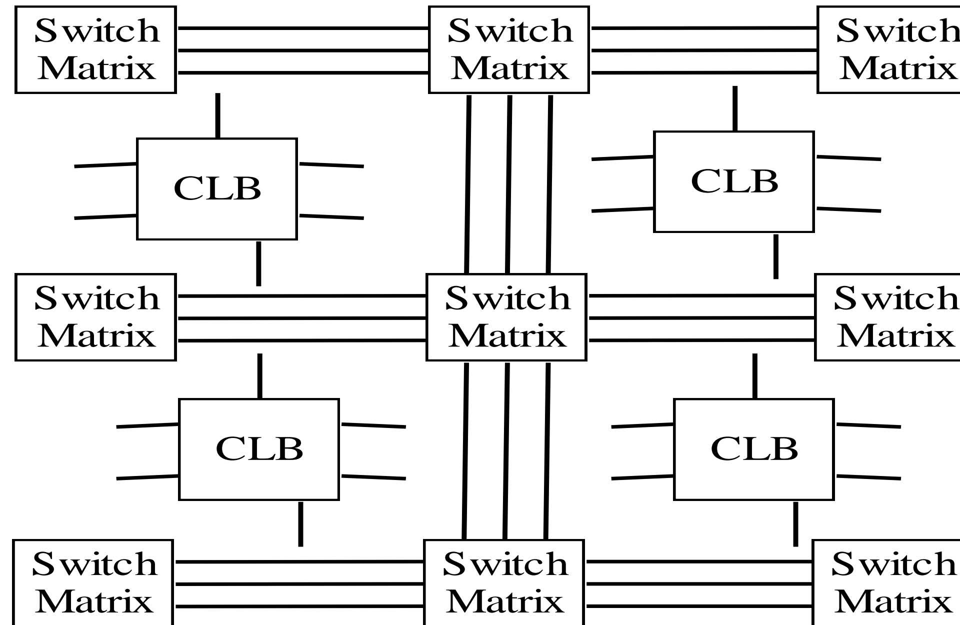
ECED 4260 – IC Design and Fabrication



Department of Electrical and Computer Engineering
Dalhousie University, Halifax, NS, Canada

COURSENOTES OUTLINE

Section I Design

0. Introduction
1. Review of Classical Sequential Logic Design
2. VHDL and Simulation
3. Digital Design and Applications
4. Design of Arithmetic Circuits
5. Digital System Testing

Section II Fabrication

1. Basic Transistor Characteristics
2. Processing Technologies

Section I. Design
0. INTRODUCTION

Classification of Integrated Circuits by Complexity

ERA	DATE	COMPLEXITY (# of logic blocks)
Single transistor	1959	<1
Unit logic	1960	1
Multi-function	1962	2-100
MSI	1967	100-1,000
LSI	1972	1,000-20,000
VLSI	1980	20,000-500,000
ULSI	1985	>500,000

Note: Boundaries are not artificially created. Crossing boundaries required new technologies and new design methodologies.

There are two common measures of the complexity:

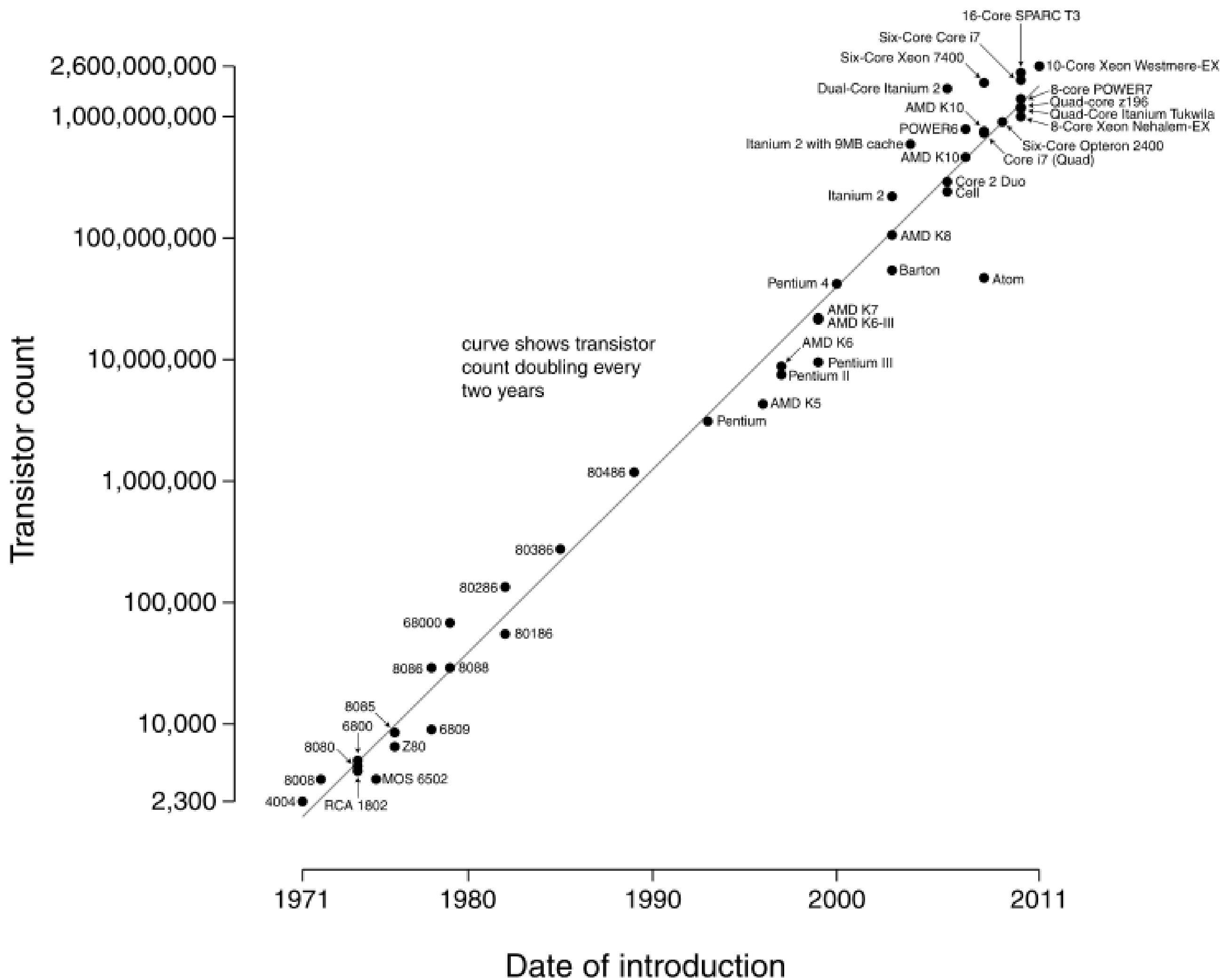
1. Scale of integration;
2. Minimum feature size.

   The number of devices per chip is determined by the minimum line widths that can be defined in the fabrication process.

0.1 Factors Influencing the IC Industry

0.1.1 Moore's Law

In 1979 Gordon Moore of Intel made the observation that chip complexity (i.e. the number of transistor per chip) was growing at an exponential rate with time.



0.1.2 Changes in Fabrication

Detailed understanding of the fabrication processes has led to better controlled and more repeatable processing. This knowledge has created "silicon foundries" companies that are solely concerned with manufacturing ICs, not necessarily the design of such ICs.

1970's – design and manufacturing was within the same company

- high volume, general purpose devices

1980's – design and manufacturing separate (silicon foundries)

- low volume special purpose ICs

0.1.3 Faster Design

Faster designs are possible if the designer works at a higher level of design. This is possible with the use of standard cells – that is the creation of a catalog of a well defined library of cells. The designer then carries out his design with these cells. Design tools such as place and route automatically make the geometrical layout required for a IC design.

0.1.4 Faster Turnaround

With a short implementation time many systems can be economically produced in silicon. With the introduction of gate arrays, which require fewer final processing steps the time from design to a completed IC is greatly reduced.

0.1.5 Lower Manufacturing Costs

With lower manufacturing costs it becomes economical to produce ICs. Lower manufacturing costs are a result of:
- higher yields, i.e. more of the manufactured ICs work.
- gate arrays, with fewer specialized processing steps.
-multi-project chips, where several designs are produced on a single wafer. The mask costs per design are shared amongst many designs.
- large wafers, more die per wafer

0.2 Economic Consideration

For the development of an integrated circuit the major areas of cost are:

- development costs
  - engineering and overhead costs
    - CAD tools can help reduce engineering time, and hence reduce costs. CAD tools are essential to carry out large designs

- mask costs
  - each photolithography step requires a mask. The number of masks may vary from 13 or higher to 6 (for a typical 2 metal gate array process)
  - each mask can cost in the neighborhood of 2 to 3 thousand dollars

• wafer and processing costs

the cost of the wafer and the processing cost involve several factors. The following table itemizes some of these costs

Item	Per Wafer	Per Die
Wafer Fabrication		
Blank wafer	x_1	
Wafer processing	x_2	
Wafer probe	x_3	
Wafer sawing	x_4	
Die attach, bonding	x_5	
Packaging		x_6
Final test		x_7

θ_{probe} – fraction of good die based on wafer probe test

$\theta_{package}$ – fraction of good die based on final die tests on packaged die

$$Cost/chip = \left[\frac{x_1 + x_2 + x_3 + x_4 + x_5}{N \times \theta_{probe}} + x_6 + x_7\right] \times \frac{1}{\theta_{package}}$$

N = number of die per wafer

Cost reduction can be achieved by

• smaller feature size – this means more die for a given wafer (yield also increases)
• larger wafers – more die for essentially the same processing effort

Originally 3 inch wafers (75mm), now 6 inch wafers (150mm) are the industry standard.

• careful processing can have a great impact on yield – purity of the clean rooms, experience and quality of the staff

Package costs will determine a large portion of the final IC cost. Some typical package cost are:

Plastic DIP	8 pin	0.032
Plastic DIP	64 pin	0.70
Ceramic DIP	64 pin	4.95
Ceramic PGA	68 pin	6.40

0.3 Yield

Yield is the percentage of die that meet the performance specifications. Yield decreases as size increases

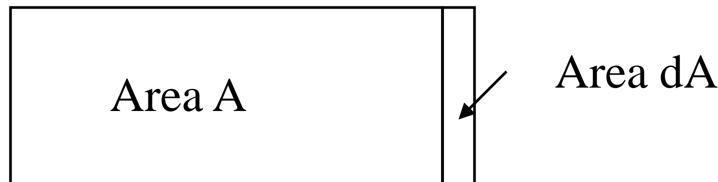
The following factors will influence the yield

dust particles
crystal defects
mask defects
alignment errors at successive masking steps
parameter variations in the process (these are called soft faults)

Reference: Poisson Distribution

A simple fault model is to assume that defects are uniformly distributed over the surface of the wafer. The average defect density is D where

D is the number of defects /cm^2



For dA sufficiently small

Prob{1 defect in dA} = D dA

Prob{0 defect in dA} = 1-D dA

i.e. Prob{2 or more defects in dA} = 0

If we assume the probability of a defect in dA is independent of the defects in A

Prob{0 defects in A+dA}=Prob{0 defects in A} x Prob{0 defects in dA}

$\frac{d}{dA}$ Prob{0 defects in A}

$$= \frac{\text{Prob\{0 defects in A+dA\}- Prob\{0 defects in A\}}}{\text{dA}}$$

$$= \frac{\text{Prob\{0 defects in A\}x [Prob\{0 defects in dA\}-1]}}{\text{dA}}$$

$$= - \frac{\text{Prob\{0 defects in A\}x D dA}}{\text{dA}}$$

$$\frac{\text{dProb\{0 defects in A\}}}{\text{Prob\{0 defects in A\}}} = \text{-D dA}$$

Prob{0 defect in A} = $e^{-D\,dA}$

Typical values are D of 2 defects /cm^2

Prob{0 defect in a die 1cm x 1cm} = e^{-2} = 0.135

Prob{0 defect in a die 0.5cm x 0.5cm} = $e^{-1/2}$ = 0.606

The Poisson Distribution is more pessimistic than experience indicates. Other yield models that are more accurate are:

1) Seed's model

$$Y = e^{-\sqrt{AD}}$$

Seed's model is used for yields less than 30%.

2) Murphy's model

$$Y = \left(\frac{1 - e^{-AD}}{AD}\right)^2$$

Murphy's model is used for yields that are greater than 30%.

3) Price's model

$$Y = \frac{1}{1 + AD}$$

Current research is concerned with determining better models to predict yield. One approach is to partition the die into different regions i.e. interconnect areas, active areas and apply different yield models to each area.

0.4 MOSFET Technology

MOS technology is the dominant technology for large scale integration. The advantage of MOS technology over bipolar are:

1) Higher device density
   30x TTL
   5x I^2L
   Individual devices are smaller
2) Can use dynamic circuits which leads to simpler circuits
3) MOS fabrication is less critical
4) MOS scales better – speed improves as device dimensions are reduced (better than bipolar)
5) More flexible technology – both analog and digital circuits as well as memory devices can be built
6) Lower power

There are some disadvantages to MOS technology when compared to bipolar technology

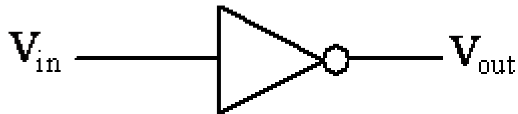
1) Slower than bipolar by a factor of 2
2) Limited current driving capability – not easily used to drive high capacity or low impedance loads

In addition to bipolar technology other viable competitors are BiCMOS and GaAs. BiCMOS is the technology that is growing most rapidly and may become the dominant technology.

