Latches".
- Studied only if time permits.

Synchronous : Ch. (5–8)

- Clock frequency (1GHz, 2GHz, etc.) determines speed.
- Basic elements are "Flip Flops" (but they are constructed from Latches)
- Clock design is an "Electronics" topic.
- Ch. 5–7 will be studied
- Ch. 8 is the introduction to "Computer Organization" course.





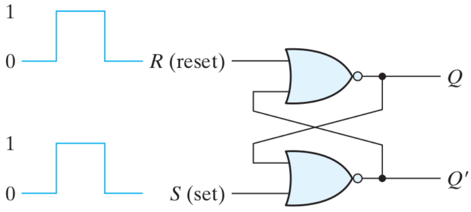
(a) Block diagram

(b) Timing diagram of clock pulses

5.3 Storage Elements: Latches (for asynchronous circuits)

5.3.1 SR Latch

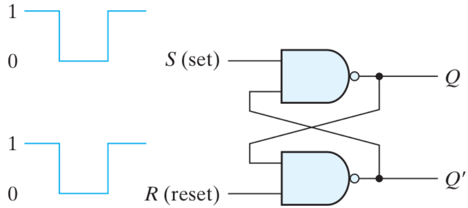
- $Q(t+1)\left(Q(t),\ S,\ R\right)$
- We will prove Q and Q'.
- Value of Q => "state".
- Q = 1 => "set".
- Q = 0 => "reset".
- SR Latch with `NAND` implementation ($S'R'$ Latch)
- SR Latch with Enable => "when"



(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

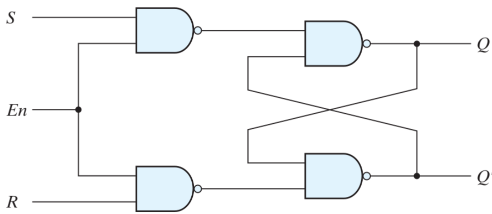
(b) Function table



(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table



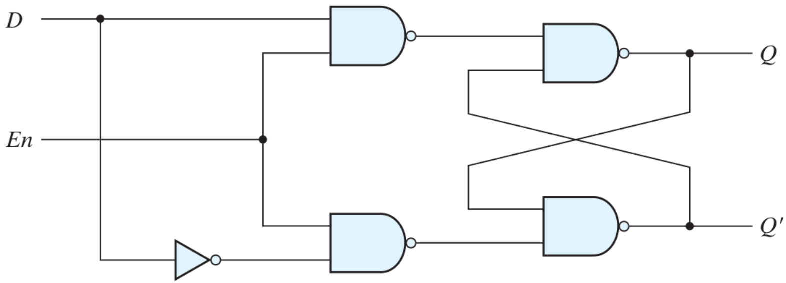
(a) Logic diagram

En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

5.3.2 D Latch (Transparent Latch)

- A remedy to the forbidden condition => D Latch.
- D passes directly and stores into Q (transparent).
- What is the difference between D Latch and simply a wire!! => (memory)

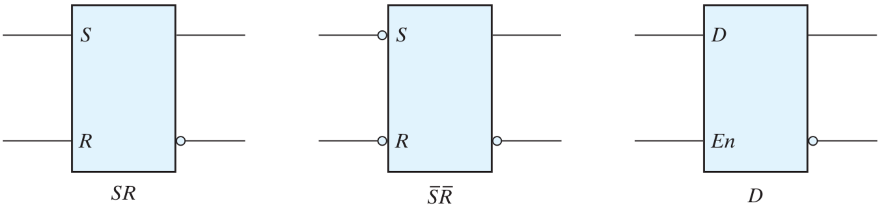


(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

Graphic Symbols



5.4 Storage Elements: Flip Flops (for synchronous circuits)

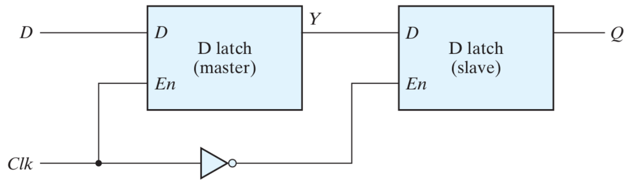
- Problem with Enabled D Latch is output change during the whole positive Enable period!
- Response at ONLY edge Enabled is therefore required => CLOCK (positive or negative).

(a) Response to positive level

(b) Positive-edge response

(c) Negative-edge response

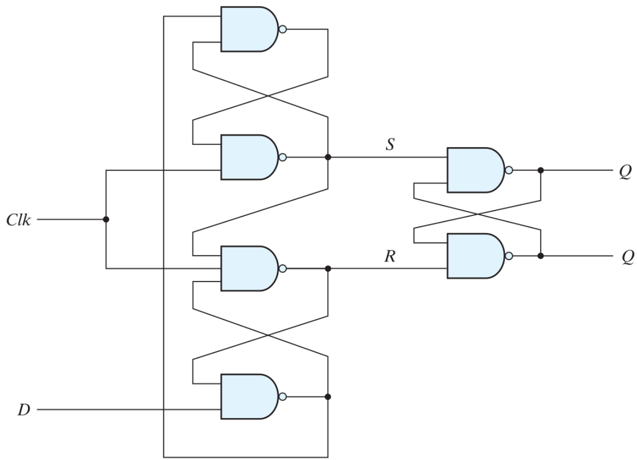
5.4.1 Edge-Triggered D Flip-Flop



From Latches:

- Master-Slave Flip-Flip
- This is negative-edge triggered.

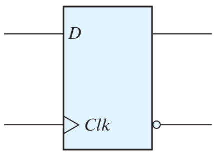
From Gates:



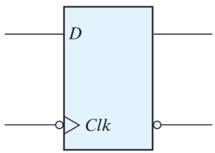
General Notes:

- Either way, clock frequency (hence system speed) is confined to gate delay.
- $f = 1/T$
- The function table is the same as D-Latch with replacing En with Clk

Clk	D	Next State of Q
0/1	X	No change
$\uparrow$	0	Q=0 reset
$\uparrow$	1	Q=1 set



(a) Positive-edge



(a) Negative-edge

5.4.2 Other Flip-Flops: JK and T Flip-Flops

- Let's understand them right now.
- How to reach the design of these and other Flip-Flops and convert from any Flip-Flop to others is detailed in Section 5.9

JK Flip-Flop (from D Flip-Flop)

Input Equation

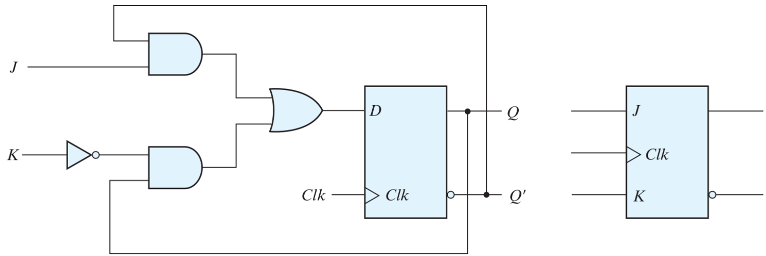
$$D = JQ' + K'Q$$

Characteristic Equation

$$\begin{aligned} Q(t+1) &= D \\ &= JQ'(t) + K'Q(t). \end{aligned}$$

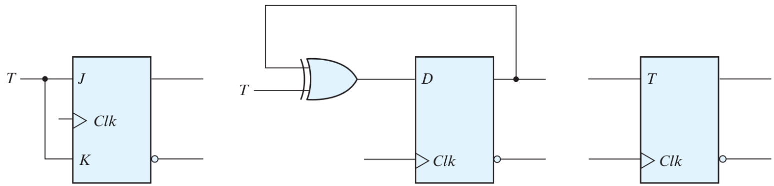
Characteristic Table

J	K	$Q(t+1)$	
0	0	$Q(t)$	(no change)
0	1	0	(reset)
1	0	1	(set)
1	1	$Q'(t)$	(complement)



(a) Circuit diagram (b) Graphic symbol

T Flip-Flop (from D or JK Flip-Flops)



(a) From JK flip-flop (b) From D flip-flop (c) Graphic symbol

Input Equation

$$D = T \oplus Q$$

Characteristic Equation

$$\begin{aligned} Q(t+1) &= D \\ &= T \oplus Q(t). \end{aligned}$$

Characteristic Table

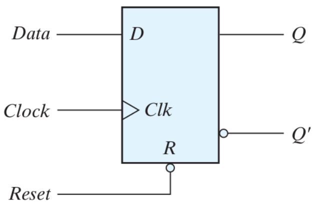
T	$Q(t+1)$	
0	$Q(t)$	(no change)
1	$Q'(t)$	(complement)

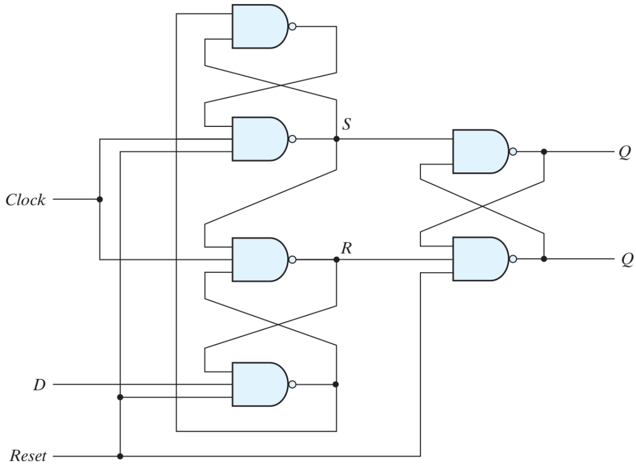
5.4.3 Direct Inputs (asynchronous reset)

- When power turns on, "state" is unknown.
- We need instant/direct/asynchronous reset.
- Here, active-low reset (similar to the Inv-En).

Function Table

R	Clk	D	Q	Q'
0	X	X	0	1
1	$\uparrow$	0	0	1
1	$\uparrow$	1	1	0





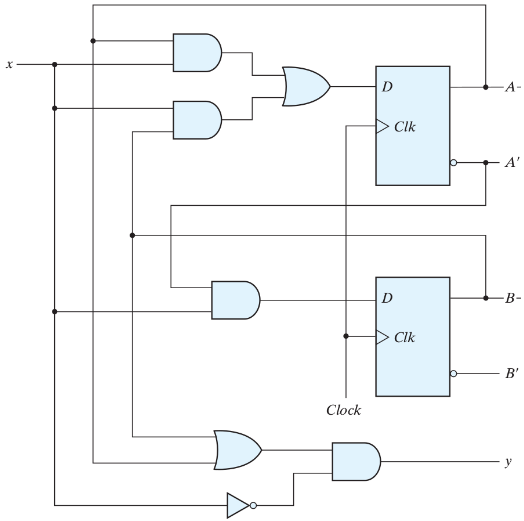
5.5 Analysis of Clocked Sequential Circuits (Synchronous Circuits)

We represent a sequential circuit (as well as combinational) by:

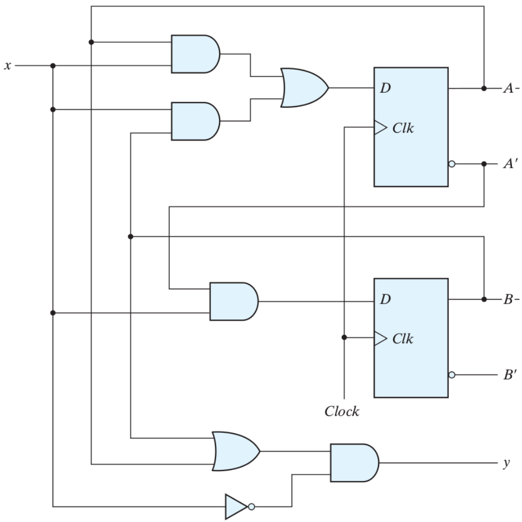
- circuit drawing.
- equation ("algebraic expression", "state equation", "transition equation"),
- table ("state table", "transition table") same as "truth table", "characteristic table", "functional table"

in addition:

- state diagram ("Finite State Machine (FSM)", "transition diagram"),
- timing diagram.



Example 29 (Analysis with D flip-flop) .



1- Input Equation (appropriate notation):

$D_A = Ax + Bx, \qquad D_B = A'x.$

2- Flip-Flop Characteristic Equation:

$A(t+1) = D_A, \qquad B(t+1) = D_B$

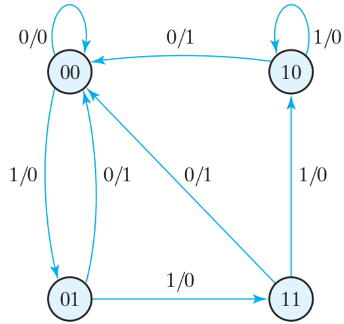
3- Output and State Equations:

$A(t+1) = Ax + Bx, \quad B(t+1) = A'x,$

$y = (A+B)x'.$

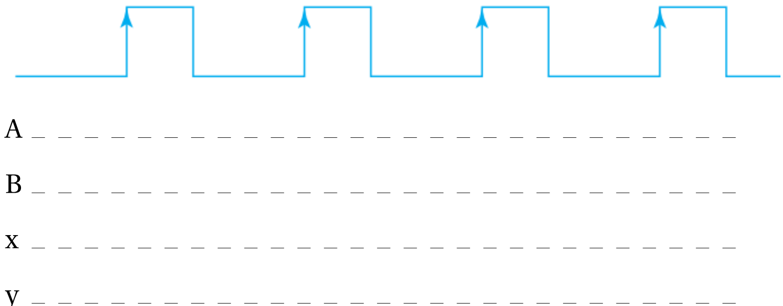
4,5- State Table and State Diagram (FSM)

Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

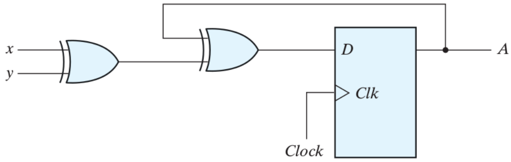


Present State		Next State				Output	
		***x* = 0**		***x* = 1**		***x* = 0**	***x* = 1**
A	***B***	***A***	***B***	***A***	***B***	***y***	***y***
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

6- Timing Diagram: $A(0) = 0,\ B(0) = 0,\ x = (1, 0, 1).$



Example 30 (a simpler example) .