1. Review of Classical Sequential Logic Design

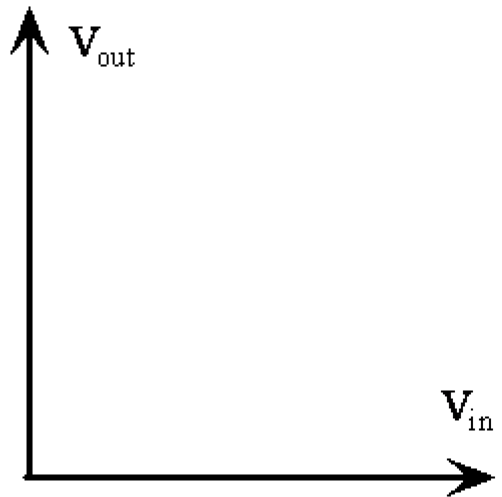
1.1 Introduction to Combinational Logic
1.2 Boolean Algebra
1.3 Mixed Logic
1.4 Algebraic Specification of Combinational Logic
1.5 Memory Elements
1.6 Synthesis Procedure
1.7 Introduction to Programmable Logic Devices (PLDs)

1.1 Introduction to Combinational Logic

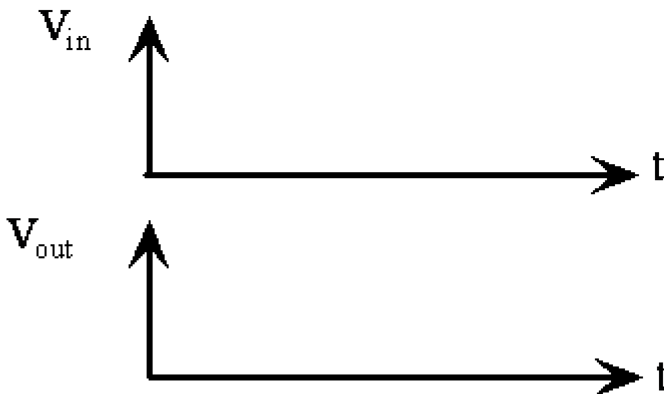


Truth Table

V_{in}	V_{out}

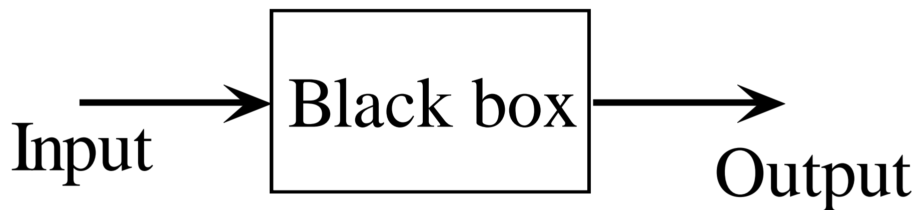


Dynamic Behavior:
Analog view



1.1 Introduction to Combinational Logic (cont'd)

Logic gates: single input

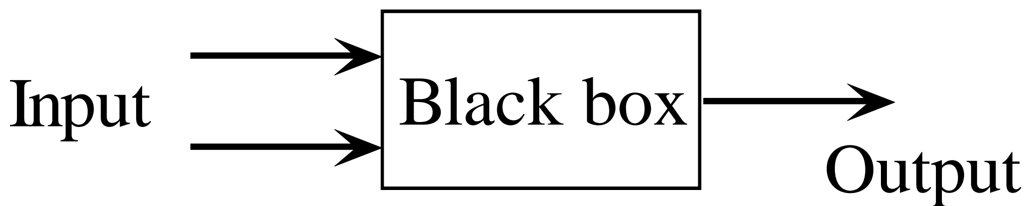


A	F1	F2	F3	F4
L	L	L	H	H
H	L	H	L	H

What are these functions?

1.1 Introduction to Combinational Logic (cont'd)

Logic gates: Two input

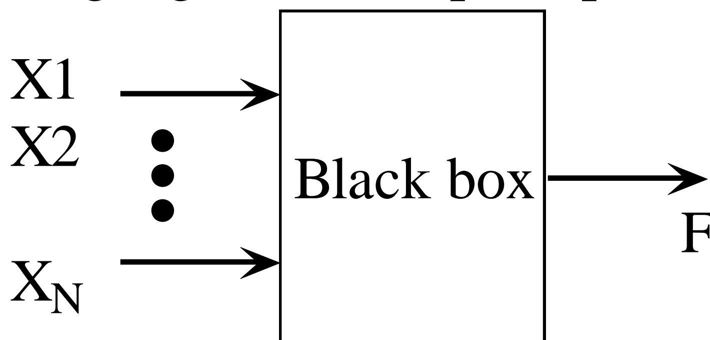


A	B	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
L	L	L	L	L													H
L	H	L	L	L													H
H	L	L	L	H													H
H	H	L	H	L													H

What are these functions?

1.1 Introduction to Combinational Logic (cont'd)

Logic gates: Multiple input



If X1=X2=…=X_N=H, then F=H; otherwise, F=L.

If X1=X2=…=X_N=L, then F=L; otherwise, F=H.

If X1=X2=…=X_N=H, then F=L; otherwise, F=H.

If X1=X2=…=X_N=L, then F=H; otherwise, F=L.

If an odd number of Xi are H, then F=H; otherwise, F=L.

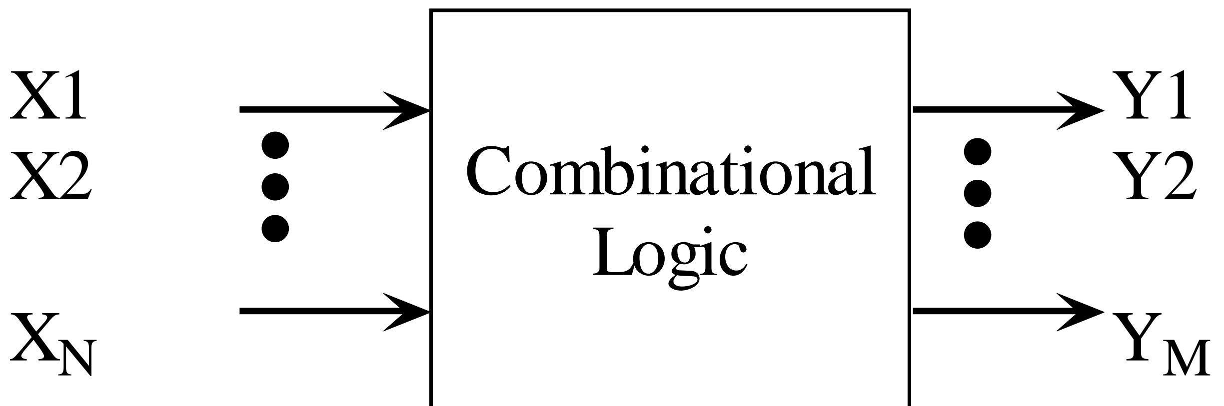
If an even number of Xi are H, then F=H; otherwise, F=L.

What are these functions?

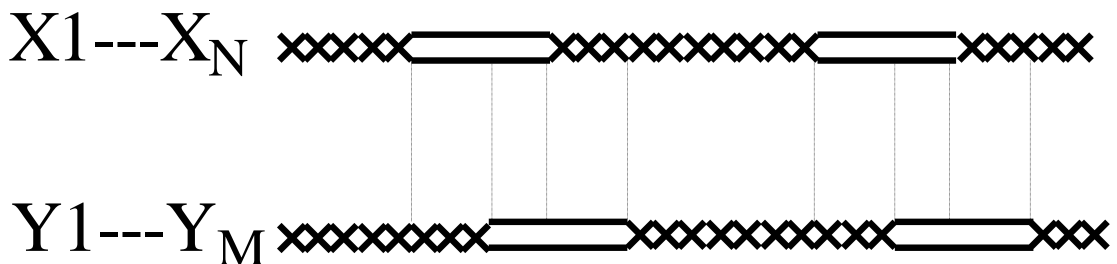
1.1 Introduction to Combinational Logic (cont'd) Remarks

- After all signals have stabilized, the present output signals are entirely determined by the present input signals
- Combinational logic has no memory of past inputs
- Combinational logic can always be implemented as a network of basic logic gates
- Typically there are no signal paths that loop back on themselves.

1.1 Introduction to Combinational Logic: (cont'd)



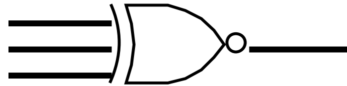
Time diagram:



1.1 Introduction to Combinational Logic

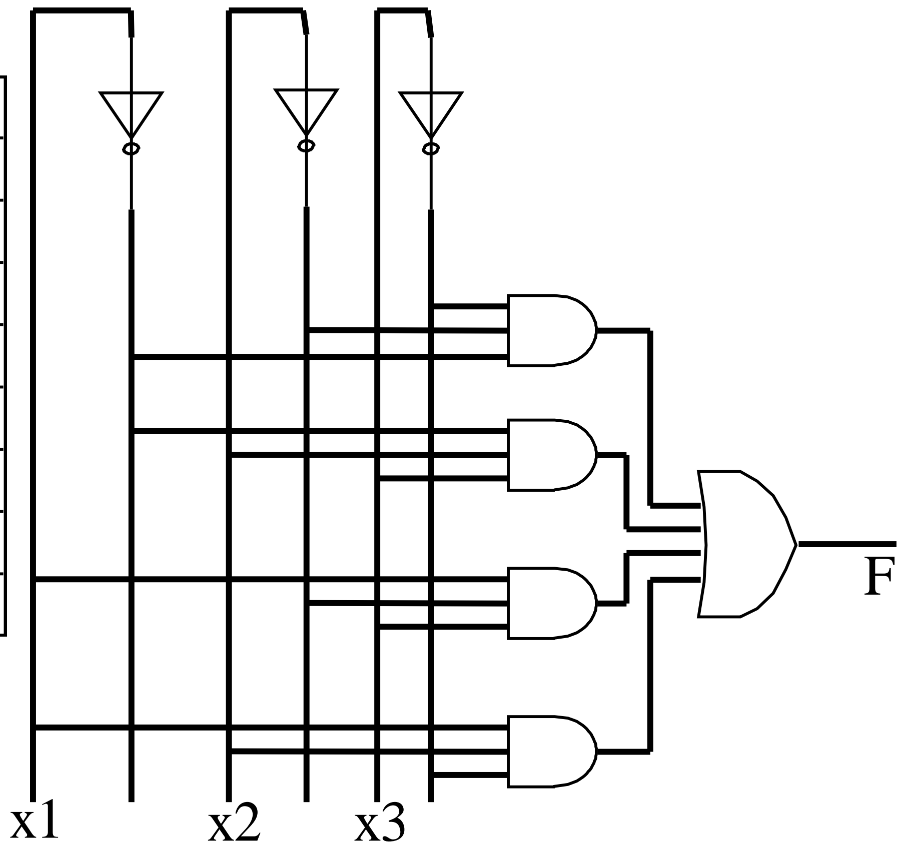
Example: Design a 3-input NXOR gate using only NOT, AND and OR gates.

Behavior:



Truth Table:

X1	X2	X3	F
L	L	L	H
L	L	H	L
L	H	L	L
L	H	H	H
H	L	L	L
H	L	H	H
H	H	L	H
H	H	H	L



1.2 Boolean Algebra

A)There is a set S of possible values. Set S must contain at least two special values, which are denoted by 0 and 1.

B) There are two binary operations, · and +, that can be applied to pairs of elements from S to produce elements from S.

C) Duality: the dual of Boolean equation is the equation obtained by interchanging all ·'s and +'s, and interchanging all 0's and 1's.

Principle of duality: A Boolean equation is true if and only if the dual Boolean equation is true.

Basic Boolean Identities

	Identity	Comments
1.	$A+0=A$	Operations with 0 and 1
2.	$A+1=1$	Operations with 0 and 1
3.	$A+A=A$	Idompotent
4.	$A+\overline{A}=1$	Complementarity
5.	$A\cdot 0=0$	Operations with 0 and 1
6.	$A\cdot 1=A$	Operations with 0 and 1
7.	$A\cdot A=A$	Idompotent
8.	$A\cdot\overline{A}=0$	Complementarity
9.	$\overline{\overline{A}}=A$	Involution
10.	$A+B=B+A$	Commutative
11.	$A\cdot B=B\cdot A$	Commutative
12.	$A+(B+C)=(A+B)+C=A+B+C$	Associative
13.	$A\cdot(B\cdot C)=(A\cdot B)\cdot C=A\cdot B\cdot C$	Associative
14.	$A\cdot(B+C)=(A\cdot B)+(A\cdot C)$	Distributive
15.	$A+(B\cdot C)=(A+B)\cdot(A+C)$	Distributive
16.	$A+(A\cdot B)=A$	Absorption
17.	$A\cdot(A+B)=A$	Absorption
18.	$(A\cdot B)+(\overline{A}\cdot C)+(B\cdot C)=(A\cdot B)+(\overline{A}\cdot C)$	Consensus
19.	$\overline{A+B+C+\ldots}=\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\ldots$	De Morgan
20.	$\overline{A\cdot B\cdot C\cdot\ldots}=\overline{A}+\overline{B}+\overline{C}+\ldots$	De Morgan
21.	$(A+\overline{B})\cdot B=A\cdot B$	Simplification
22.	$(A\cdot\overline{B})+B=A+B$	Simplification

1.3 Mixed Logic

- Draw backs of entirely positive or negative logic conventions:
  - The rules for synthesizing and analyzing multiple level NAND/NOR networks from AND/OR networks are complicated
  - It is difficult sometimes to read off Boolean equations from a circuit diagram that uses only positive logic
  - Mixed logic is the practice of using both positive and negative signal encoding on the same circuit diagram.
  - Mixed logic is also called "direct polarity indication"
  - If used properly, mixed logic can improve the readability of circuit diagrams.

1.3 Mixed Logic: Main Ideas

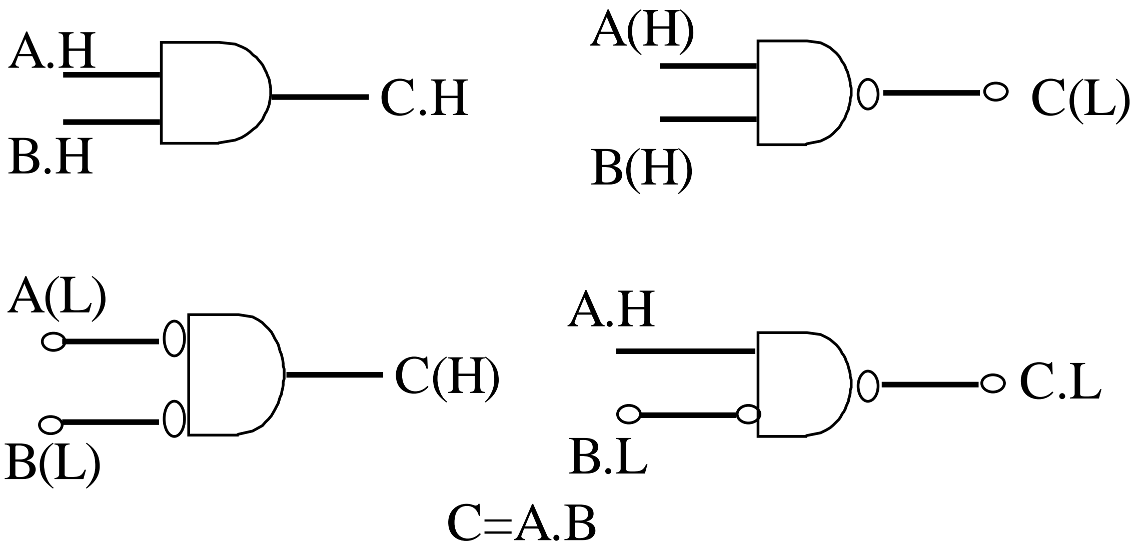
- The shape of a gate symbol should reflect the logical function
- Bubbles are used on all inputs and outputs to indicate how Boolean 0's and 1's are encoded using physical L's and H's
  - No bubble → positive logic
  - Bubble → negative logic

1.3 Mixed Logic: Notations

Logic signal	Positive logic	Negative logic
A	A.H	A.L
B	B(H)	B(L)
C	C	C/
$\overline{D}$	$\overline{D}.H$	$\overline{D}.L$

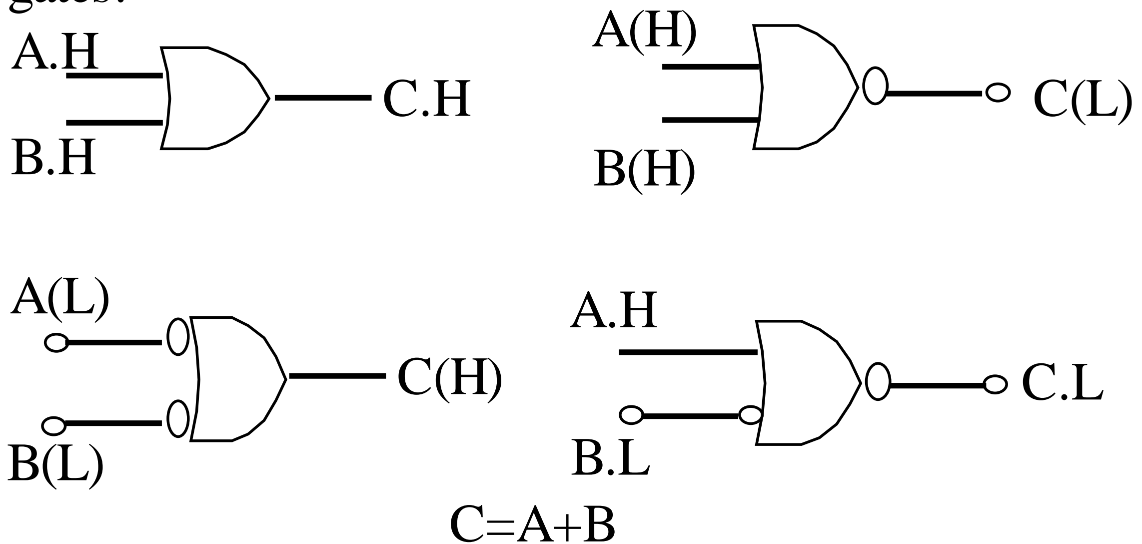
1.3 Mixed Logic:

AND gates:



C=A.B

OR gates:



C=A+B

1.4 Algebraic Specification of Combinational Logic

- Definitions
  - A literal is a variable or its complement form
  - A minterm of n variables is product of n literals in which each variable appears exactly once in either true or complement form
  - A maxterm if n variable is a sum of n literals in which each variable appears exactly once in either true or complement form.

1.4 Algebraic Specification of Combinational Logic

1). Boolean Function Truth Table

Mi	X1	X2	X3	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

X1	X2	X3	F
L	L	L	L
L	L	H	H
L	H	L	H
L	H	H	L
H	L	L	H
H	L	H	L
H	H	L	L
H	H	H	H

2) As a sum of minterms:

$$F(x_1, x_2, x_3) = m_1 + m_2 + m_4 + m_7$$
$$= \overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1\overline{x_2}\,\overline{x_3} + x_1x_2x_3$$

3) As a product of maxterms

$$F(x_1, x_2, x_3) = M_0 * M_3 * M_5 * M_6$$
$$= (X1 + X2 + X3)(X1 + \overline{X2} + \overline{X3})(\overline{X1} + X2 + \overline{X3})(\overline{X1} + \overline{X2} + X3)$$

1.4.1 Simplification of Boolean Functions

- Motivation:
  - Simpler circuit, fewer gates, save money
- Definition
  - Implicants: a group of 2^n adjacent 1's in a k-map.
  - Prime implicant: an implicant that cannot be doubled in size by applying the adjacency theorem
  - Minimum cover: smallest set of prime implicants that covers all the 1's or 0's
  - Minimum S.O.P: a sum of products that corresponding to a minimum cover of the 1's
  - Minimum P.O.S: a product of sums that corresponds to a minimum cover of the 0's

1.4.2 Karnaugh Maps

- A Karnaugh Map is a binary truth table in which the function values are arranged to facilitate the visual detection of of logical relationships and therefore the manual application of the laws and theorems of Boolean Algebra
- Find the F of following:

BC \ A	0	1
00	1	0
01	1	1
11	0	1
10	0	0

1.4.2 Example 1.3

Design a two-level combinational circuit that converts from binary coded decimal(BCD) to excess 3 code

	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
	1	0	1	0	Φ	Φ	Φ	Φ
	1	0	1	1	Φ	Φ	Φ	Φ
	1	1	0	0	Φ	Φ	Φ	Φ
	1	1	0	1	Φ	Φ	Φ	Φ
	1	1	1	0	Φ	Φ	Φ	Φ
	1	1	1	1	Φ	Φ	Φ	Φ

W

CD \ AB	00	01	11	10
00			Φ	
01			Φ	
11			Φ	Φ
10			Φ	Φ

X

CD \ AB	00	01	11	10
00	0	1	Φ	0
01	1	0	Φ	1
11	1	0	Φ	Φ
10	1	0	Φ	Φ

Y

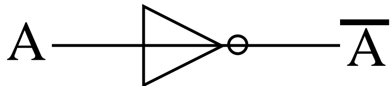
CD \ AB	00	01	11	10
00	1	1	Φ	1
01	0	0	Φ	0
11	1	1	Φ	Φ
10	0	0	Φ	Φ

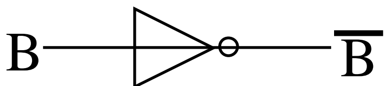
Z

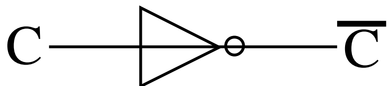
CD \ AB	00	01	11	10
00	1	1	Φ	1
01	0	0	Φ	0
11	0	0	Φ	Φ
10	1	1	Φ	Φ

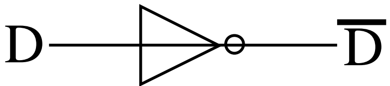
1.4.2 Example 1.3

Design a two-level combinational circuit that converts from binary coded decimal(BCD) to excess 3 code







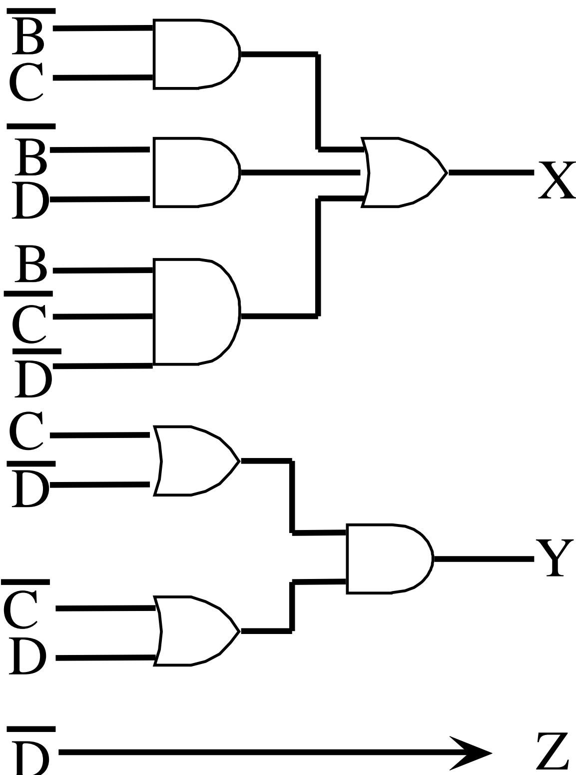


$$W =$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D}$$

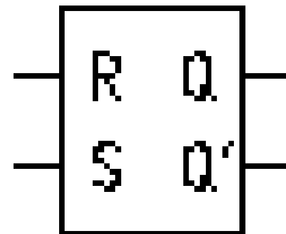
$$Y = \left(C + \overline{D}\right)\left(\overline{C} + D\right)$$

$$Z = \overline{D}$$

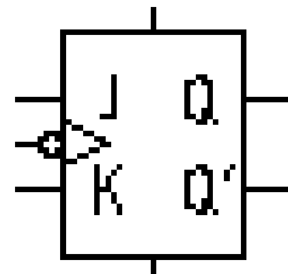


1.5 Memory Elements: Latches

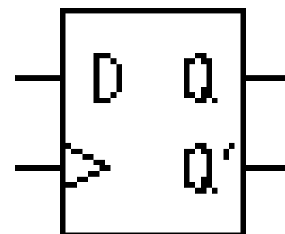
SR latches:



JK Flip-Flop:



D Flip-Flop:

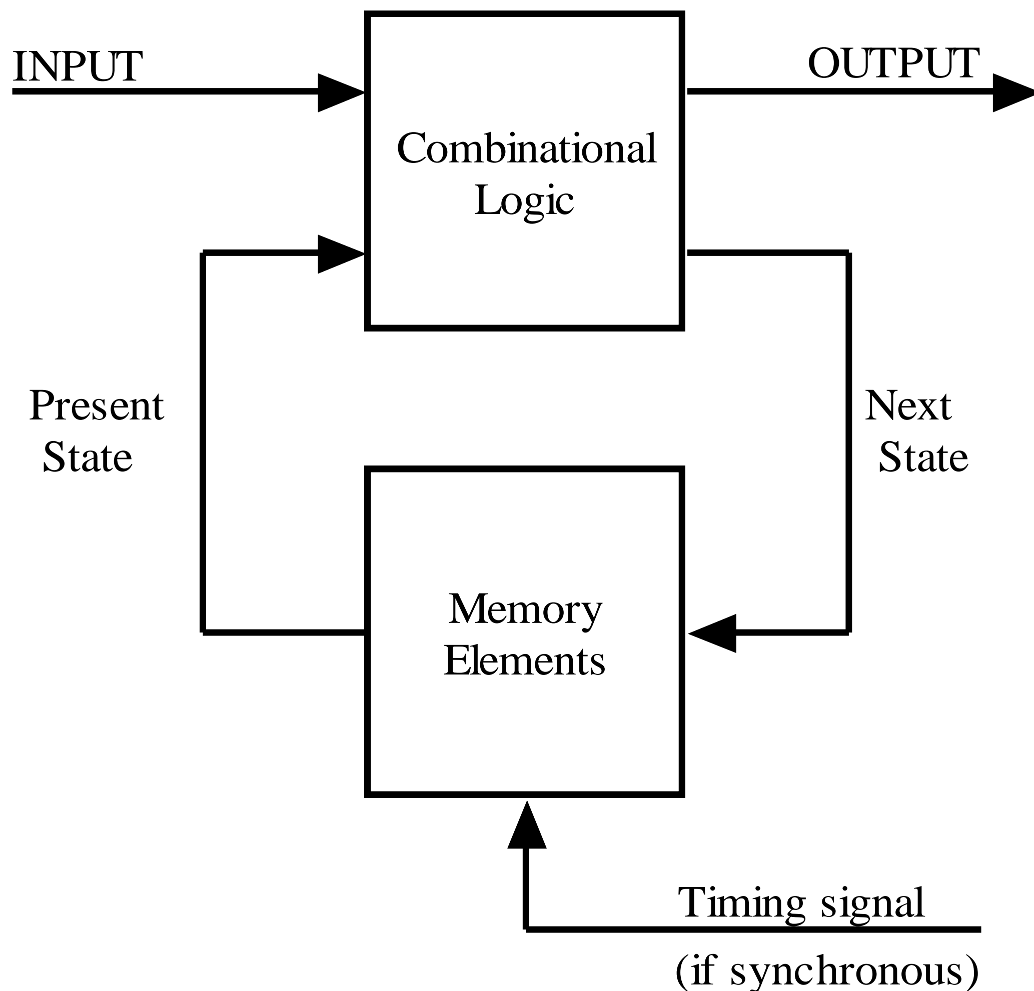


1.5 Sequential Logic: Remarks

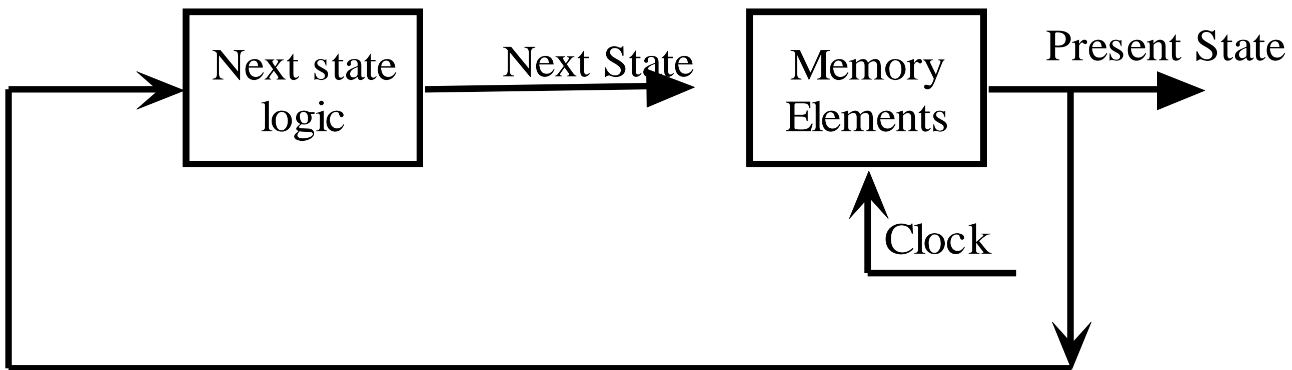
- After all signals have stabilized, present output signals are determined by both the present and past input signals
- Circuit has closed loops with implement memory elements.
- Two class of sequential logic:
  - Synchronous sequential logic: a timing signal is used.
  - Asynchronous sequential logic: no timing signal is used. The memory elements are free to change state immediately after input signals change