1- Input Equation (appropriate notation):

$D_A = A \oplus x \oplus y$

2- Flip-Flop Characteristic Equation:

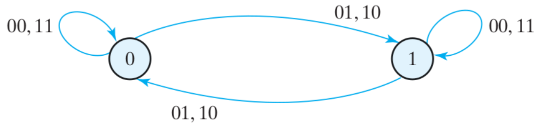
$A(t+1) = D_A$

3- Output and State Equations:

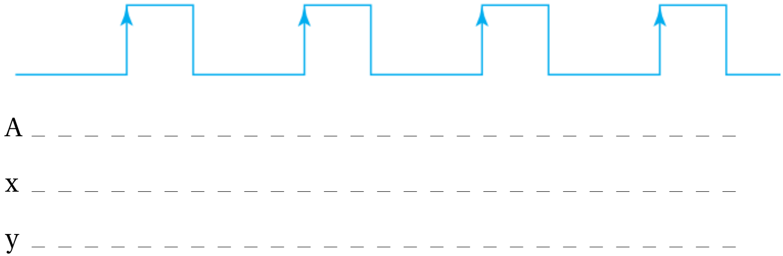
$A(t+1) = A \oplus x \oplus y$

4,5- State Table and State Diagram (FSM)

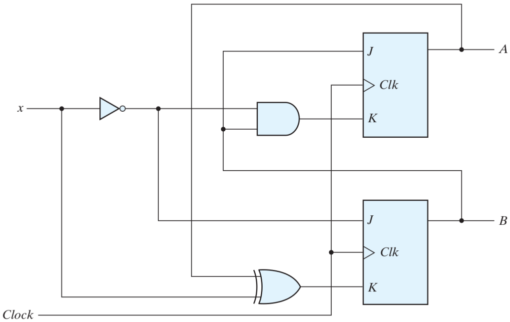
Present state	Inputs		Next state
A	x	y	A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



6- Timing Diagram: $A(0) = 0,\ x = (1,0,1),\ y = (0,1,1)$.



Example 31 (Analysis with j-k flip-flop) .



1- Input Equation (appropriate notation):

$J_A = B \quad K_A = Bx'$,

$J_B = x' \quad K_B = A \oplus x$.

2- Flip-Flop Characteristic Equation:

$A(t+1) = J_A A' + K'_A A$

$B(t+1) = J_B B' + K'_B B$

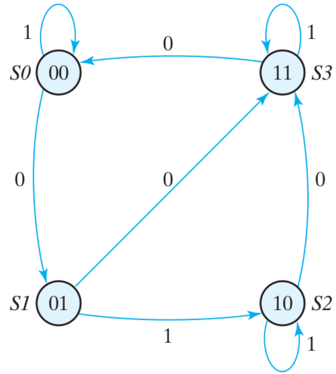
3- Output and State Equations:

$A(t+1) = BA' + (Bx')'A = A'B + AB' + Ax$

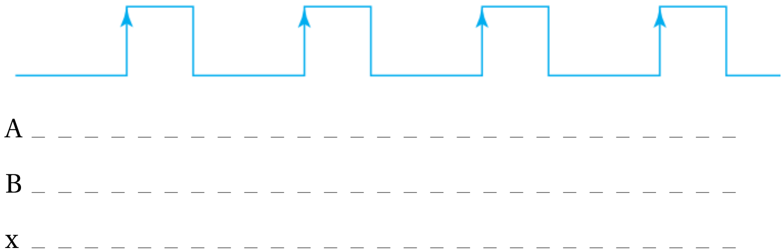
$B(t+1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$.

4,5- State Table and State Diagram (FSM)

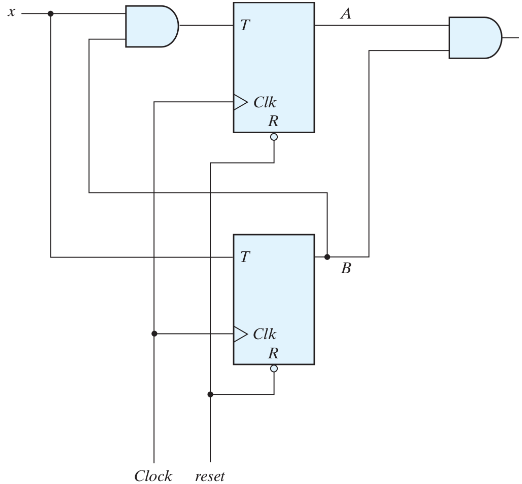
Present State		Input	Next State	
A	**B**	**x**	**A**	**B**
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1



6- Timing Diagram: $A(0) = 0,\ B(0) = 0,\ x = (1, 0, 1)$.



Example 32 (Analysis with T flip-flop) .



1- Input Equation (appropriate notation):

$T_A = Bx, \quad T_B = x.$

2- Flip-Flop Characteristic Equation:

$A(t+1) = T_A \oplus A, \quad B(t+1) = T_B \oplus B$

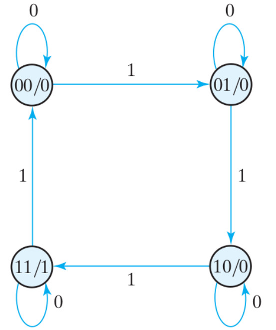
3- Output and State Equations:

$A(t+1) = Bx \oplus A, \quad B(t+1) = x \oplus B,$

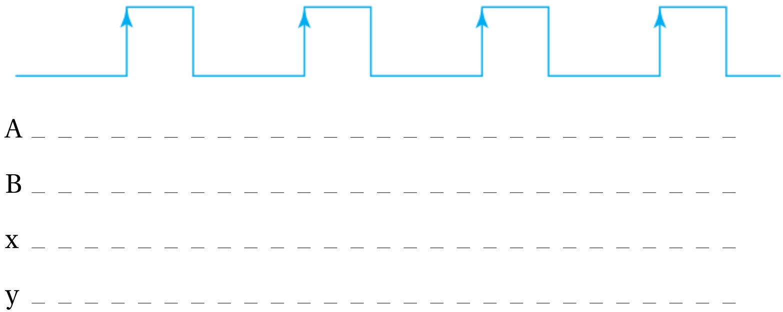
$y = AB.$

4,5- State Table and State Diagram (FSM)

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1



6- Timing Diagram: $A(0) = 0, \ B(0) = 0, \ x = (1, 0, 1).$

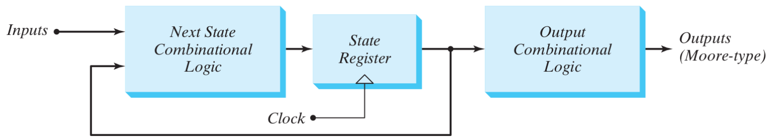


5.5.1 Mealy and Moore Models of FSM

Moore:

- Output is fully synchronized with flip-flops output and, of course, clock even if the input is not synced.

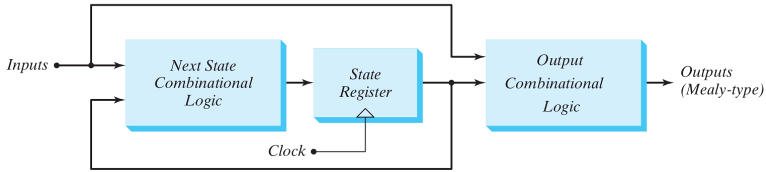
Moore Machine

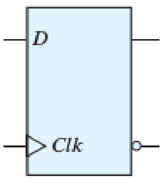


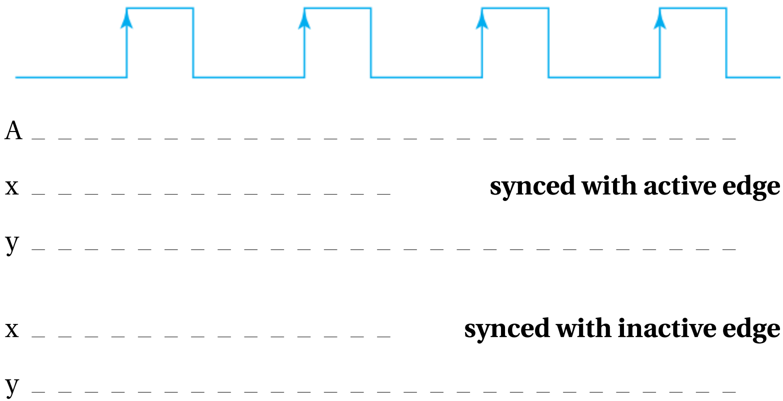
Mealy:

- Output may change during the cycle if input changes!
- Input should be synchronized with clock (either with active or inactive edge)
- The moment right before the active edge is guaranteed to be correct.
- Very simple example: $D = x,\ y = Ax,\ A(0) = 0,\ x = (1,0)$.

Mealy Machine

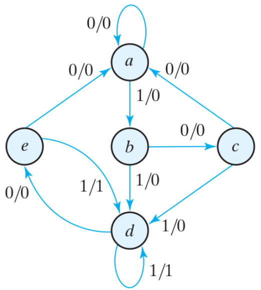


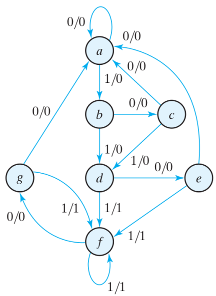




Homework

5.7 State Reduction and Assignments (needed for design)





Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*e*	*f*	0	1

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

Flip-Flop Excitation Tables (inverse function)

characteristic table & equation

excitation table

T	$Q(t+1)$	
0	Q	(no change)
1	Q'	(complement)

$Q(t+1) = T \oplus Q$

T	Q	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

J	K	$Q(t+1)$	
0	0	Q	(no change)
0	1	0	(reset)
1	0	1	(set)
1	1	Q'	(complement)

$Q(t+1) = JQ' + K'Q.$

J	K	Q	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Q	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D	$Q(t+1)$	
0	0	(reset)
1	1	(set)

$Q(t+1) = D$

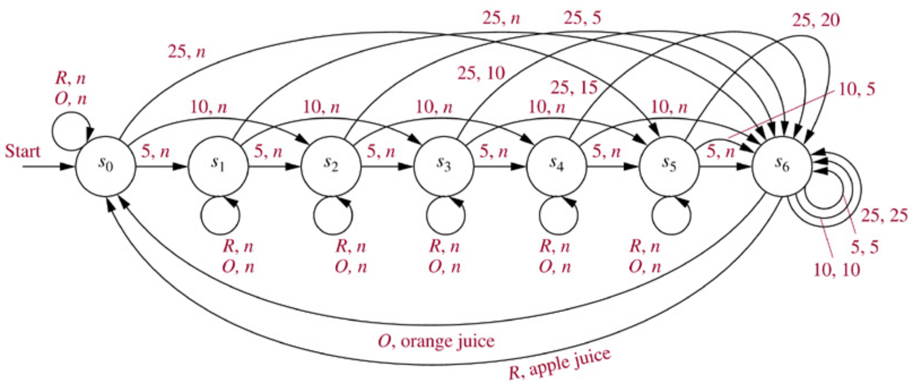
5.8 FSM: a generic preview

5.8.1 Rigorous Mathematical Definition (Rosen, 2007, Ch. 12)

Definition 33 (Finite-State Machines with Outputs and Finite State Automata (without outputs)) *:*
A Finite-State Machine (FSM) $M = (S,\ I,\ O,\ f,\ g,\ s_o)$ *consists of:*

- S*, a finite set of states,*
- I*, a finite input alphabet,*
- O*, a finite output alphabet,*
- $f : S \times I \to S$*, a transition function such that assigns to each state and input pair a new state,*
- g*, an output function that assigns to each state and input pair and output, and*
- s_o*, an initial state.*

Example 34 (Vending Machine) *:*



5.8.2 FSM and Languages:

- Language Recognition
- Compiler Design
- Natural Language Processing
- Formal Languages
- ⋮

Theory of Computation (Sipser, 2006)

1.3 REGULAR EXPRESSIONS 69

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(a \cup b)^*aba$ to an NFA. A few of the minor steps are not shown.

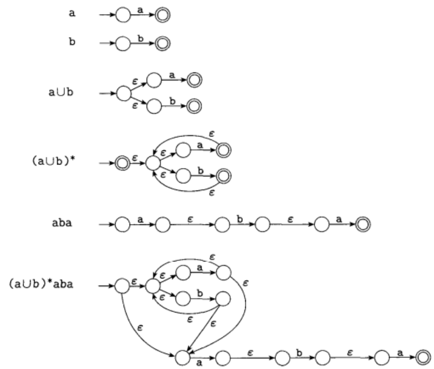


FIGURE 1.59
Building an NFA from the regular expression $(a \cup b)^*aba$

Now let's turn to the other direction of the proof of Theorem 1.54.