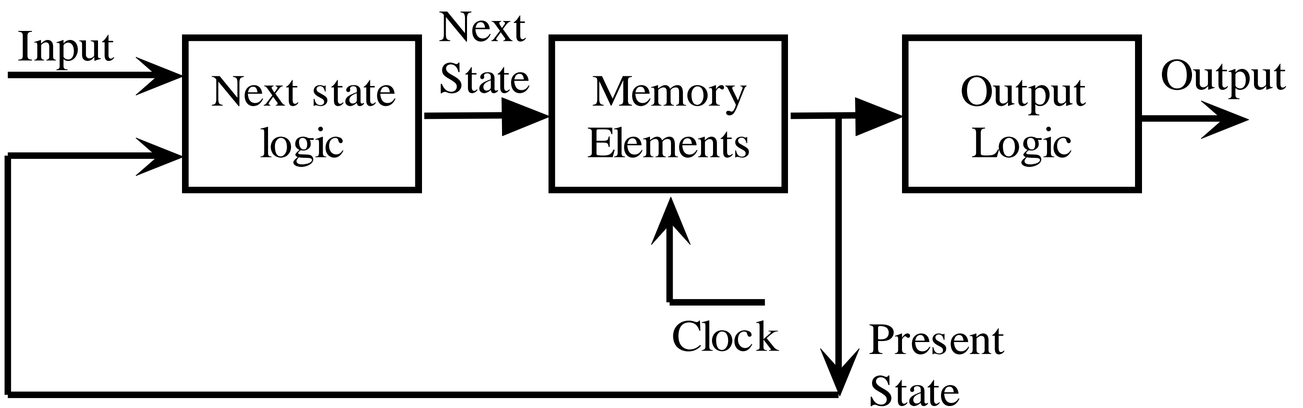
1.5 Sequential Logic: Remarks (cont'd)

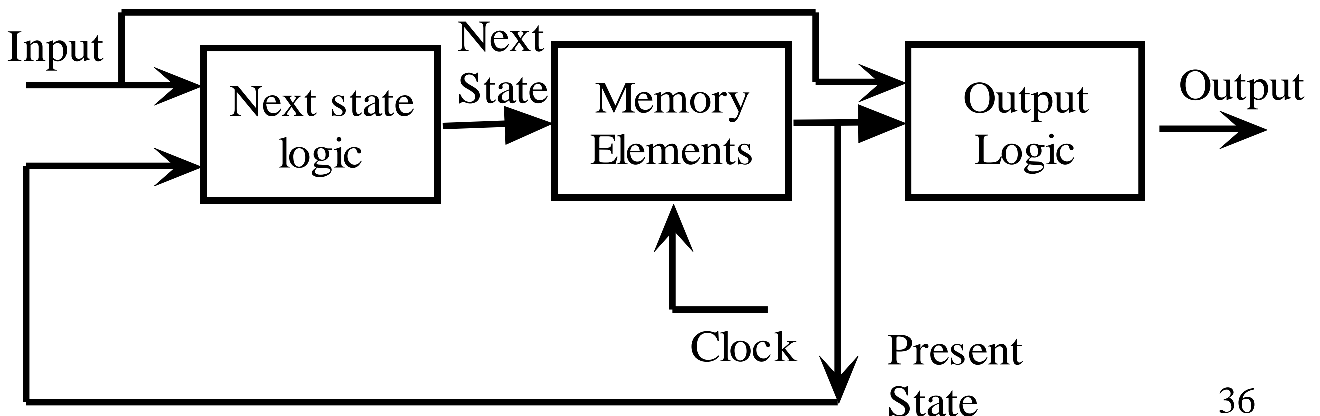
- General structure of a sequential circuit:



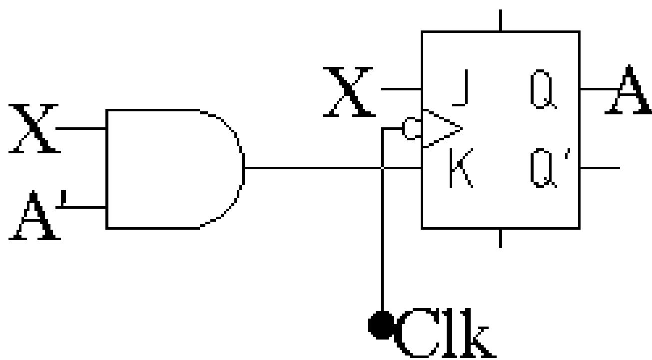
1.5 Sequential Logic: Special Classes







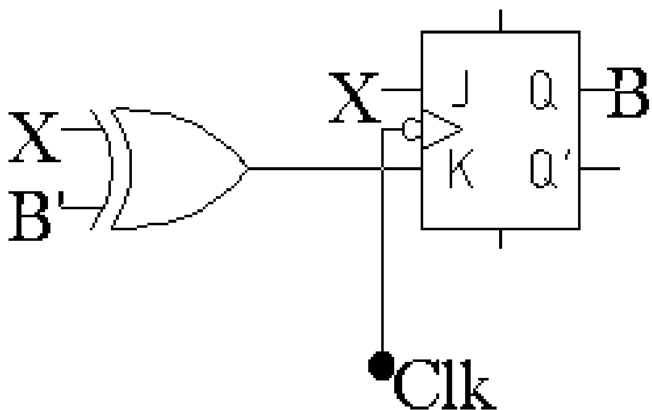
1.5 Sequential Logic: Example 1.4



X: input

A, B, states

$Q^{+} = Q\overline{K} + \overline{Q}J$

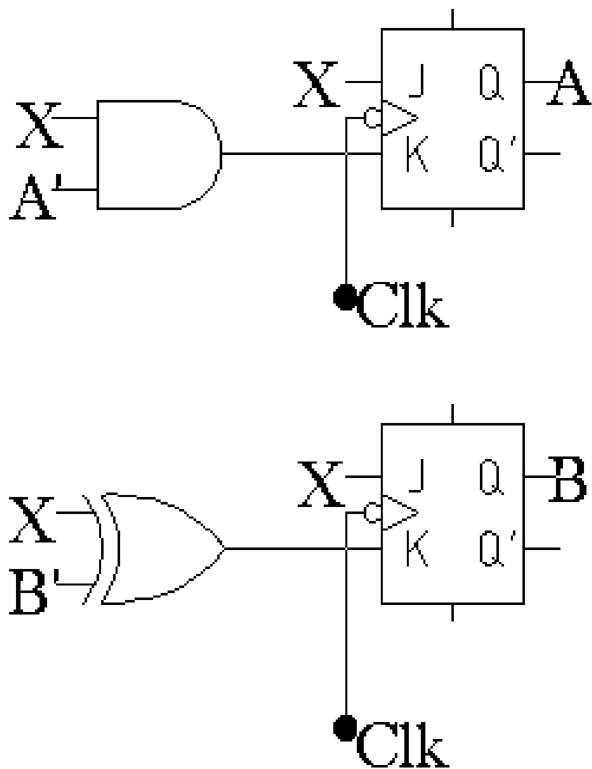


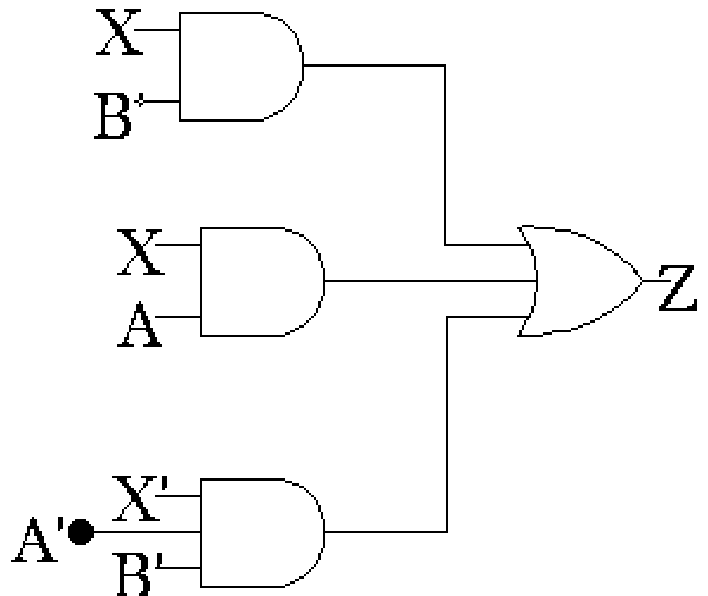
$A^{+} = A*\overline{(X\overline{A})} + \overline{A}X$

$B^{+} = B\overline{(X \oplus \overline{B})} + \overline{B}X$

$Z = B$

1.5 Sequential Logic: Example 1.4





X: input

A, B, states

$Q^{+} = Q\overline{K} + \overline{Q}J$

$A^{+} = A*\overline{(X\overline{A})} + \overline{A}X$

$B^{+} = B\overline{(X \oplus \overline{B})} + \overline{B}X$

$Z = X\overline{B} + XA + \overline{X}\,\overline{A}\,\overline{B}$

1.5 Sequential Logic: Remarks

- Moore outputs: depend on present state only
- Mealy outputs depend on present state and input

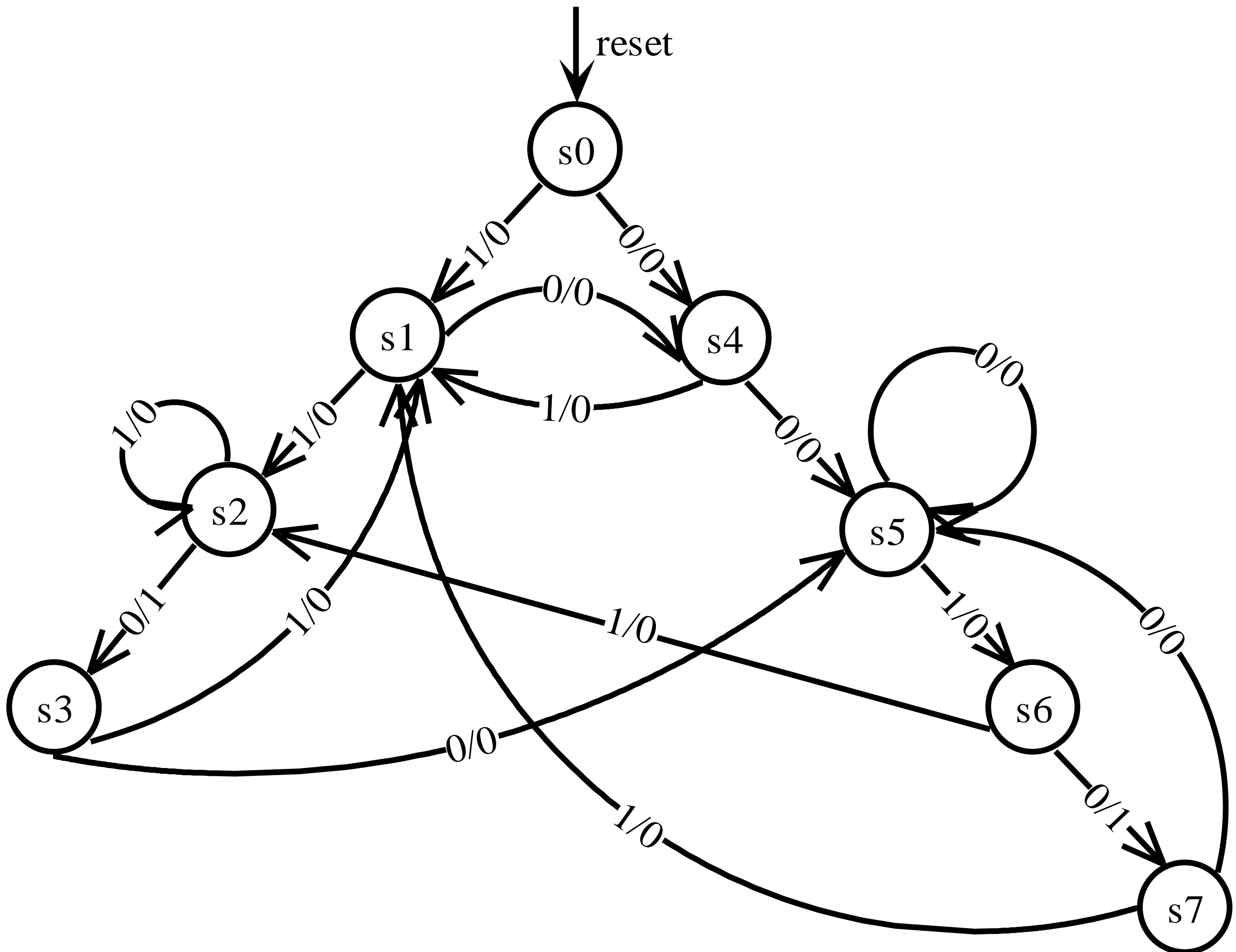
1.6 Synthesis Procedure

- From the behavior specification derive a symbolic state table
- Reduce the state table by merging equivalent states
- Find a promising state assignment
- Construct the binary state table
- Find minimum flip-flop input functions
- Find minimum output functions.
- Draw the complete schematic diagram
- Verify the design using simulation and /or analysis

Example 1.5: Design a Mealy Machine

- Input X and output Z
- Output Z=1 whenever either of the sequences 110 or 0010 has just been observed. At all other times Z is set to 0. Note that the machine does not reset once one of the two sequences is detected.

Example 1.5 Step1: Derive the Symbolic State Table



Example 1.5 Step1: Derive the Symbolic Table

Present state	Next state		Output, Z	
State	X=0	X=1	X=0	X=1
S0	s4	S1	0	0
S1	S4	S2	0	0
S2	S3	S2	1	0
S3	S5	S1	0	0
S4	S5	S1	0	0
S5	S5	S6	0	0
S6	S7	S2	1	0
S7	s5	s1	0	0

Example 1.5 Step2: Reduce the State Table

Equivalent state: same next state, same output

S2=s6 iff s3=s7

S3=s4=s7

S1							
S2							
S3							
S4							
S5							
S6							
S7							
	S0	S1	S2	S3	S4	S5	S6

Example 1.5 Step2: Reduce the State Table

Present state	Next state		Output, Z	
State	X=0	X=1	X=0	X=1
S0	S3	S1	0	0
S1	S3	S2	0	0
S2	S3	S2	1	0
S3	S5	S1	0	0
S5	S5	S2	0	0

Draw the reduced state diagram…

Example 1.5 Step3: Find a Promising State Assignment

- Heuristic rules--states which satisfy the following constraints should be given adjacent binary assignments:
  - 1. States that have the same next state for a given input
  - 2. States that are the next states of the same state
  - 3. States that have the same output for the same input.

Example 1.5 Step3: Find a Promising State Assignment

- States that have the same next state for a given input
  - {s0,s1,s2} {S3,S5} x=0
  - {s0,s3} {s1,s2,s5} x=1
- states that are the next state of the same state
  - So{s3,s1} S1{s2,s3} s2{s2,s3}
  - S3{s1,s5} s5{s2 s5}
- States that have the same output for the same input.
  - X=0 {s0 s1 s3 s5}
  - X=1 {s0,s1,s2,s3,s5}

Example 1.5 Step3: Final State Assignment

BC \ A	0	1
00	s0	s2
01	s1	s5
11		
10		s3

BC \ A	0	1
00	s0	
01	s1	
11	s2	
10	s5	s3

Final state assignment				
State	A	B	C	
S0	0	0	0	
S1	0	0	1	
S2	1	0	0	
S3	1	1	0	
S5	1	0	1	

Example 1.5: Step4: Construct the Binary State Table

Present	Next State		output	
State,ABC	X=0	X=1	X=0	X=1
000	110	001	0	0
001	110	100	0	0
100	110	100	1	0
110	101	001	0	0
101	101	100	0	0

Example 1.5 Step5,6: Find Minimum Flip-flop Input & Output Functions

A+

BC \ XA	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	Φ	Φ	Φ	Φ
10	Φ	1	0	Φ

B+

BC \ XA	00	01	11	10
00	1	1	0	0
01	1	0	0	0
11	Φ	Φ	Φ	Φ
10	Φ	0	0	Φ

C+

BC \ XA	00	01	11	10
00	0	0	0	1
01	0	1	0	0
11	Φ	Φ	Φ	Φ
10	Φ	1	1	Φ

Z

BC \ XA	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	Φ	Φ	Φ	Φ
10	Φ	0	0	Φ

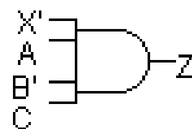
$$\overline{A^{+}} = X\bar{A}\bar{C} + BX$$

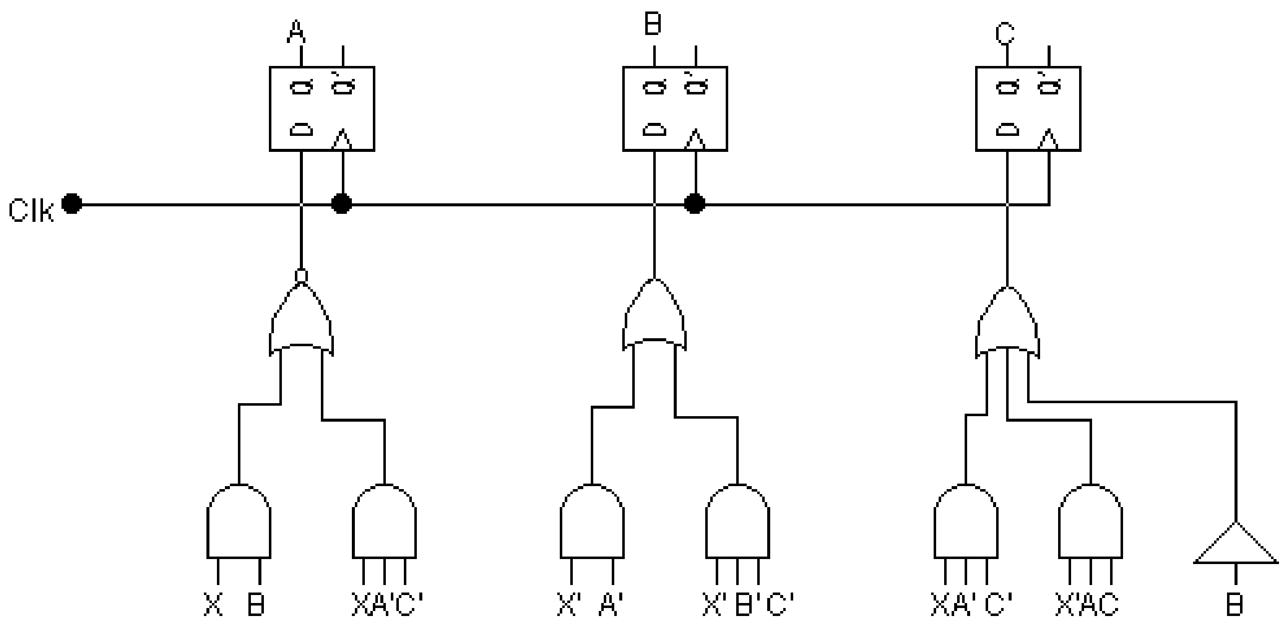
$$C^{+} = B + \bar{X}AC + X\bar{A}\bar{C}$$

$$B+ = \bar{X}\bar{A} + \bar{X}\bar{B}\bar{C}$$

$$Z = \bar{X}A\bar{B}\bar{C}$$

Example 1.5 Step7: D Flip-Flop Implementation





Example 1.5: Step7: J-K Flip Flop Implementation

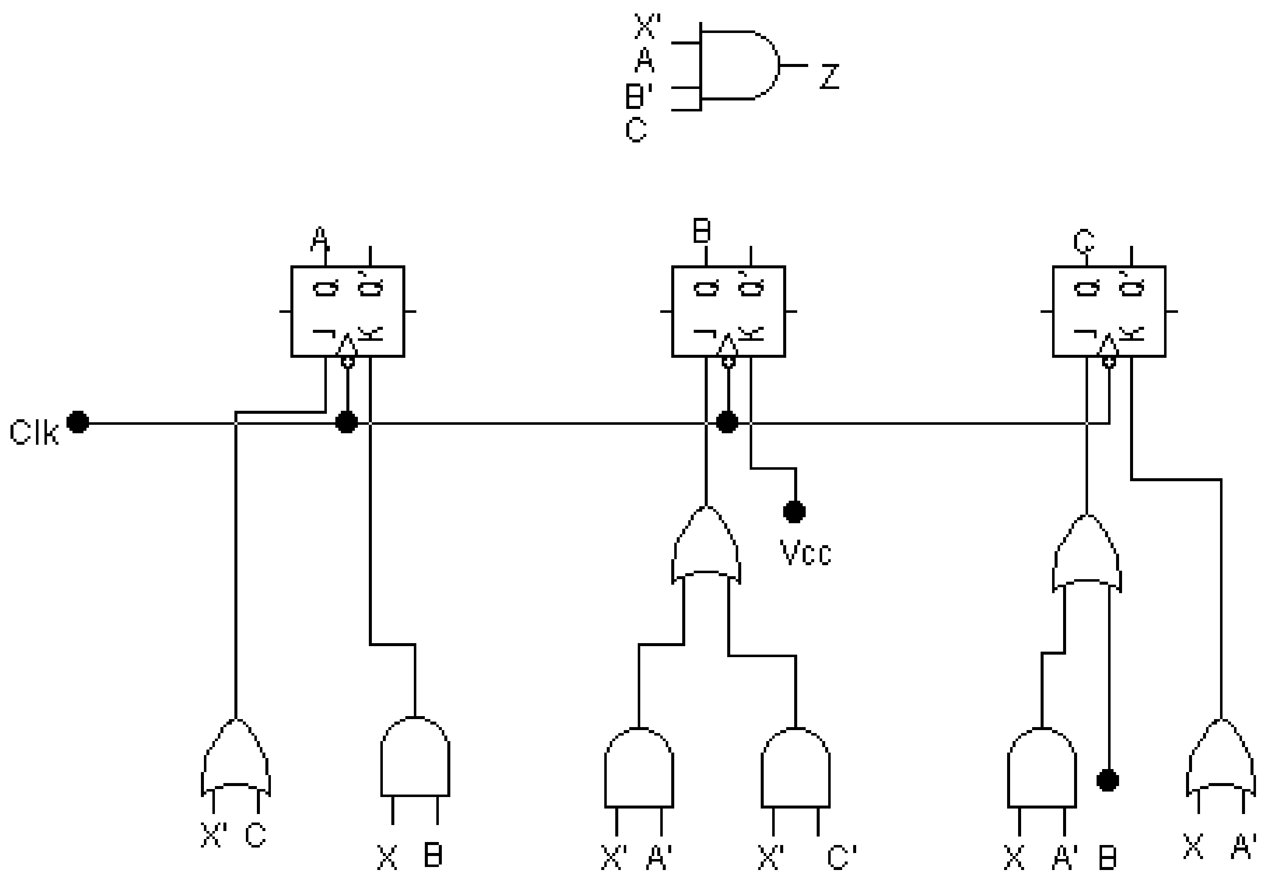
$$Q^{+} = \overline{Q}J + Q\overline{K}$$

JA: Set A=0 choose 1	JB: Set B=0 choose 1	JC: Set C=0 choose 1
KA: set A=1 choose 0	KB: set B=1 choose 0	KC: set C=1 choose 0

$J_A = \overline{X} + C$ $\quad J_B = \overline{X}\,\overline{C} + \overline{X}\,\overline{A}$ $\quad J_C = B + X\overline{A}$

$K_A = XB$ $\quad K_B = 1$ $\quad K_C = X + \overline{A}$

Example 1.5: Step7: J-K Flip Flop Implementation



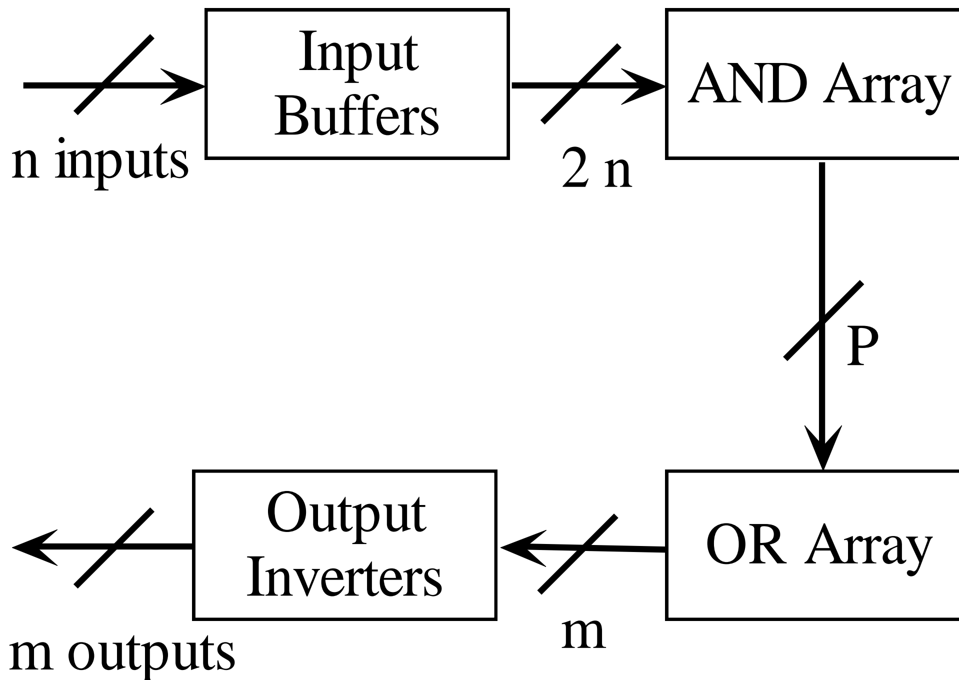
Example 1.5 Step8: Verification

- How to verify it?