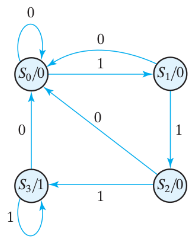
LEMMA 1.60

If a language is regular, then it is described by a regular expression.

PROOF IDEA We need to show that, if a language A is regular, a regular expression describes it. Because A is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.

5.9 Design Procedure: (all is about FSM design; others are routine stuff)

Example 35 (Detect 3 ones in row (overlap)) .



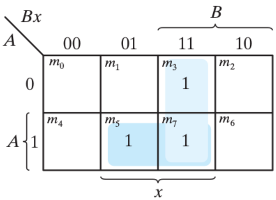
State table, reduction, and assignment

P.S.	N.S. $x=0$	N.S. $x=1$	Y
S_0	S_0	S_1	0
S_1	S_0	S_2	0
S_2	S_0	S_3	0
S_3	S_0	S_3	1

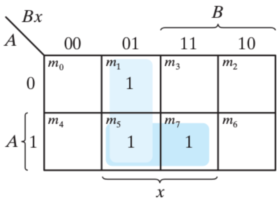
$A\,B$	x	$A\,B$	y	$D_A D_B$	$J_A\,K_A$	$J_B\,K_B$
0 0	0	0 0	0		0 X	0 X
0 0	1	0 1	0		0 X	1 X
0 1	0	0 0	0		0 X	X 1
0 1	1	1 0	0		1 X	X 1
1 0	0	0 0	0		X 1	0 X
1 0	1	1 1	0		X 0	1 X
1 1	0	0 0	1		X 1	X 1
1 1	1	1 1	1		X 0	X 0

excitation table

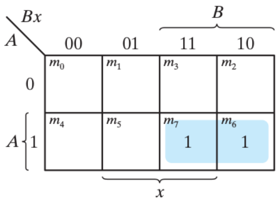
$Q,\ Q(t+1)$	$J\,K$
0 0	0 X
0 1	1 X
1 0	X 1
1 1	X 0



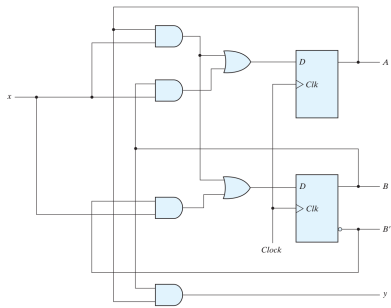
$D_A = Ax + Bx$



$D_B = Ax + B'x$

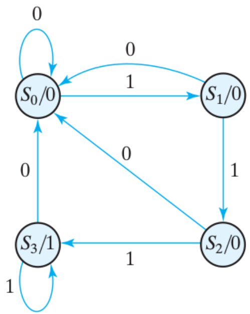


$y = AB$

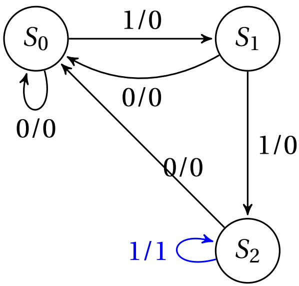


Example 36 (Detect 3 ones in row (with/without overlap) both (Moore and Mealy)) .

overlap Moore



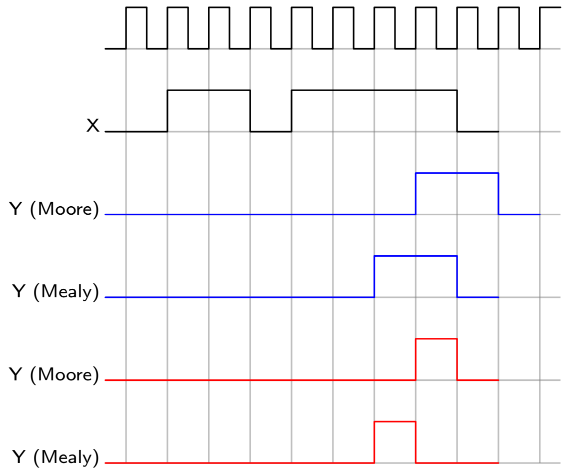
overlap Mealy



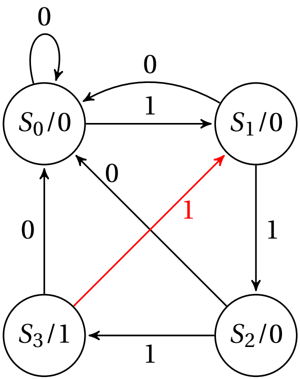
compare to page 112

P.S.	N.S./Y	
	$x = 0$	$x = 1$
S_0	$S_0/0$	$S_1/0$
S_1	$S_0/0$	$S_2/0$
S_2	$S_0/0$	$S_2/1$

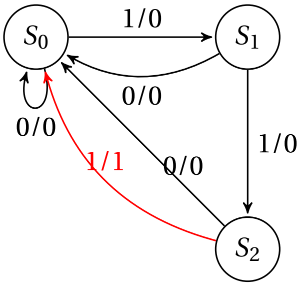
input sequence: 011011110



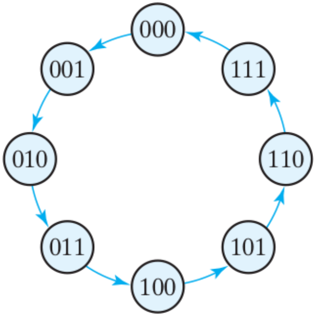
no overlap Moore



no overlap Mealy

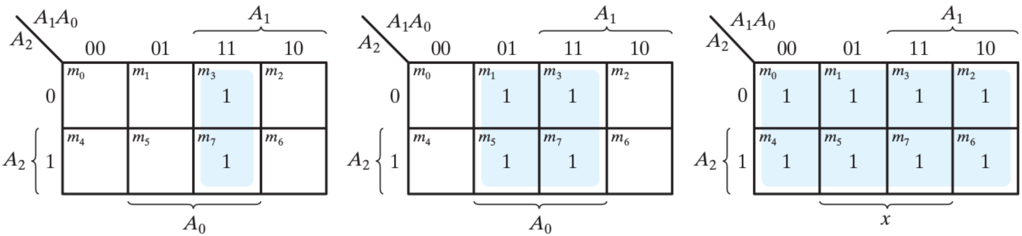


Example 37 (Using a T FF, design a circuit that counts from 0 to 7. comp. to D and JK P. 109) .

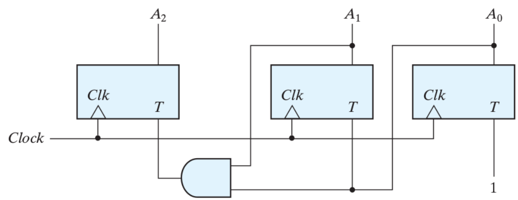


Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

Q	$Q(t+1)$	T
0	0	0
0	1	1
1	0	1
1	1	0



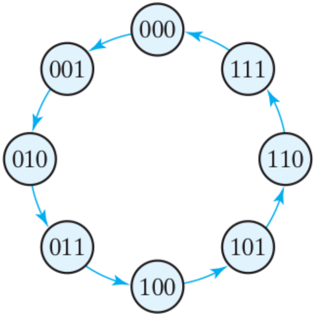
$T_{A2} = A_1A_0$ $\quad$ $T_{A1} = A_0$ $\quad$ $T_{A0} = 1$



$$T_0 = 1$$

$$T_i = \prod_{j=0}^{J=i-1} A_j, \qquad i > 0$$

Example 38 (Using a T FF, design a circuit that counts from 0 to 4 and check for unused states!) .



Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

Q	$Q(t+1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

0	0	1	0
1	X	X	X

$T_{A_2} = A_2 + A_1 A_0$

0	1	1	0
0	X	X	X

$T_{A_1} = A_0$

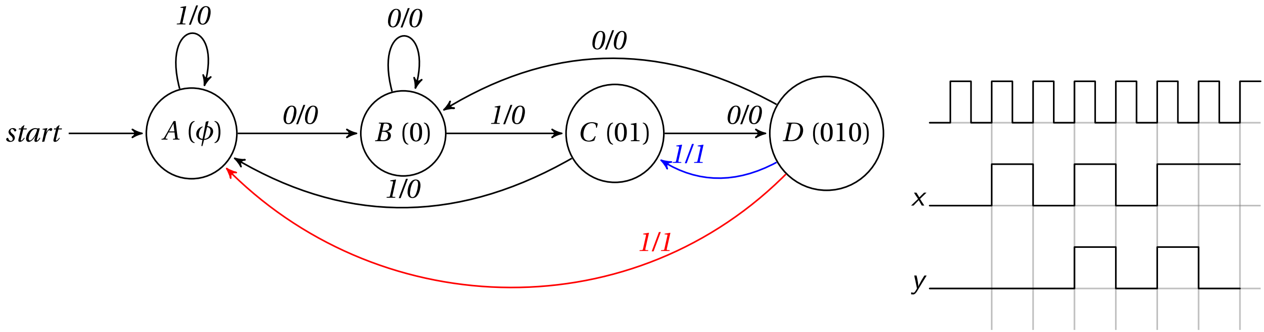
1	1	1	1
0	X	X	X

$T_{A_0} = A_2'$

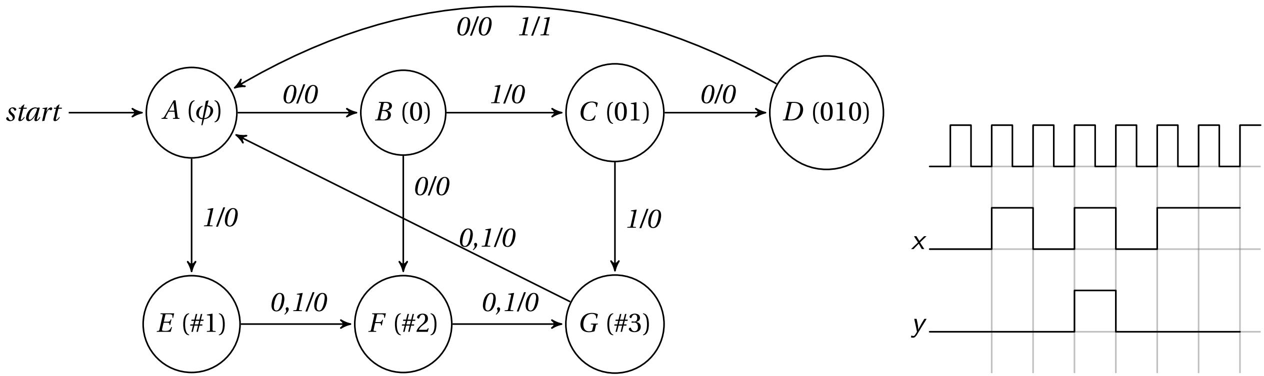
unused states analysis:

P.S.	T	N.S.
$A_2 A_1 A_0$	$T_{A_2} T_{A_1} T_{A_0}$	$A_2 A_1 A_0$
101	110	011
110	100	010
111	110	001

Example 39 (Detect 0101 with/without overlap) .



Example 40 (Detect whether each 4 cycles contain 0101) .



Example 41 (Convert from a JK flip-flop to T flip-flop) .

This is equivalent to: using JK flip-flop design a circuit whose state-table is the characteristic table of T flip-flop (both are essentially the same of course).

No need for transition diagram:

T	Q	$Q(t+1)$	J	K
0	0	0	0	X
0	1	1	X	0
1	0	1	1	X
1	1	0	X	1

$Q, Q(t+1)$	$J\,K$
0 0	0 X
0 1	1 X
1 0	X 1
1 1	X 0

0	X
1	X

$J = T$

X	0
X	1

$K = T$

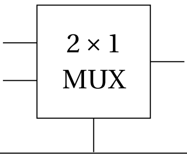
Same as P. 109

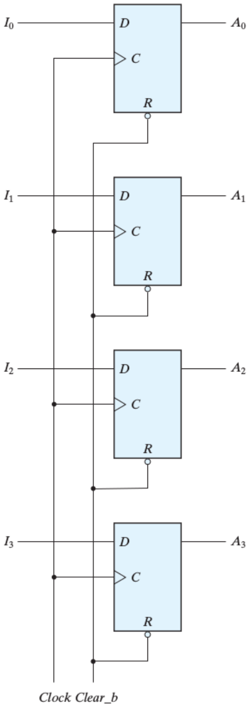
Chapter 6

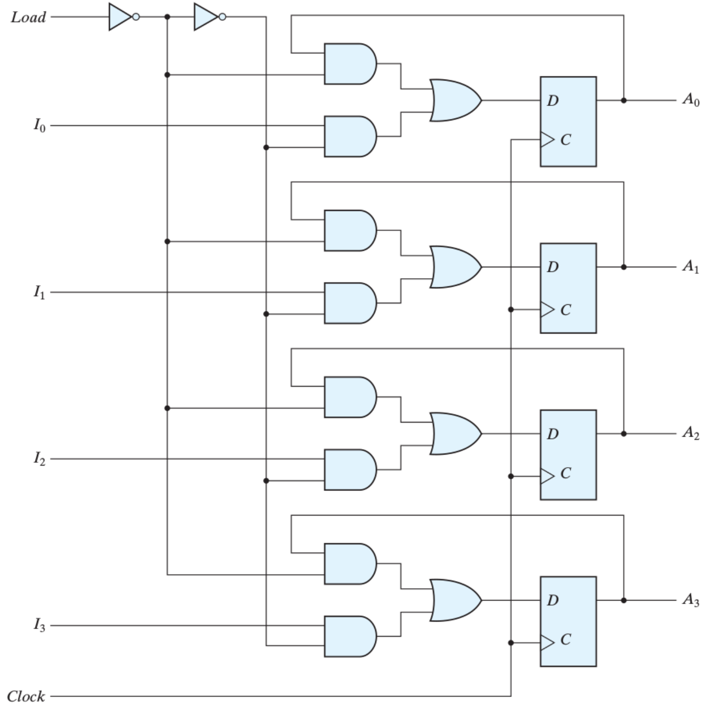
Registers and Counters

Same sections from book but in different order

6.1 Registers (n-bits, with parallel load)







Info. transfer.

Load circuit: MUX.

Modular vs. discrete Logic

L	function	$A_i(t+1)$	D	J	K
0	no change	A_i	A_i	0	0
1	Load	I_i	I_i	I_i	I_i'

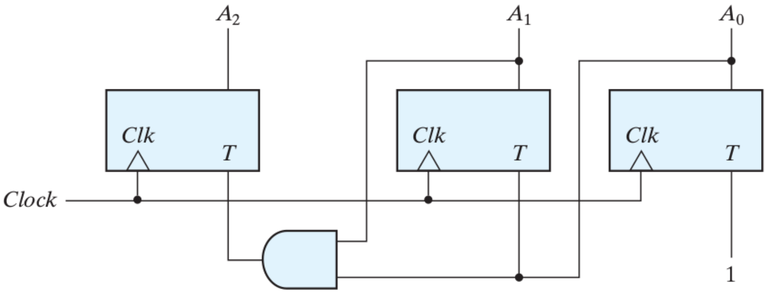
$$D_i = L'A_i + LI_i$$

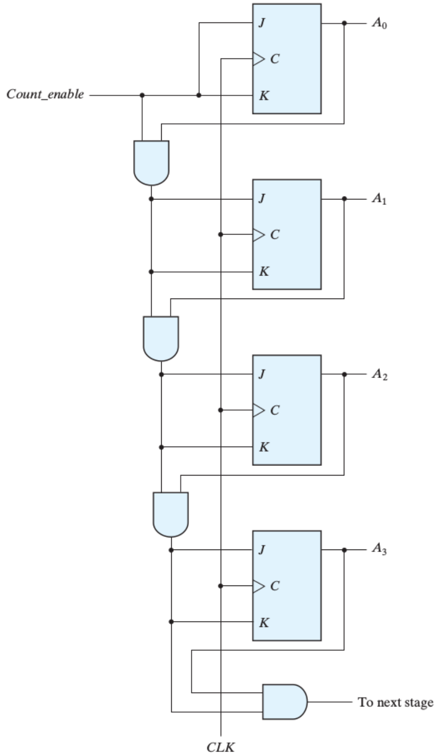
$$J_i = LI_i,$$

$$K_i = LI_i'$$

6.2 Synchronous Counters

6.2.1 Binary Counters





$$T_0 = 1 \qquad \text{(Up/Down)}$$

$$T_i = \prod_{j=0}^{J=i-1} A_j, \qquad i > 0 \qquad \text{(Up)}$$

$$T_i = \prod_{j=0}^{J=i-1} A'_j, \qquad i > 0 \qquad \text{(Down)}$$

- As in Example 37
- Clock could be +/-
- Other complementing FF (P. 109)
- Count-Down counter is immediate by:
- Count-enable (C)

C	Function	T
0	no change	0
1	count up	EQ

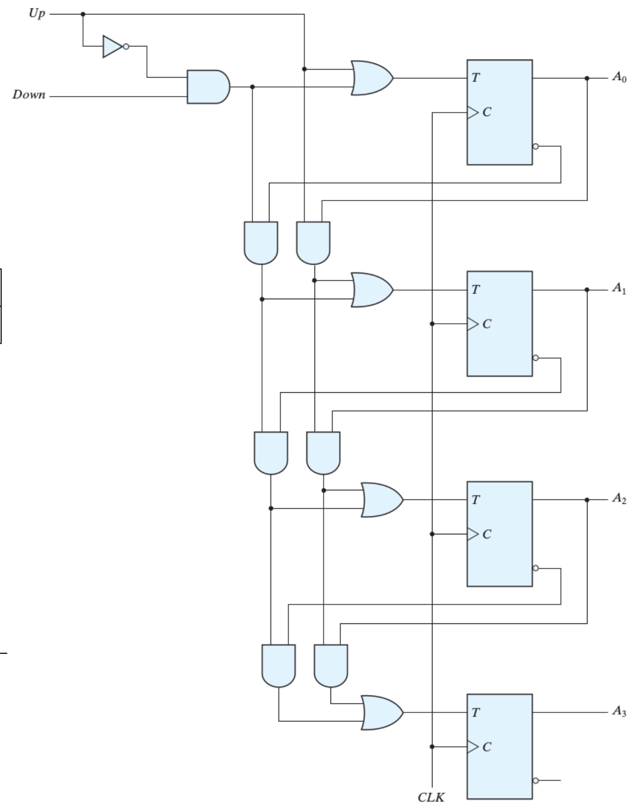
$T = C \cdot EQ$

6.2.2 Up-Down Binary Counter

U	D	Function	T_i
0	0	no change	0
0	1	down	$\prod_0^{i-1} A'_j,\ T_0 = 1$ (EqD)
1	x	up	$\prod_0^{i-1} A_j,\ T_0 = 1$ (EqU)

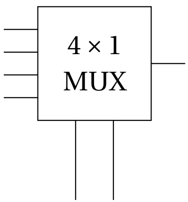
$$T_i = U'D \cdot EqD + U \cdot EqU,$$
$$T_1 = U'D + U = U + D$$

0	EqD
EqU	EqU



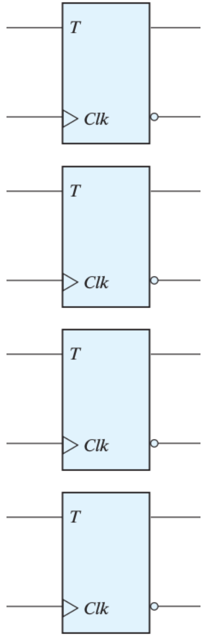
The controllers could be:

C	U	Function	T_i
0	x	no change	0
1	0	down	$\prod_0^{i-1} A'_j,\ T_0 = 1$ (EqD)
1	1	up	$\prod_0^{i-1} A_j,\ T_0 = 1$ (EqU)



6.2.3 BCD Counter

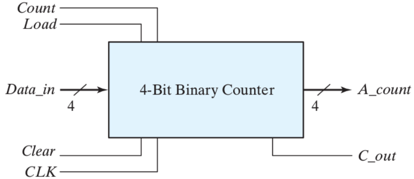
Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1



- **The output** $y = 1$ **during ALL the count 9**
- y can enable the next stage, if any.
- The unused states are taken as X.
- HW: do unused-state analysis.

$$T_{Q_1} = 1$$
$$T_{Q_2} = Q_8' Q_1$$
$$T_{Q_4} = Q_2 Q_1$$
$$T_{Q_8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1.$$

6.2.4 Binary Counter with Parallel Load

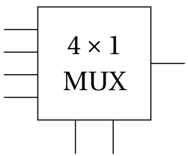


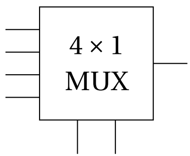
Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

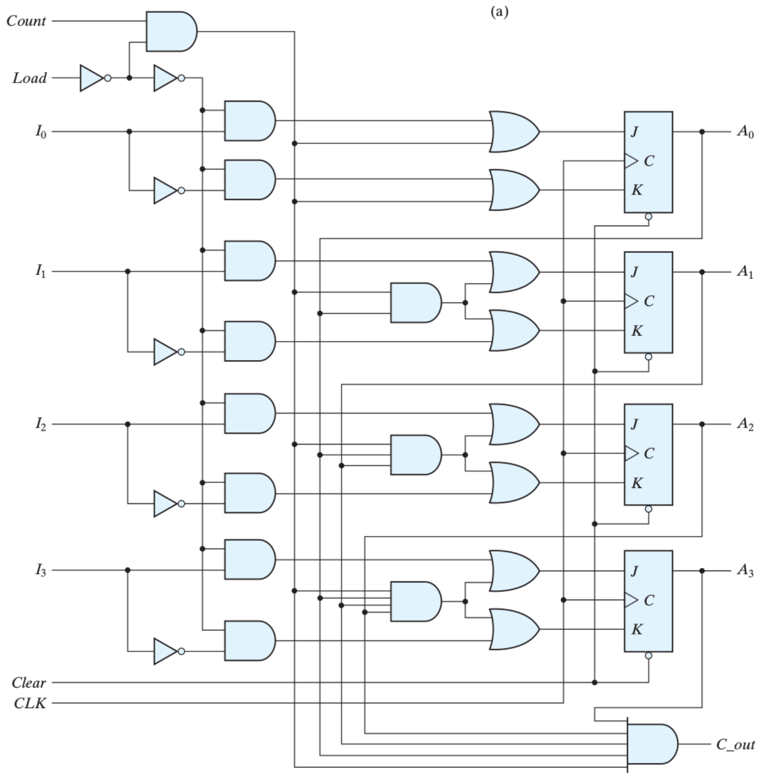
Other func are possible, e.g., sync clear.

L	C	Function	J_i	K_i
0	0	no change	0	0
0	1	Up	EqU	EqU
1	X	Load	I_i	I_i'

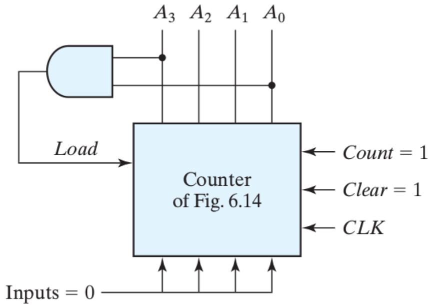
$$J_i = L'C \cdot EqU + L \cdot I_i$$
$$K_i = L'C \cdot EqU + L \cdot I_i'$$

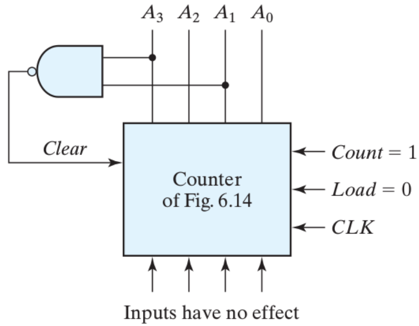






Example 42 (Design a BCD counter using the Binary Counter with Parallel Load) .



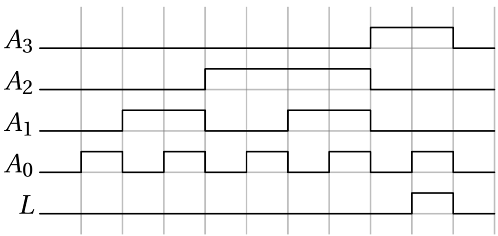


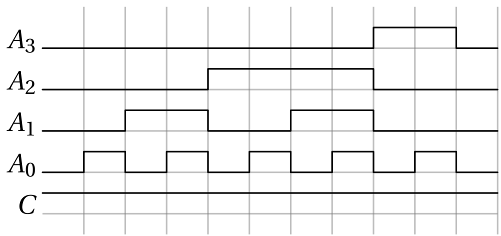
0	0	0	0
0	0	0	0
X	X	X	X
0	1	X	X

$L = A_3 A_0$

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

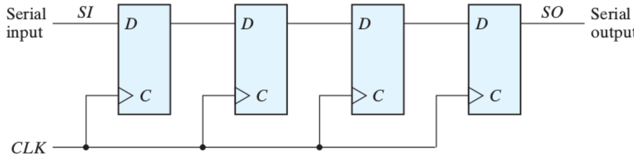
$C = A_3' + A_1' = (A_3 A_1)'$

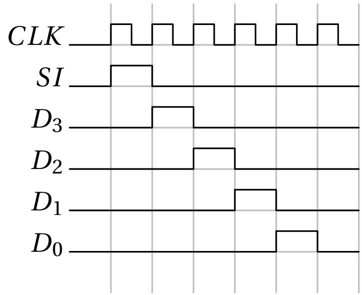


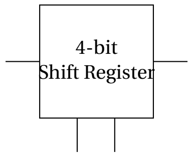


6.3 Shift Registers

- Operations in computers are parallel (faster).
- We need sometimes serial processing/communications (slower):
  - phone lines (serial communications)
  - computer processing (less hardware)
- The output will delay by the number of registers.
- Also, we can convert serial input to parallel.
- Stopping transfer is done by controlling clock (as we will see).
- Controlling should NOT be by controlling clock:
  - Not changing clock path
  - delays clocks in fast circuits (GHz and above)
  - Circuit/design-dependent.
- Alternatively, using 2x1 MUX with L for each D FF, (Sec. 6.1):
  $D_i = L'A_i + LA_{i-1}$, $D_0 = L'A_o + LI$
- Block diagram: serial input (I), serial output (O), clock (C), and transfer control (L).