1.7: Programmable Logic Devices (PLDs)

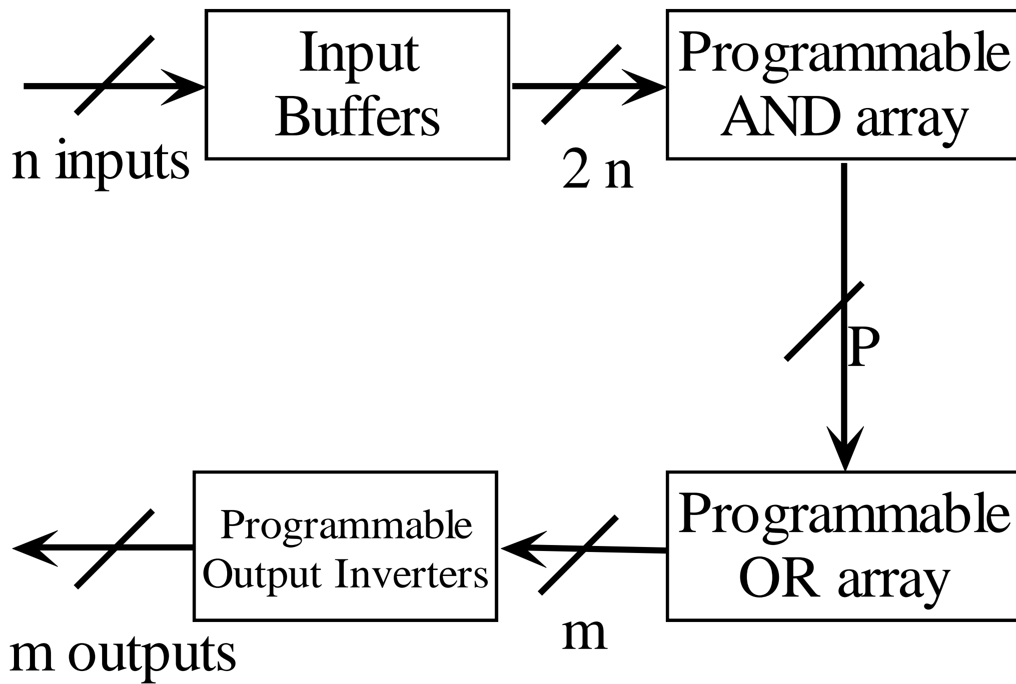
- Device that can be programmed to implement a wide variety of possible Boolean functions
- Programming is performed, other by the user, using an inexpensive programmer
- Frequently one PLD can be used to replace many SSI/MSI devices.

1.7.1General PLD Block Diagram:



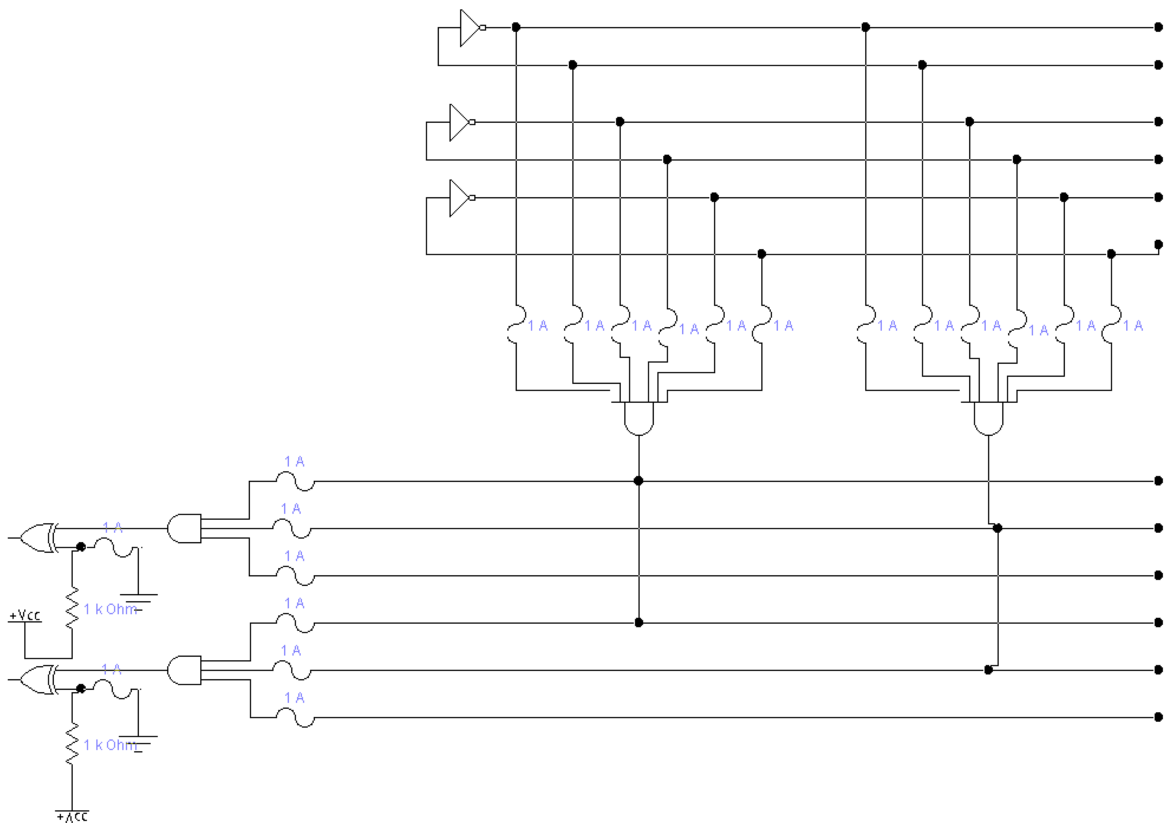
- Four main types of PLDs:
    - PLAs, PALs, PROMs, FPGAs

1.7.2 Programmable Logic Arrays (PLAs)

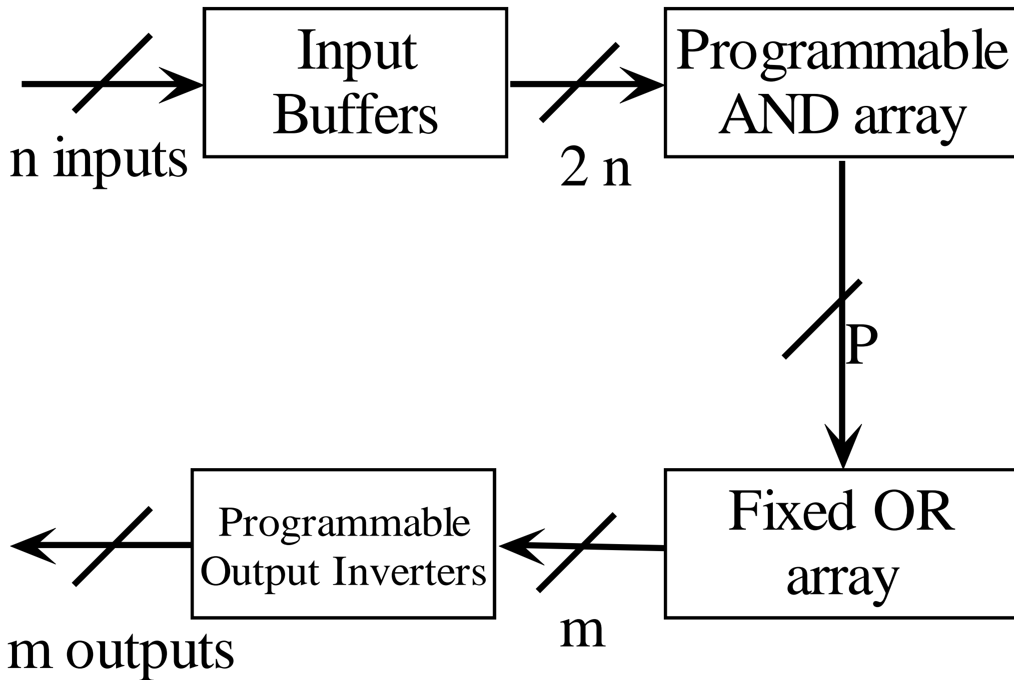


- Remarks:
  - PLAs are most efficient when the output functions share many product terms.
  - PLAs are more flexible than PALs or PROMs

1.7.2 Programmable Logic Arrays (PLAs)



1.7.3 Programmable Array Logic (PALs)

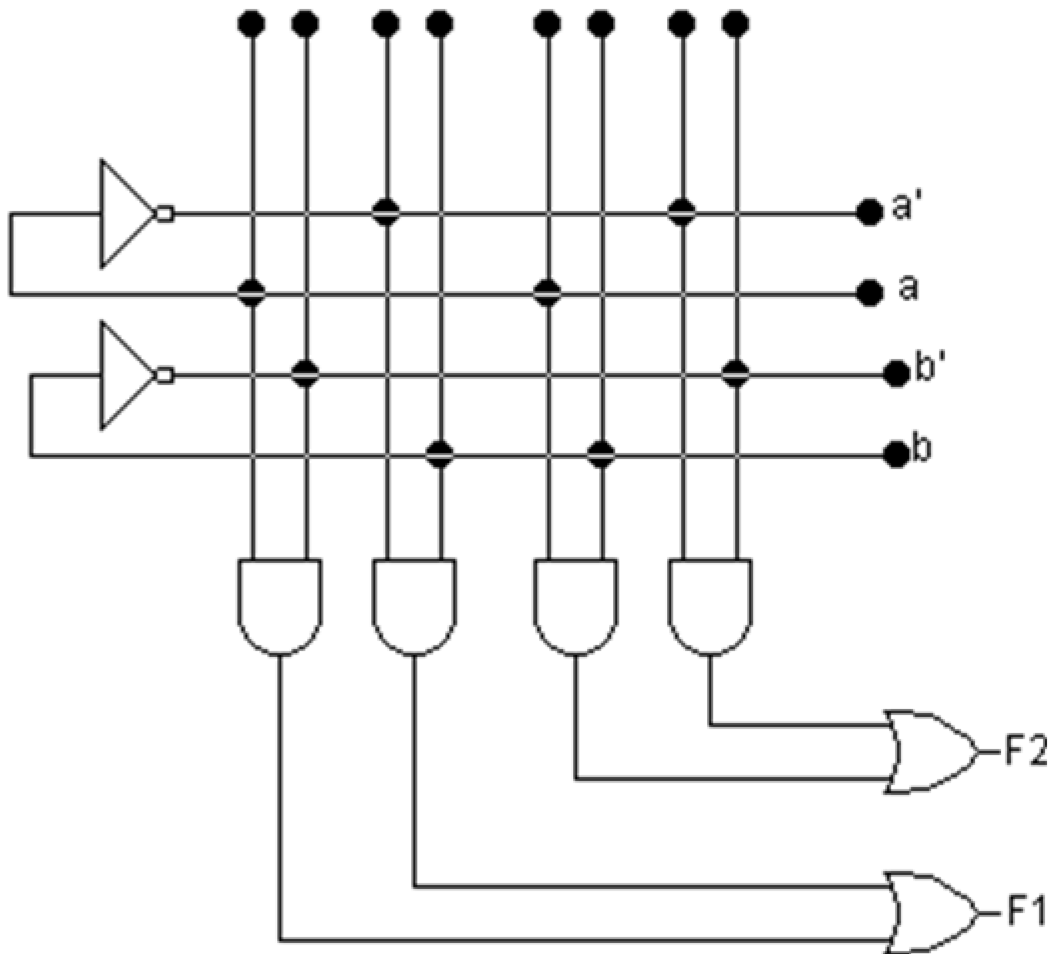


- PALs are most useful when the desired output functions do not share product terms

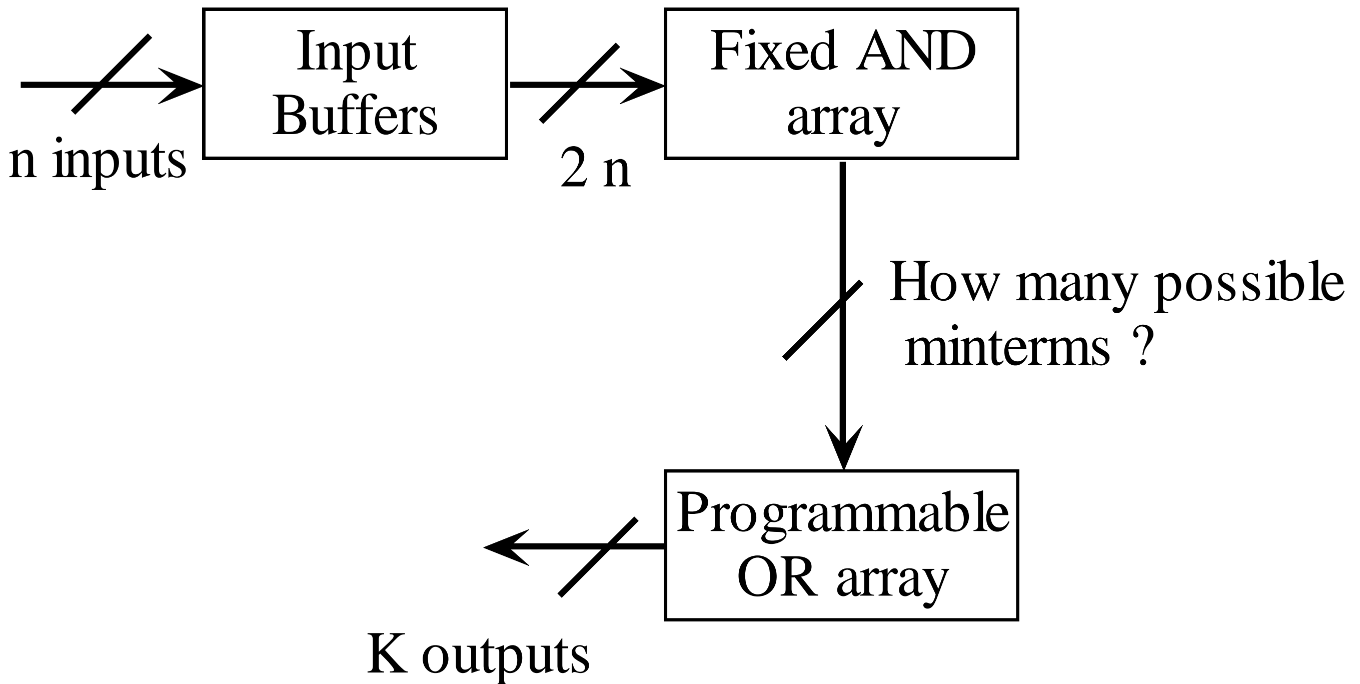
1.7.3 Example 1.6

- Use a PAL to implement the following two functions.