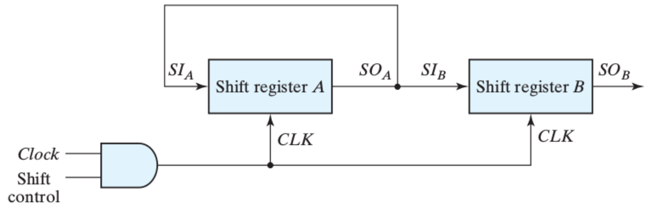


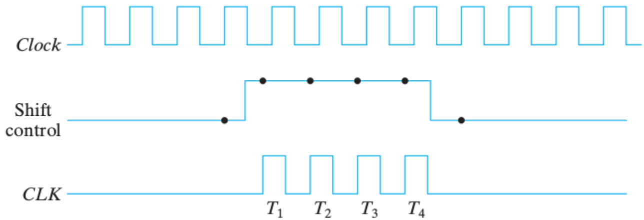




6.3.1 Serial Transfer (compare to parallel registers 6.1)

- Notice: the control signal width = the register size (4 here).
- The control signal should be synchronized with clock
- Designing the control circuit of registers is a major part of computer organization (the second major part is designing the circuits themselves, e.g., ALU)
- **Example:** Initial value of A and B:





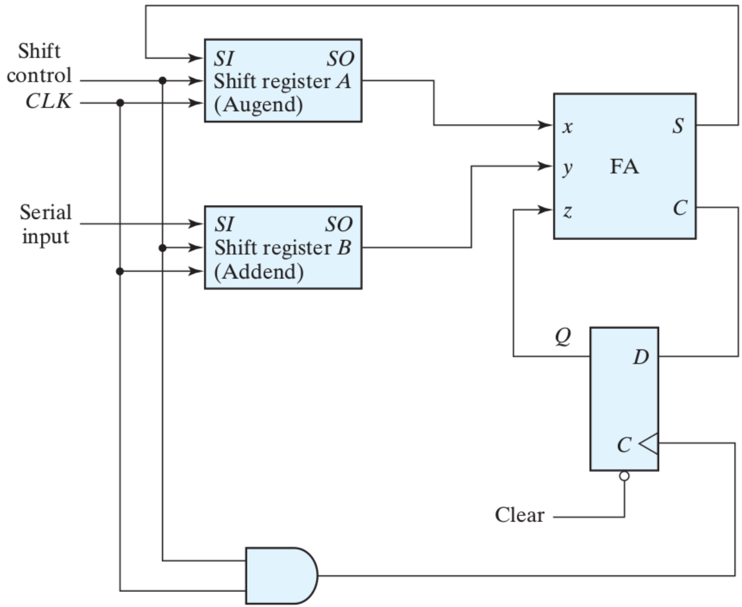
Timing Pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	1	1	0	1	1	0	0	1
After T_2	1	1	1	0	1	1	0	0
After T_3	0	1	1	1	0	1	1	0
After T_4	1	0	1	1	1	0	1	1

6.3.2 Serial Addition

Design 1 (modular):

- Recall the parallel adder of Sec. 4.5.4 to see the trade-off between hardware complexity and speed.
- The serial adder adds one-bit at a time.
- $A = A + SI$ (cumulatively).
- Initially: $A = 0,\ B = SI$.
- After n cycles: $A = A + 0$ and $B =$ new SI.
- Controlling the FF should be by "Load" as explained.
- The next carry-in (z) is the current carry-out (C) => a delay caused by a D FF.
- **Now:** Had not we realized this trick, could we design from scratch? Yes we can but longer and harder. Lets see (with keeping in mind from the FA (4.5.2) that:)

$$C = Qx + Qy + xy$$
$$S = x \oplus y \oplus Q.$$



Design 2 (longer):

- The summation $x+y$ depends on the carry (0 or 1) so it should be represented by a state Q of a single FF; draw the truth table:

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	**x**	**y**	**Q**	**S**	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

- Using D FF:

$$D = Q(t+1) = Qx + Qy + xy$$

$$S = x \oplus y \oplus Q.$$

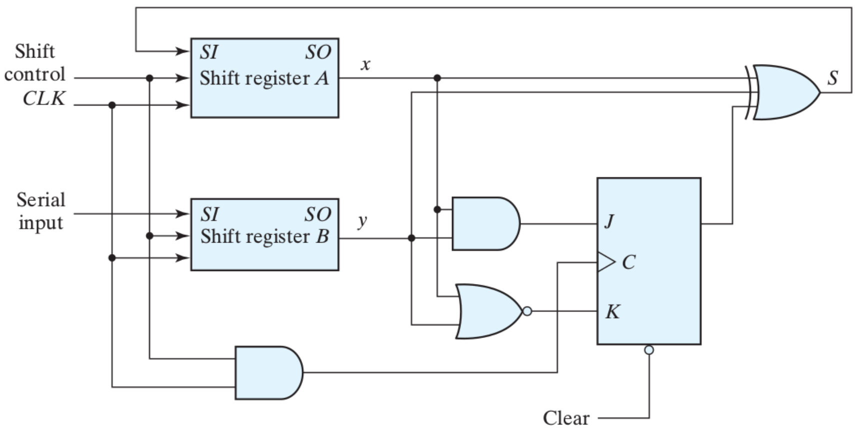
exactly the same as the outputs C and S of the FA in design 1. (A redesign for the FA of Sec. 4.5.2!!)

- Using JK FF:

$$J = xy,$$

$$K = (x+y)',$$

$$S = x \oplus y \oplus Q.$$

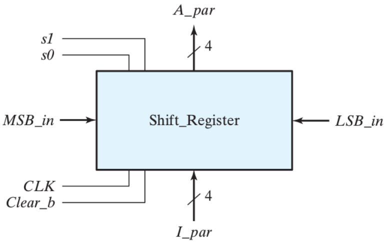


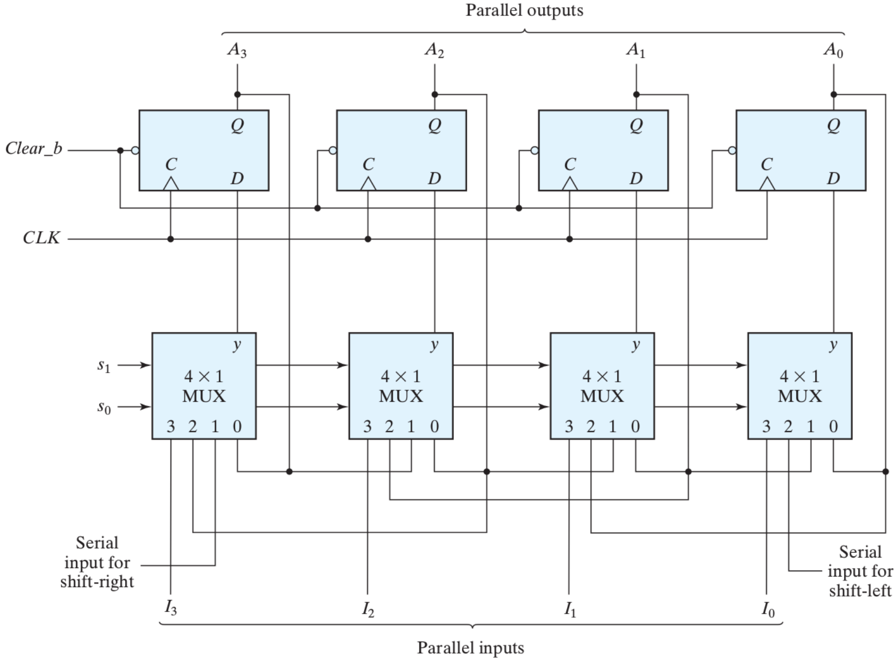
6.3.3 Universal Shift Register

- For shift right, the MSB (w.r.t. $A_3 A_2 A_1 A_0$) or the LSB (w.r.t. SI) should be the input to the MUX.

- Vise versa for shift left.

Mode Control		
s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



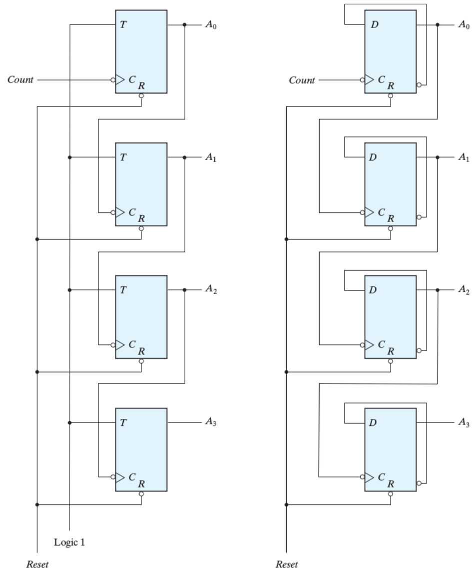


6.4 Ripple Counters

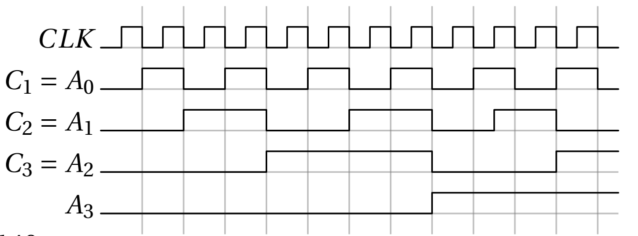
6.4.1 Binary Ripple Counters (all counts exist) recall synchronous 6.2.1

Hint: all toggles at -ve edge (even T_o) because in counting, A_i toggles when A_{i-1} goes from 1 to 0

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0



Using T FF: $Clk_i = A'_{i-1}$, $Clk_0 = Clk'$, $T_i = 1$.
Using D FF: $Clk_i = A'_{i-1}$, $Clk_o = Clk'$, $D_i = A'_i$



Advantages: clock divider, counter, simpler (less hardware, and hence power consumption).
Disadvantages: clock ripples (propagates) and hence delays; so number of stages (n) is restricted. Design is very tricky for non binary counters (see BCD next:)

Delay comparison between synchronous and ripple counters:
Suppose that the FF delay is Δ_1, the `AND` delay is Δ_2, the clock period is T, and # of FFs is n_r, n_s, respectively.

Ripple:
A_0 appears after Δ_1
A_1 appears after $\Delta_1 + \Delta_1 = 2\Delta_1$, and so on...
Then, A_{n_r-1} appears after $n_r\Delta_1$.
The last bit should appear before T (the next edge) => $n_r\Delta_1 < T$.

Synchronous: (Sec. 6.2.1)
All A_i appears after Δ_1 (since clock is common).
However, the input to n^{th} FF suffers from $n_s - 1$ delays of `AND` gates. => $(n_s - 1)\Delta_2 < T$
Almost, $\Delta_1 > 2\Delta_2$ (because the FF master-slave latch has at least 2-level gates (Sec. 5.3.2));
=> $n_s > 2n_r$ => # of counts will be more than squared.

6.4.2 BCD Ripple Counters

Recal the counting mechanism of BCD synchronous counters 6.2.3:

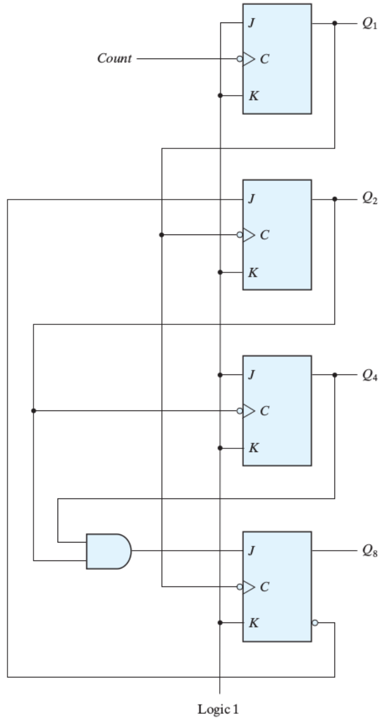
Present State			
Q_8	Q_4	Q_2	Q_1
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1

x	1	1	x
x	1	1	x
x	x	x	x
x	0	x	x

$J_2 = Q_8'$

x	0	0	x
x	0	1	x
x	x	x	x
x	x	x	x

$J_8 = Q_4 Q_2$



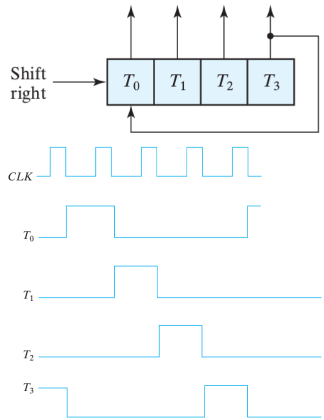
All FFs should either flip or resets => $K_i = 1$ and J_i either 0 or 1
If Q_i feeds the clock of a FF => at +ve edges no transition => x.

FF1: flips when $Ck : 1 \to 0$ => $C_1 = Ck'$ AND no exception => $J = 1$
FF2: flips when $Q_1 : 1 \to 0$ => $C_2 = Q_1'$, AND at 9 $Q_2 : 0 \to 0$ => $J_2 = 0$
FF4: flips when $Q_2 : 1 \to 0$ => $C_4 = Q_2'$, AND no exception => $J_4 = 1$
FF8: flips when $Q_1 : 1 \to 0$ => $C_8 = Q_1'$, AND:
at 7 ($Q_8 : 0 \to 1$ => $J_8 = 1$) OR at 9 ($Q_8 : 1 \to 0$ => $J_8 =$x)

6.5 Other Counters

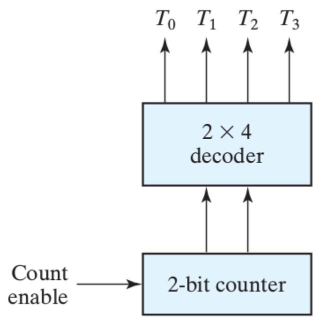
6.5.1 Ring Counters

In control units, we need to keep track of the order of 2^n clock pulse (in which cycle we are now in?) Only one wire of the 2^n will be one at a clock cycle.

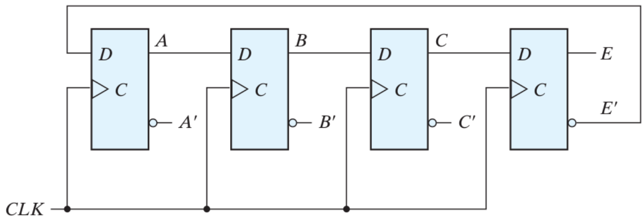


Design 1 (ring counter):
2^n-bit shift register with initially "$1000\cdots$"
=> 2^n FFs

Design 2 :
one n-bit binary counter + one $n \times 2^n$-decoder
=> (n FFs + 2^n n-input AND gates).



6.5.2 Johnson Counter



Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

Design 3 : (modified ring counter):
2^{n-1}-bit shift register with initially "$00\cdots$" + 2^n 2-input AND gates
=> 2^{n-1} FFs + 2^n 2-input AND gates.

Advantages : It is half the number of FFs than the ring counters with additional AND gates. The AND gates are 2-input (to be compared with the n-input of **Design 2**)

Disadvantages : will not get out of any visited unused state!

Homework

Chapter 7

Memory and Programmable Logic

7.1 Introduction

Motivation : memory for storing information to be processed by processor

- All circuits designed so far will be part of the processor ALU.
- The second part of the processor is the CU.
- Processor needs to access information to be processed.

Memory : collection of cells for storing information (bits) to be fetched from/to processor

- Random-Access Memory (RAM):
  - Storing data (writing)
  - Retrieving data (reading)
  - volatile
- Read-Only Memory (ROM) is a PLD device.

Programmable Logic Device (PLD) : programming is a hardware procedure specifying bits inserted into the device for future use. PLD may has 100s–1000,000s of gates.

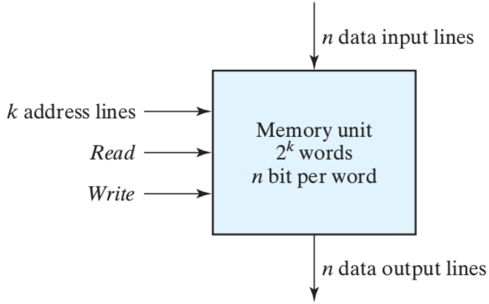
- Read-Only Memory (ROM)
- Programmable Logic Array (PLA)
- Programmable Array Logic (PAL)
- Field-Programmable Gate Array (FPGA)

7.2 Random-Access Memory (RAM): block-design

- "Random-Access": access time is constant
- "Byte": 8 bits
- "Word": multiple of bytes
- "Address": points at word, $(0) - (2^k - 1)$.
- "capacity" or "size": total number of bytes in memory.
- Ex. for 10-bit address and 2-byte word length:

$$\begin{aligned} Size &= 2^{10}\ \textbf{Words} \\ &= 1024\ \textbf{Words} \\ &= 1\ \textbf{K Words} \\ &= 1\ \textbf{K} \times 2\ \textbf{Bytes} \\ &= 2\ \textbf{KB} \end{aligned}$$

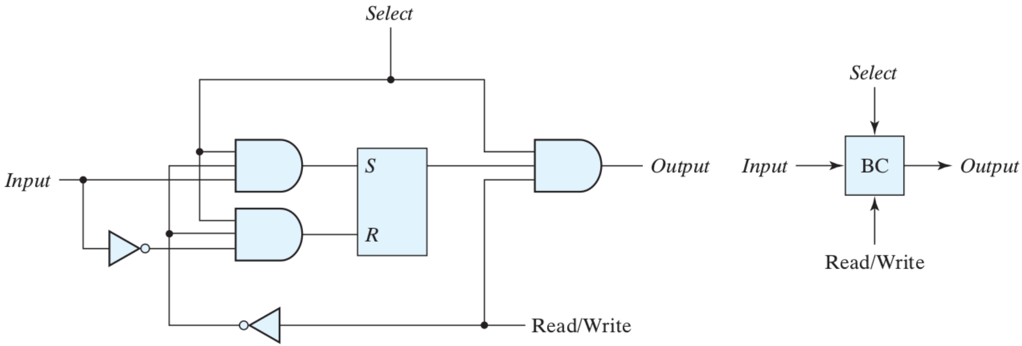
- $1\ \text{G} = 1024\ \text{M} = 2^{10}\ \text{M}$
- $1\ \text{M} = 1024\ \text{K} = 2^{10}\ \text{K}$
- $1\ \text{K} = 1024 = 2^{10}$
- $1\ \text{G} = 2^{30}$



Memory address		
Binary	**Decimal**	**Memory content**
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

7.3 Memory Decoding

7.3.1 Internal Construction of 1-bit RAM



(a) Logic diagram (b) Block diagram

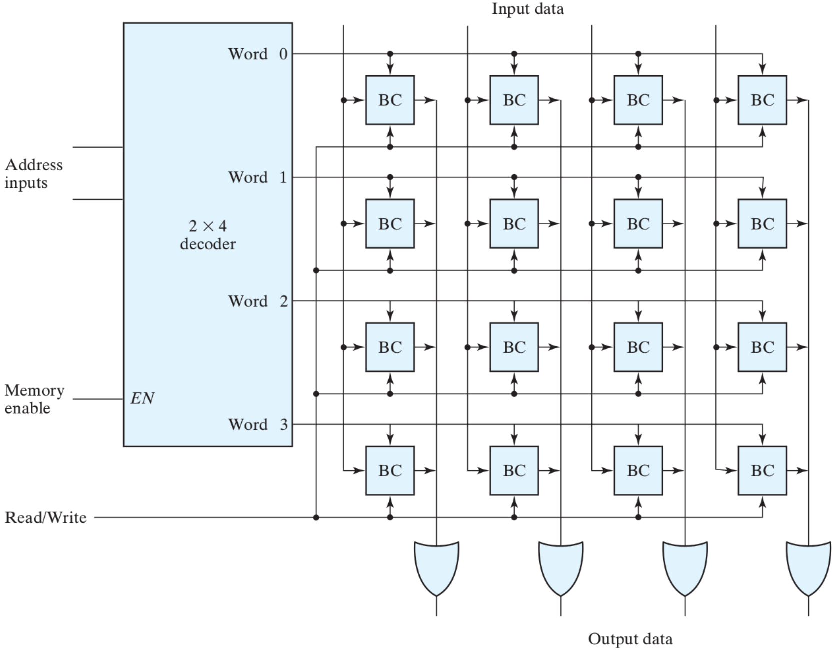
- RAM is fabricated directly not implemented as connected components (this is just a logic diagram)
- Registers vs. RAM (inside processor, sync. with clock, faster, not expandable, etc.)

sel	read	function	output	memory content (Q)	S	R
0	X	no. change no. output	0	no change	0	0
1	0	write	0	changes	I	I'
1	1	read	Q	no change	0	0

$$S = sel \cdot read' \cdot I$$
$$R = sel \cdot read' \cdot I'$$
$$output = sel \cdot read \cdot Q$$

2^k**-words-**n**-bit RAM**:
(4 × 4 in this example)

- The OR gates are represented as array logic.
- Decoder Enable accounts for Enable to the whole memory unit.
- Lets save the hardware of the decoder:



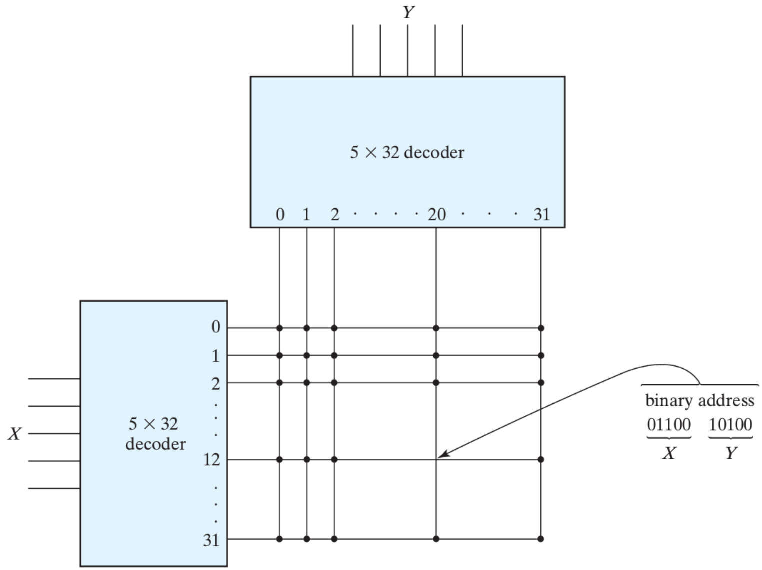
(a) Conventional symbol (b) Array logic symbol

7.3.2 Coincident Decoding

- Redesign each BC has 2 selects from 2 decoders (for row/column); and hence each word has two selects.
- So, memory cells are arranged in a matrix (a dot is a word).
- The address now is: $k/2$ MSBs (X) and $k/2$ LSBs (Y).
- $k \times 2^k$-decoder (prev. design:) 2^k k-input AND gates.
- $2\ (k/2) \times 2^{(k/2)}$-decoder (coinc. dec.): $2 * 2^{(k/2)}\ (k/2) - input$ AND gates.
- $2^k > 2^{1+(k/2)}$ if $k > 2$.
- For this decreased HW, can we decrease the # of pins? Lets see:

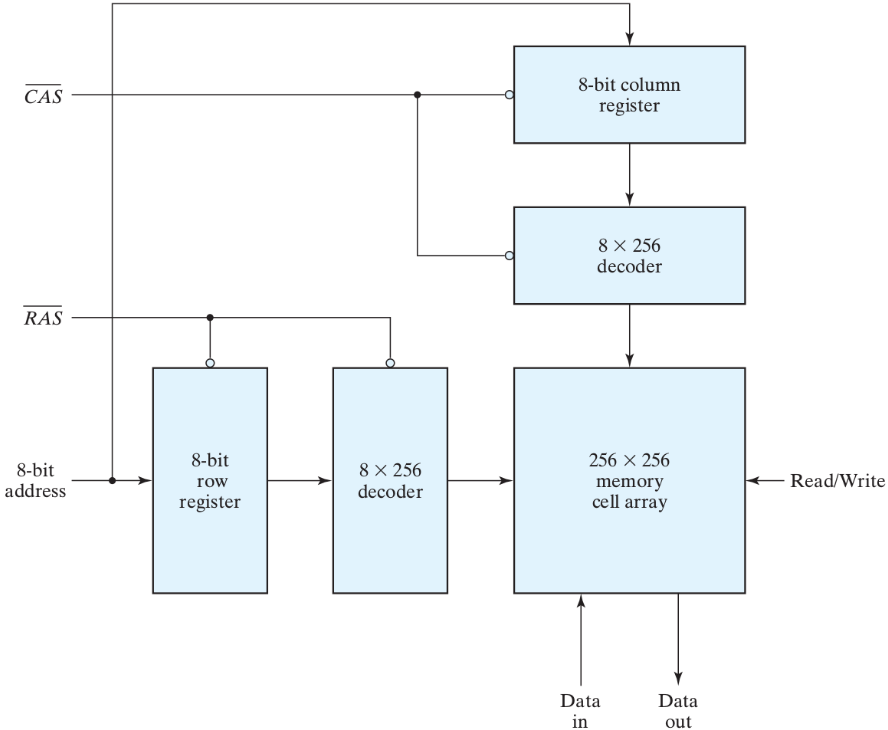
Example: $1024 \times n$ memory

- The word number 404 is addressed by 0110010100 in a direct list ordering.
- but addressed as $X = 01100$ and $Y = 10100$ in coincident decoding.



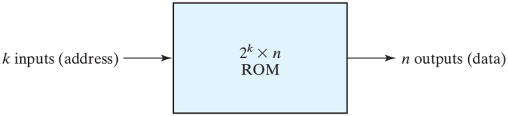
7.3.3 Address Multiplexing

- The Row/Column Address Strobe (RAS/CAS) selects the MSB/LSB part of the address line (only $k/2$). A **very smart** trick to reduce the number of pins in the IC is to use only $k/2$ but in two cycles using 2 registers to save X and Y.
- This is always needed in DRAM, which is designed with only one MOS transistor and a capacitor that stores information (but discharges with time; so it needs refreshing); while SRAM BC almost needs 6 transistors;
- DRAM saves: space, cost, power.