$F1 = a\overline{b} + \overline{a}b$ $\quad F2 = ab + \overline{a}\,\overline{b}$



1.7.4: ROM



- Remarks:
  - Essentially, a ROM stores the entire truth table of one or more Boolean functions.
  - Different types of ROMs:
    - Mask ROM: programmable once at the factory
    - PROM: programmable once by the user
    - EPROM: can be programmed and erased many times
    - Flash memory: electrically re-programmable ROM

1.7.5: FPGAs

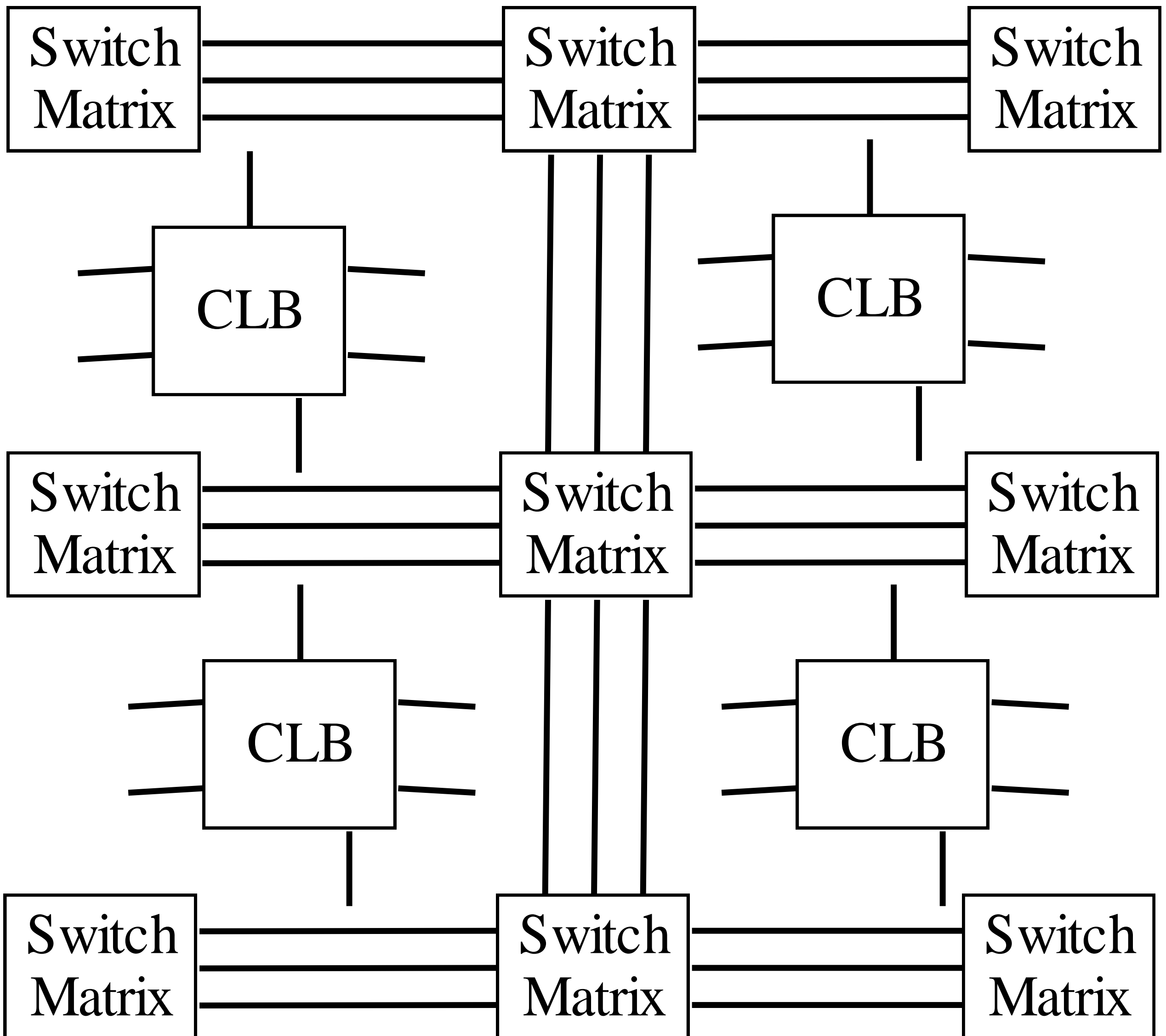
A **Gate array** is an integrated circuit containing pre placed, but unconnected, transistor clusters and/or logic gates. A final layer of metallization is etched at the factory to form the interconnections that join up the pre-placed elements into a useful circuit.

A **field-programmable gate array** is a programmable integrated circuit containing pre-placed logic gate circuits that ate interconnected by wiring and fuse or anti-fuse-programmable switches. The programming step can be performed "in the field" using a PC controller programmer unit

PLAs, PALs, and PROMs Vs. FPGAs

PLAs, PALs, and PROMs	FPGAs
✓Transistor elements	✓Gate element
✓Two-level implementation	✓Multi-level implementation
✓Bus connection	✓Matrix connection
✓Fuse programming	✓Fuse or anti-fuse programming

1.7.5: Example 1.7 Architecture of the Xilinx FPGA



CLB: Configurable Logic Block

II. VHDL & Simulation

2. VHDL and Simulation

2.1 Introduction to VHDL
2.2 Scalar Data Types and Operations
2.3 Sequential Statement
2.4 Composite Data Types and Operations
2.5 Modeling Constructs
2.6 Subprograms & Packages & use clause
2.7 Resolved Signals & Generic Constants
2.8 Components and Configurations
2.9 Logic Simulation
2.10 Predefined Environments

2.1 Introduction to VHDL

- Conventional Hardware Specification
  - Truth tables
  - Boolean equations
  - State diagrams
  - Pseudo-code behavioral algorithms
  - Schematic diagrams
  - Netlist, proprietary cad formats
- Advantages
  - Similar to methods in other engineering areas
  - Familiar, graphical
- Disadvantages
  - Too many different method
  - Specification languages typically are not defined in syntax or semantics
  - Specifications are not typically manipulatable

2.1 Introduction to VHDL (cont'd)

- Motivation of HDLs
  - Obtain benefits of an unambiguous, standard specification language
  - Facilitate use of computer-aided design (CAD) and computer-aided engineering tools
  - Facilitate exploration of rapidly improving logic synthesis technology
  - Increase designer efficiency, permit rapid prototyping, reduce time-to market, etc.
- Some Common HDLs
  - ABEL
  - Verilog-HDL(cadence, IEEE 1364)
  - VHDL (US DoD, now IEEE 1164 and 1076

2.1 Introduction to VHDL cont'd: Synthesis Technology

Evolution of synthesis technology

- logic minimization software
- PLA synthesis software
- Multiple-level combination logic synthesis
- Sequential logic synthesis
- Automatic mapping to gate arrays, standard cells, (PLDs), FPGAs
- VHDL and logic synthesis
- VHDL provided a key platform for commercializing logic synthesis technology
- IEEE standard 1076.3 defines a subset of VHDL for use by logic synthesis tools

2.1.1History of VHDL

- Very high speed Integrated Circuit Program
  - The US department of Defense funded the VHSIC program in the 1970’s and 1980’s to promote the improvement of semiconductor technology
  - One product of the VHSIC program was VHDL
- Originals of VHDL
  - To improve documentation of complex hardware designs and thus improve the ability to subcontract the design of military systems
  - To provide a standard modeling and simulation language
- Initial Standards: IEEE 1076 (1987)

2.1.2 Fundamental Concept:a Simple Design

1 - - eqcomp4 is a four bit equality comparator

2 **entity** eqcomp4 **is**

3 **port** (a, b: **in** bit_vector(3 **downto** 0);

4 equals: **out** bit); -- equals is output

5 **end entity** eqcomp4;

6

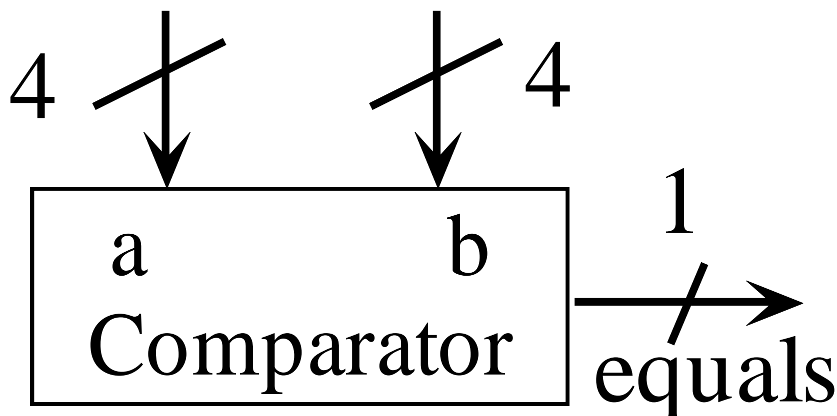
7 **architecture** dataflow **of** eqcomp4 **is**

8 **begin**

9 equals <= ‘1’ **when** (a=b) **else** ‘0’;

10 **end architecture** dataflow;

11 - - end of the program



2.1.2 Example 2.1

Design an entity of a three input And gate

1 - - three input and gate

2 **entity** and3 **is**

3 **port** (a, b, c : **in** bit;

4 d: **out** bit); -- d is output

5 **end entity** and3;

6

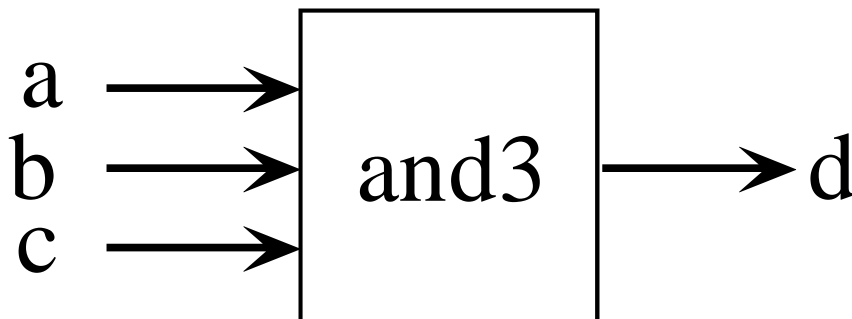
7 **architecture** structure1 **of** and3 **is**

8 **begin**

9 d <= a **and** b **and** c;

10 **end architecture** structure1;

11 - - end of the program



2.1.2 Example 2.2 : An N-bit Counter

Entity Counter **is**

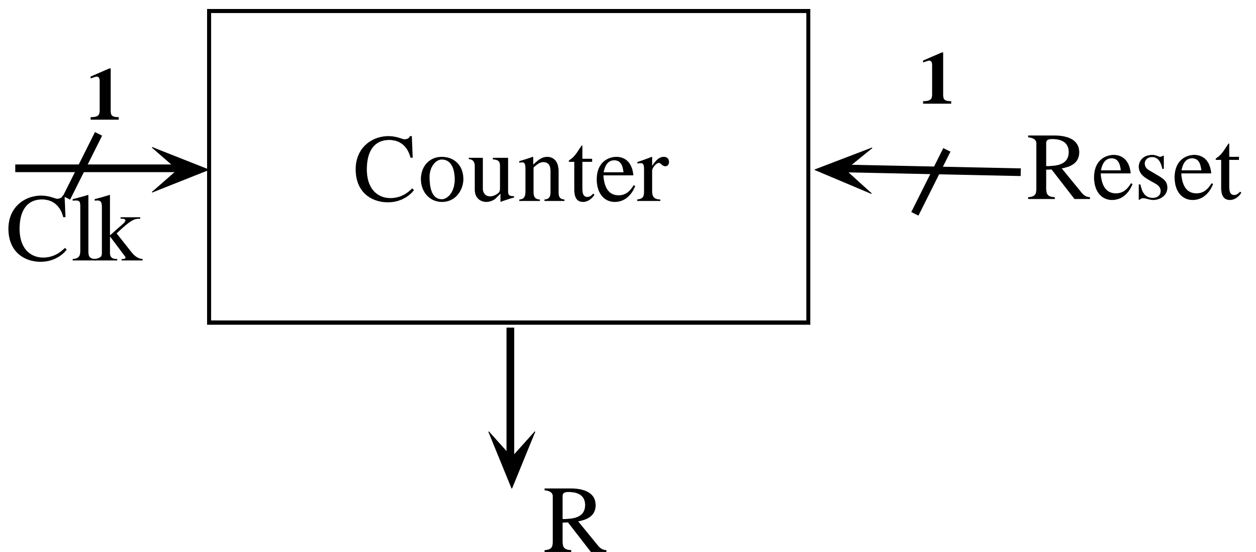
Generic (N: Natural);

Port(Clk: **in** bit;

reset: **in** bit;

R: **out** natural **range** 0 **to** N-1);

End Entity counter;