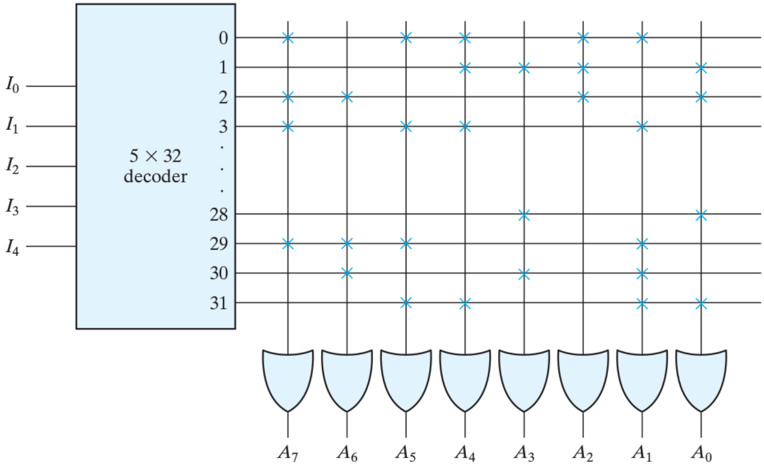
7.4 Error Dectection and Correction

7.5 Read-Only Memory (ROM): non-volatile



- Motivation: flash memory and motherboard BIOS.
- ROM: comes programmed from manufacturer (clients provide their truth table)
- PROM: Programmable ROM, is programmed using special burners.
- EPROM: Erasable PROM (erased by ultra-violet application)
- EEPROM: Electrically Erasable PROM, e.g., motherboard bios and flash memory.
- In this example, A_i is the output of `OR`ing all decoder outputs. The *crosspoints* have fuses all are normally connected to give `One` (intact and shown as stars *). When blown up (by high voltage) the *crosspoint* is disconnected to give `Zero`.

Inputs					**Outputs**							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮						⋮				
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

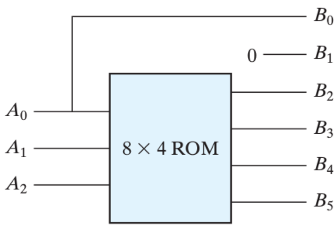


Example 43 (7.1) *: ROM can be seen as:*

Storage device *: if we look at the table row-wise. The k-bit input will selected the appropriate row.*

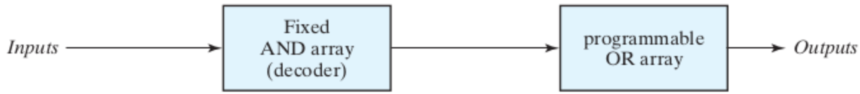
Combinational functions *: if we look at the table column-wise. The k-bit input has n corresponding functions A_1-A_n. For a particular input, each function provides a value.*

Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

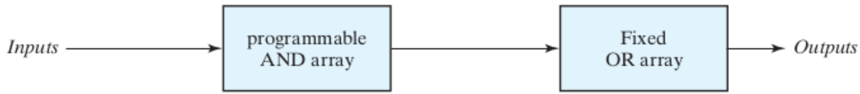


A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

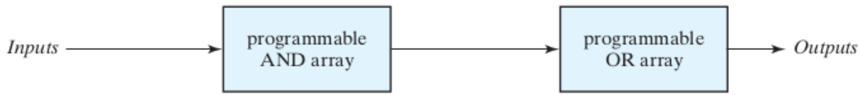
ROM is one of three types of PLDs:



(a) Programmable read-only memory (PROM)



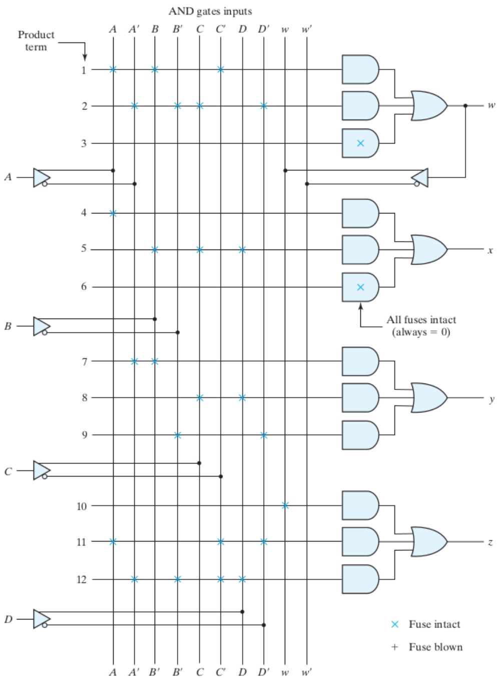
(b) Programmable array logic (PAL)



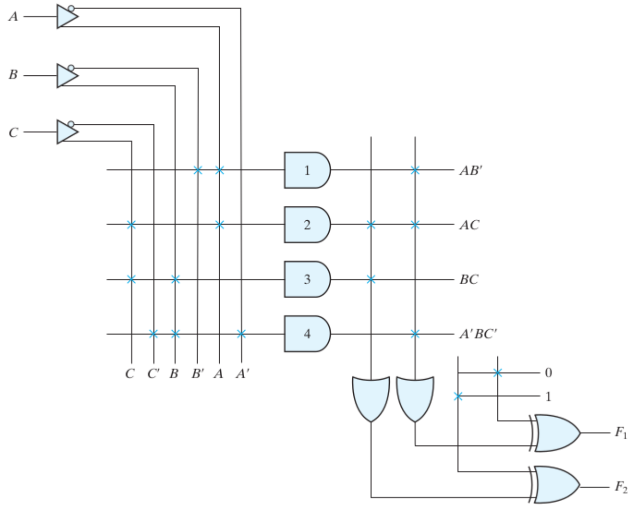
(c) Programmable logic array (PLA)

Same concept as ROM; we will just allude to that:

7.6 Programmable Array Logic (PAL)

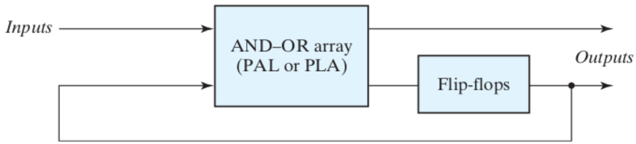


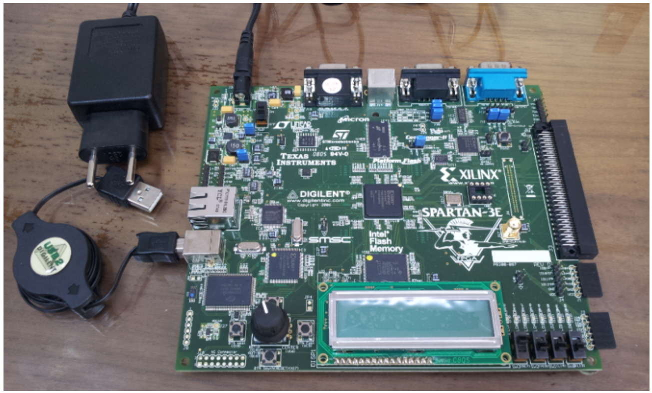
7.7 Programmable Logic Array (PLA)



7.8 Sequential Programmable Devices: (more advanced topics)

- To add the sequential circuitry to the IC for extending PLD.
- This is a topic that is very advanced for a first course in Digital Design.
- The most important and recent technology is Field-Programmable Gate Array (FPGA).
- It is: VLSI circuit that can be programmed at the user's location.
- An FPGA contains: thousands or more programmable logic blocks.
- A logic block contains: multiplexers, gates, FFs.
- Programming requires extensive HDL coding to describe the required hardware.
- The most famous FPGA is by Xilinx nc. (/'zy-lingks)
- Breadboards, Printed Circuit Boards (PCB), FPGA, Fabrication. (Sec. 2.9)





Xilinx, Spartan 3E: Summer training, 2008, for top 10 students in digital design. The power and USB-PC connector are shown.

Chapter 8

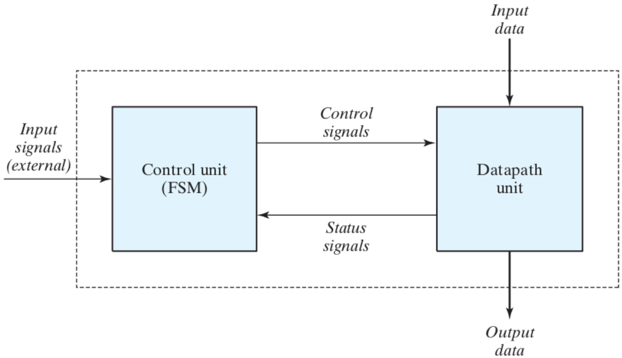
Design at the Register Transfer Level: (a gate to "Computer Organization")

8.1 Introduction

- Remember block design of combinational circuits, e.g., 4.7.2.
- We need the same for sequential circuits; otherwise, we will do state table for hundreds of states.
- When it comes to very large scale integration, it is impossible but block-design.
- Same concept in programming; no one writes high level functions in Assembly.
- We will provide essential tools here in this chapter and leave the rest to the whole course of "Computer Organization".

8.2 Register Transfer Language (RTL)

Simple examples: draw datapath vs. control



1. very simple transfer

$$R2 \longleftarrow R1$$

2. controlling the transfer (see ring counter 6.5.1 for T3)

$$\texttt{If}\ (T3 = 1) \quad \texttt{then} \quad R2 \longleftarrow R1$$

3. more transfer at same condition

$$\texttt{If}\ (T3 = 1) \quad \texttt{then} \quad R2 \longleftarrow R1,\ R1 \longleftarrow R2$$

4. other transfer and control

$$\texttt{If}\ (T2 = 1) \quad \texttt{then} \quad R1 \longleftarrow R1 + R2$$

5. Here, we need additional MUX and feedback from datapath to control.

$$\texttt{If}\ (T2 = 1) \quad \texttt{then} \quad \texttt{If}\ (R1 > 0) \quad \texttt{then} \quad R1 \longleftarrow R1 + R2$$
$$\texttt{If}\ (T3 = 1) \quad \texttt{then} \quad R2 \longleftarrow R1,\ R1 \longleftarrow R2$$

Homework

8.4 Algorithmic State Machines (ASMs)

Rectangle box is a state with name and number. Transitions happens next edge with a "Moore" control, e.g., `incr_A`.

Diamond box is a "Meally" condition.

Round rectangle : is a conditional **Mealy output** that occurs once the condition happens. **However:** the register actions take place next edge of course.

ASM Block is therefore a rectangle box, along with all diamonds, and rounded boxes stemming from it. All takes place either during the cycle (Mealy conditions and outputs) or at the next edge (register transfer).

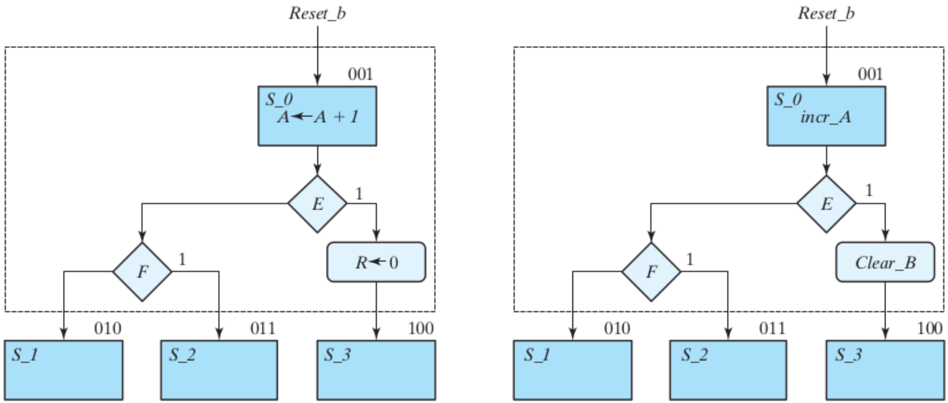
Asynchronous Start `Reset_b`.

ASM vs. Flow Chart ASM is different in the timing of actions. $A \longleftarrow A+1$, $R \longleftarrow 0$, and transition to next state **ALL** happens at the end of S_0 at the edge of next clock; while in flow-chart it is executed right away. E.g., $R \longleftarrow 0$ is in S_0 block; however, it means that the controller is ready during S_0 to cause this transfer at the exit of S_0 (next edge).

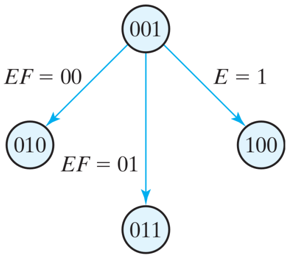
Figures (b), (c) : datapath and controls.

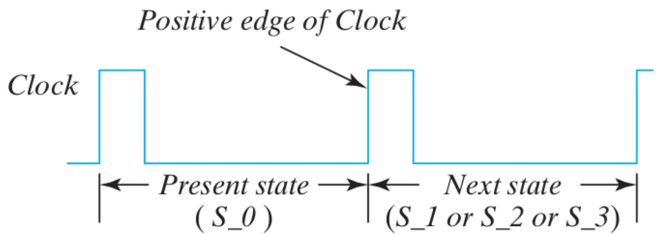
We can convert the ASM chart to a state diagram for easy design like this one =>

ASMD (Algorithmic State Machine and Datapath) combines both (a) and (b) for very clear design and implementation as will be seen next example:



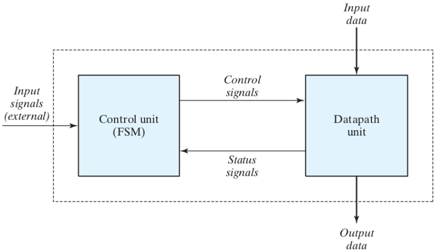
(a) (b)

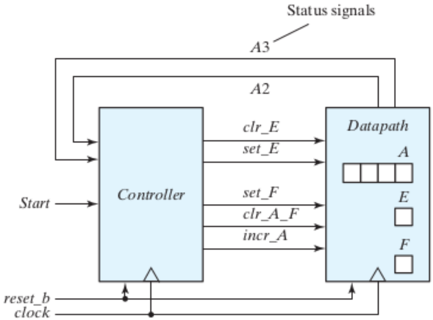




8.5 Design Example

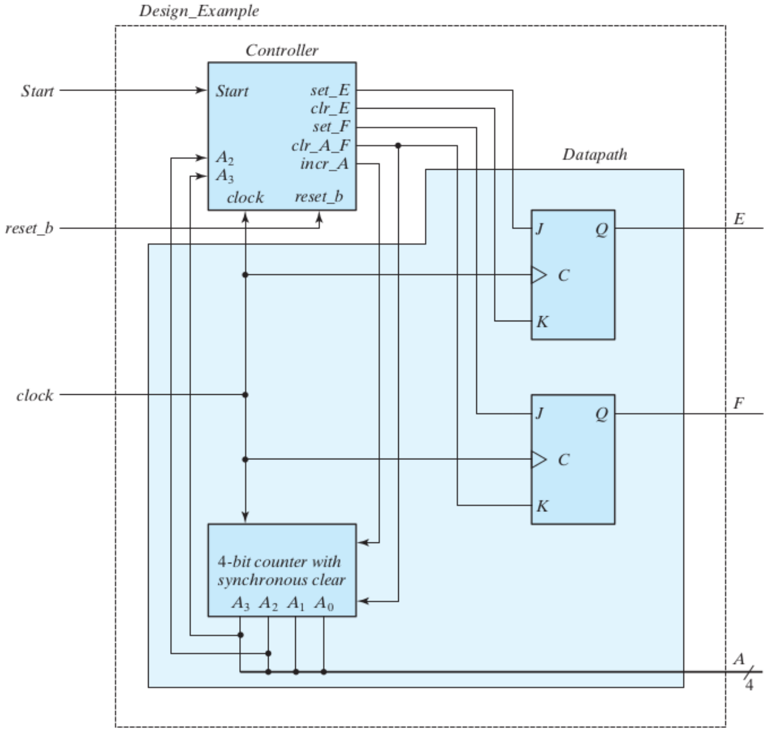
- 2 JK FF (E and F), and one 4-bit binary counter ($A[3:0]$).
- `Start` initiates the system: ($A = 0,\ F = 0$).
- Each clock pulse, A is incremented.
- If $A_2 = 0$, E is cleared and count continues.
- If $A_2 = 1$, E is set to 1; then if $A_3 = 0$, count continues, but if $A_3 = 1$, F is set to 1 on the next clock pulse and system stops counting. Then, if $Start = 0$, system remains in the initial state, but if $Start = 0$ repeat.





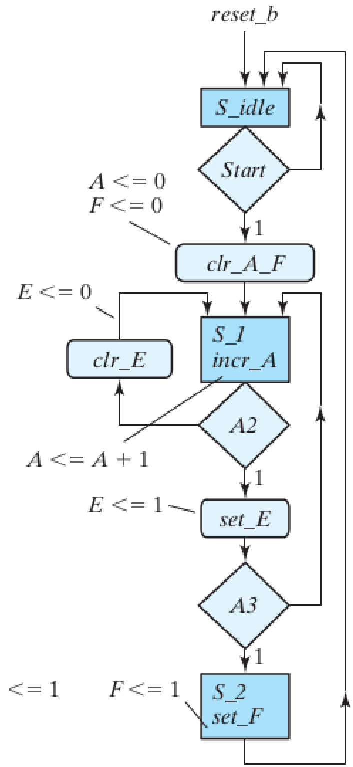
(a)

Datapath:



Cont.

- 2 JK FF (E and F), and one 4-bit binary counter ($A[3:0]$).
- `Start` initiates the system: ($A = 0,\ F = 0$).
- Each clock pulse, A is incremented.
- If $A_2 = 0$, E is cleared and count continues.
- If $A_2 = 1$, E is set to 1; then if $A_3 = 0$, count continues, but if $A_3 = 1$, F is set to 1 on the next clock pulse and system stops counting. Then, if $Start = 0$, system remains in the initial state, but if $Start = 0$ repeat.



Tracing the ASMD assuming that we come from `S_2`:

Sequence of Operations for Design Example

Counter				Flip-Flops			
A_3	A_2	A_1	A_0	E	F	Conditions	State
0	0	0	0	1	0	$A_2 = 0,\ A_3 = 0$	*S_1*
0	0	0	1	0	0		
0	0	1	0	0	0		
0	0	1	1	0	0		
0	1	0	0	0	0	$A_2 = 1,\ A_3 = 0$	
0	1	0	1	1	0		
0	1	1	0	1	0		
0	1	1	1	1	0		
1	0	0	0	1	0	$A_2 = 0,\ A_3 = 1$	
1	0	0	1	0	0		
1	0	1	0	0	0		
1	0	1	1	0	0		
1	1	0	0	0	0	$A_2 = 1,\ A_3 = 1$	
1	1	0	1	1	0		*S_2*
1	1	0	1	1	1		*S_idle*

$$D_{G1} = G_1' G_o\ A_2\ A_3$$

$$D_{G_o} = Start\ G_1' G_o' + G_1' G_0$$

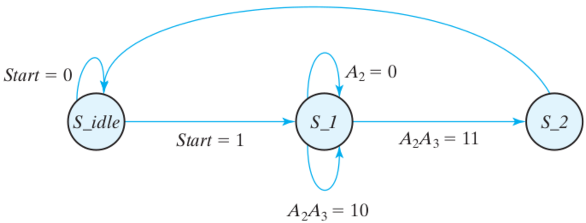
$$set\_E = G_1' G_o\ A_2$$

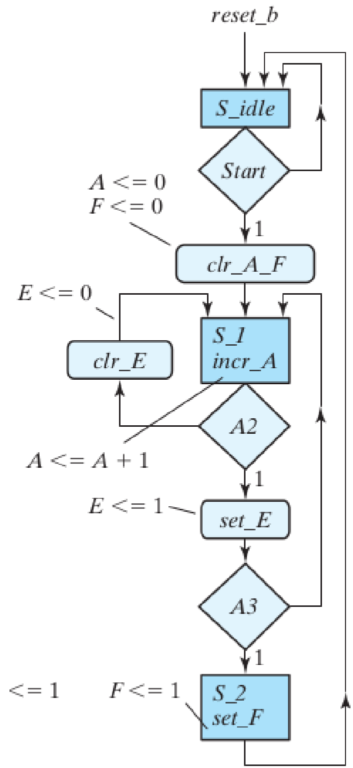
$$clr\_E = G_1' G_o\ A_2'$$

$$set\_F = G_1 G_o$$

$$clr\_A\_F = Start\ G_1' G_o'$$

$$incr\_A = G_1' G_o$$





State Table for the Controller of Fig. 8.10

Present-State Symbol	Present State G_1	Present State G_0	Inputs *Start*	Inputs A_2	Inputs A_3	Next State G_1	Next State G_0	Outputs *set_E*	Outputs *clr_E*	Outputs *set_F*	Outputs *clr_A_F*	Outputs *incr_A*
S_idle	0	0	0	X	X	0	0	0	0	0	0	0
S_idle	0	0	1	X	X	0	1	0	0	0	1	0
S_1	0	1	X	0	X	0	1	0	1	0	0	1
S_1	0	1	X	1	0	0	1	1	0	0	0	1
S_1	0	1	X	1	1	1	1	1	0	0	0	1
S_2	1	1	X	X	X	0	0	0	0	1	0	0

$$D_{G1} = G_1' G_o\ A_2\ A_3$$

$$D_{G_o} = Start\ G_1' G_o' + G_1' G_0$$

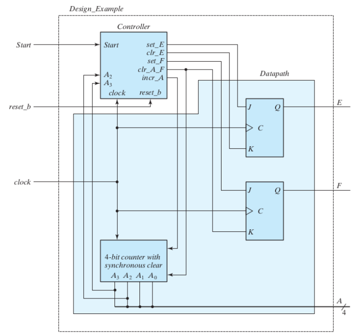
$$set\_E = G_1' G_o\ A_2$$

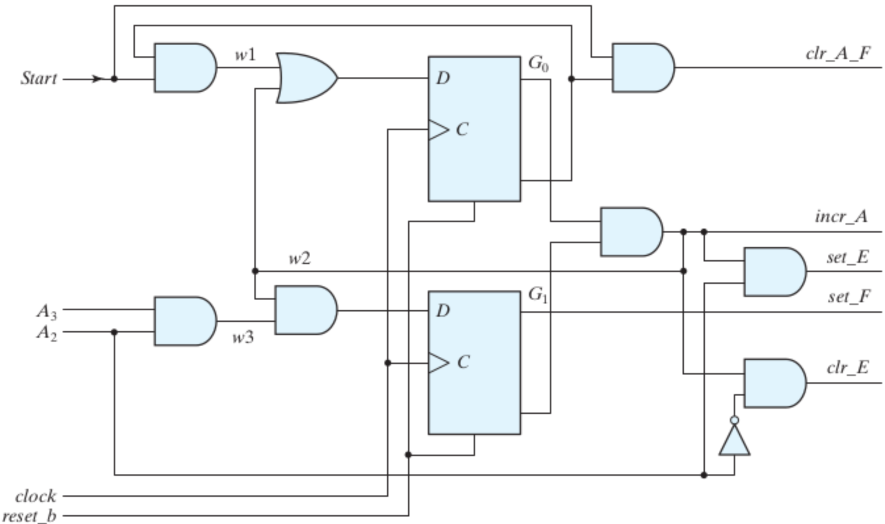
$$clr\_E = G_1' G_o\ A_2'$$

$$set\_F = G_1 G_o$$

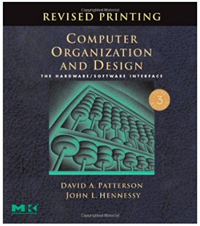
$$clr\_A\_F = Start\ G_1' G_o'$$

$$incr\_A = G_1' G_o$$

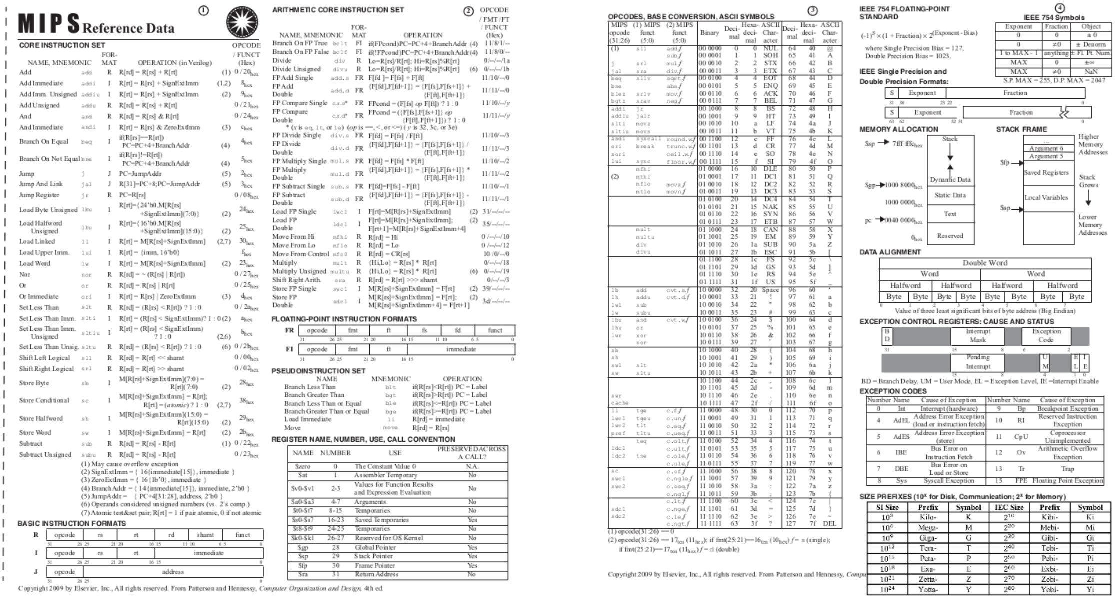




Example 44 (a whole course!) *building the MIPS computer in "Computer Organization" course:*



- *Patterson, D. A., Hennessy, J. L., 2007. Computer Organization And Design : The Hardware/Software Interface, 3rd Edition. MORGAN KAUFMANN, Boston*
- *"The hardware/software interface" wow!. Look at the branching statements and connection to function call, if conditions, etc.*



Appendix A

Numbering System and Binary Numbers

Appendix B

Boolean Algebra

Appendix C

Digital Integrated Circuits (ICs)

Bibliography

Mano, M. M., Ciletti, M. D., 2007. Digital design, 4th Edition. Prentice-Hall, Upper Saddle River, NJ.

Patterson, D. A., Hennessy, J. L., 2007. Computer Organization And Design : The Hardware/Software Interface, 3rd Edition. MORGAN KAUFMANN, Boston.

Rosen, K. H., 2007. Discrete mathematics and its applications, 6th Edition. McGraw-Hill Higher Education, Boston.

Sipser, M., 2006. Introduction to the theory of computation, 2nd Edition. Thomson Course Technology, Boston.