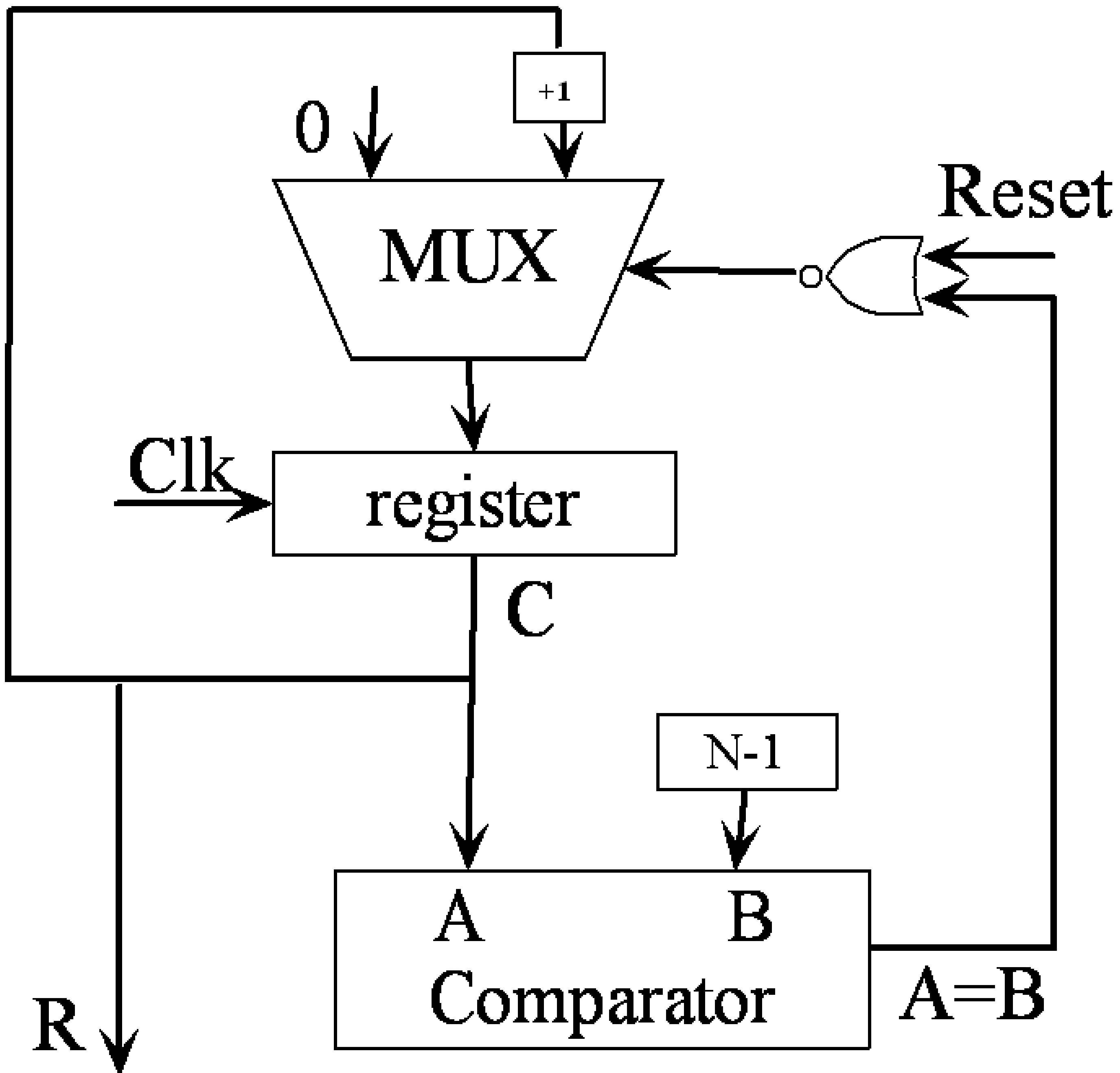


2.1.2 Example 2.2 : An N-bit Counter (cont'd)

Counter with Synch. reset

```
Architecture sync of counter is
   Signal C: Natural range 0 to N-1;
Begin
   R<= C;
   P_count : process (Clk) is
   begin
     If Clk ='1' and Clk'event then
        If reset ='1' or C='N-1' then
           C<=0; -- clear counter
        Else
           C<=C+1;
        End if;
     End if;
   End process P_count;
End architecture Sync
```

2.1.2 Example 2.2 : An N-bit Counter (cont'd) : one error



2.1.3 Lexical Elements

- Comments -- comments are important
- Identifiers are a sequence of non-space characters that obey the following rules
  - Every character is either a letter, a digit, or the underscore (_)
  - The first character in the sequence is a letter
  - The sequence contains no adjacent underscores, and the last character is not an underscore

Remarks: VHDL identifiers are case-insensitive

Some examples:

Last@value 5bit_counter _AO

Clock__pulse good_one

Extended identifiers: \999\

2.1.3 Lexical Elements (cont'd)

- Reserved words: words or keywords are reserved for special use in VHDL. They can't be used as identifiers

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant
disconnect
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
group
guarded
if
impure
in
inertial
inout
is
label
library
linkage
literal
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out
package
port
postponed
procedure
process
pure
range
record
register
reject
rem
report
return
rol
ror
select
severity
signal
shared
sla
sll
sra
srl
subtype
then
to
transport
type
unaffected
units
until
use
variable
wait
when
while
with
xnor
xor

2.1.3 Lexical Elements (cont'd)

- Special symbols
  - $ ' ( ) * +, - . / : ; < => |
  - => ** := /+ >= <= <>
- Numbers: integer literal and real literal
  - Example 10 0 102 4.13
  - Exponential notation 46E3 1E+12 5e0 3.0e-3
  - Base other than 10
    - Base of 2 2#10000000#
    - Base of 8 8#0.4#, what is this in decimal?
  - Underline as separators:
    - 123_456 3.141_592_6 2#1111_1100_0000#
- Characters
  - 'A' --uppercase letter
  - 'z' -- lower case letter
  - ',' -- coma
  - ' ' --the separator character space

2.1.3 Lexical Elements (cont'd)

- Strings: a sequence of characters
  - "a string"
  - "we can include and printing characters"
  - ""
  - "string in string ""a string """. "
  - "if we can't write a string in one line"
  - &" then we break it into two lines"
- bit Strings
  - B (base of 2)   B"0101 0011"
  - b"1111_0010"
  - O (base of 8) O"372"
  - X (base of 16) X"FA" what is this ?
  - X "10" what is this one ?

2.1.4 Syntax Description

- Combine lexical elements to form valid VHDL description
- Syntactic category
- Rules of syntax EBNF (Extended Backus-Naur Form)
  - Example of a variable assignment:
  - Variable_assignment <= target:=expression;
  - D0 := 25+6;
- Optional component [ ]
  - Function_call <= name[(association_list)]
- Combine alternatives |
  - Mode <= in|out|inout
  - Example for process statement

```
Process_statement <=
Process is
        {process_item}
        Begin
        {sequential_statement}
End Process;-- will be talked about later on
```

2.2 Scalar Data Types and Operations

- 2.2.1 Constants, variables, signals
  - Constant_declarations <=
  - **Constant** identifier {,…} : subtype_indication [:= expression];
  - -- constant have a value that is defined once during initialization, and then remains unchanged.
  - -- constants in subprograms are recomputed each time the subprogram is called
  - Examples:
  - **Constant** number _of_bytes : integer :=4;
  - **Constant** e : real := 2.718;
  - **Constant** prop_delay : time := 3 ns;
  - Variable_declarations <=
  - **variable** identifier {,…} : subtype_indication [:= expression];
  - -- have a value that is updated immediately as a result of an assignment statement.
  - Examples:
  - **Variable** index : integer := 0;
  - **Variable** start,finish : time := 0 ns;

2.2.1 Constants, Variables, Signals

- Signals:
- used to model hardware signal conductors
- information is communicated between design components only via signals.
- signals can have fixed links to other signals.
- signals have a present value, as well as a sequence of past and projected future values (variables only have a present value).
- signal values are scheduled to be changed by means of assignment statements:
- A<= new_constant_value;

2.2.1 Variable Versus signals?

Variable and signals are easily confused at first.

- Both signals and variables can be assigned values (also, a signal can be assigned the value of a variable, and vice versa)

Differences between the variables & signals

- Signal correspond to physical signals associated with conductors or busses.
- Variables are a convenience for more easily describing algorithms that might be used in process and subprograms. There is not necessarily any hardware associated with a variable.
- A variable's value can be changed immediately as a result of an assignment statement (which must use the := symbol).
- A signal's value can be changed no sooner than the beginning of the next simulation cycle. The <= symbol must be used.

2.2.2 Scalar types

Type declarations

Type_declaration <= **type** identifiers **is** type_definition;

Example:

Type apples **is range** 0 **to** 100;

Example 2.2 :

Package int_types **is**

    type small_int **is range** 0 **to** 255;

End package int_types;

Use work.int_types.all;

Entity small_adder **is**

  port(a,b: **in** small_int; s: **out** small_int);

End entity small_adder;

2.2.2 Scalar types (cont’d)

Integer Types

Integer _Type_declaration <=

type identifiers **is range** expression (**to** | **downto**) expression;

Example:

Type day_of_month **is range** 0 **to** 31;

Declare variable of this type:

Variable today: day_of_month := 9;

Floating Types …

Arithmetic operations:

- + - * /
- Mod rem abs **

2.2.2 Scalar types (cont'd)

Physical Types

Physical_type_definition <=

type identifiers **is range** expression (**to** | **downto)** expression

Units

Identifier;

{identifier=physical_literal;}

End units [identifier]

Example:

Type resistance **is range** 0 **to** 1E9

Units

Ohm;

kohm = 1000 ohm;

Mohm = 1000 kohm;

End units resistance;

Declare variable of this type:

Variable R1: resistance := 900 ohm;

2.2.2 Scalar types (cont'd)

Enumeration Types:

Type water level **is** (too_low, low, high);

Characters …

Boolean types

Type boolean **is** (false, true);

Bits

Type bit **is** ('0','1');

Standard Logic

Type std_ulogic **is** ('U', --uninitilized
'X', --forcing unkown
'0', --zero
'1', --one
'Z', --high impedance
'W', -- weak unkown
'L', -- weak zero
'H', -- weak one
'-'); --don't care

Sub types

Subtype small_int **is** integer **range** –128 **to** 127

2.2.2 Scalar types (cont'd)

Type qualification

Type Logic_level **is** (unkown, low, undriven, high);

Type system_state **is** (unkown, ready, busy);

To distinguish between common unknown: Use logic_level'(unkown) and
system_state'(unkown)

Type conversion:

real(123) integer(12.4)

Attributes of Scalar types:

- T'left – first(leftmost) value in T
- T'right last(rightmost) value in T
- T'low
- T'high
- T'ascending True if T is an ascending
- T'image(x)
- T'value(s)

2.2.2 Scalar Types (cont’d)

- VHDL is a strongly typed language
- every object has a unique type.
- objects of different types cannot be mixed together in expressions.
- object typing can be determined statically
- the type of every object must be clear from the VHDL program before any simulation has taken place.
- the types of object must be declared explicitly in all program scopes.

2.3 Sequential Statement

- 2.3.1 if statement
- 2.3.2 case statement
- 2.3.3 Null statement
- 2.3.4 loop statement
- 2.3.5 assertion and report statements

2.3.1 If Statement

- Syntax rule
  - If_statement <=
    - [if_label:]
    - **If** boolean_expression **then**
    - {sequential_statement}
    - {**elsif** boolean_expression **then**
    - {sequential_statement}}
    - [**else**
    - {sequential_statement}]
    - **End if** [if_label];

- Example for If_statement <=

If (count =“00”) **then**

a<=b;

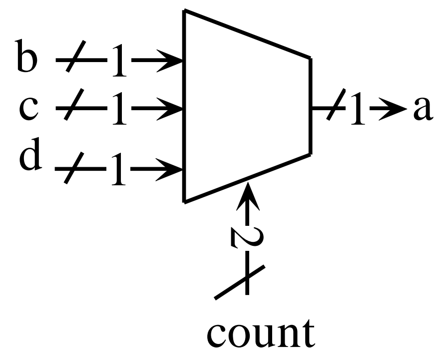
Elsif (count =“10”) **then**

a<=c;

Else

a<=d;

End if ;

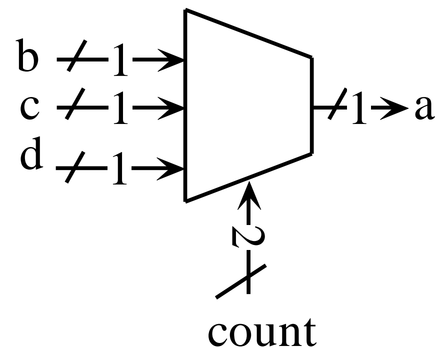


2.3.2 Case Statement

- Syntax rule
  - case_statement <=
    - [case_label:]
    - **case** expression **is**
    - (**when** choices => {sequential_statement})
    - {…}
    - **End case** [case_label];

  - Example for case_statement <=

```
case count is
When "00" =>
        a<=b ;
When "10" =>
        a<=c ;
When others =>
        a<=d ;
End case ;
```



2.3.3 Null Statement

- Null_statement <= [label:] null;
  - Example
  - **case** count **is**

    When “00” =>

    a<=b ;

    When “10” =>

    a<=c ;

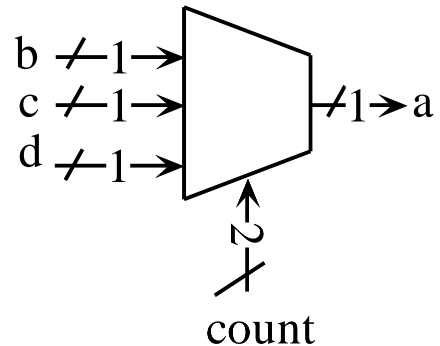
    When “01” =>

    a<=d ;

    When “11” =>

    null ;

    End case ;



2.3.4 Loop Statement

- Infinite loop:
- Loop_statement <=
- [loop_label:]
- **loop**
- {sequential_statement}
- **End loop** [loop_label] ;
- Example: **Loop**
- **wait** until clk ='1';
- count <= count_value;
- **end loop;**
- While loop:
- Loop_statement <=
- [loop_label:]
- **while** condition **loop**
- {sequential_statement}
- **End loop** [loop_label] ;

2.3.4 Loop Statement (cont'd)

Example for While loop:

n := 1;

Sum := 0;

while n <100 **loop**

n := n+1;

Sum := sum +n;

End loop ;

For loop:

Loop_statement <=

[loop_label:]

For identifiers **in** discrete range **loop**

{sequential_statement}

End loop [loop_label] ;

Example for the for loop

For n **in** 1 **to** 100 **loop**

Sum := sum +n;

End loop ;

2.3.4 Loop Statement (cont'd)

Exit statement <=

[label:] exit [loop_label] [when boolean_expression];

Loop

wait until clk ='1' or reset ='1';

Exit when reset = '1';

count <= count_value;

end loop;

NEXT statement

Loop

statement 1;

Next when condition

Statement 2;

End loop;

Loop

statement 1;

If not condition **then**

Statement 2;

End if;

End loop;

2.3.5 Assertion and Report Statement

- Assertion_statement <=
  - [label:] **assert** boolean_expression [**report** expression] [**severity** expression];
  - **assert** initial_value <= max_value
  - **report** “ initial value too large”

2.4 Composite Data Types and Operations

- Array types
  - **Type** BIT **is range** 0 to 1;
  - **Type** word **is array** (31 **downto** 0) of bit;
  - Example:
  - **Signal** MEM_BUS: WORD;-- will be defined later
  - MEM_BUS(0) <= 0 ;
  - MEM_BUS(1) <=0 ;
  - MEM_BUS(2) <=1 ;
- Records
  - **Type** time_stamp **is record**
  - seconds: integer **range 0** to 59;
  - minutes: integer **range 0** to 59;
  - hours : integer **range 0** to 23;
  - **End record** time_stamp;

Variable sample_time, current_time: time_stamp;

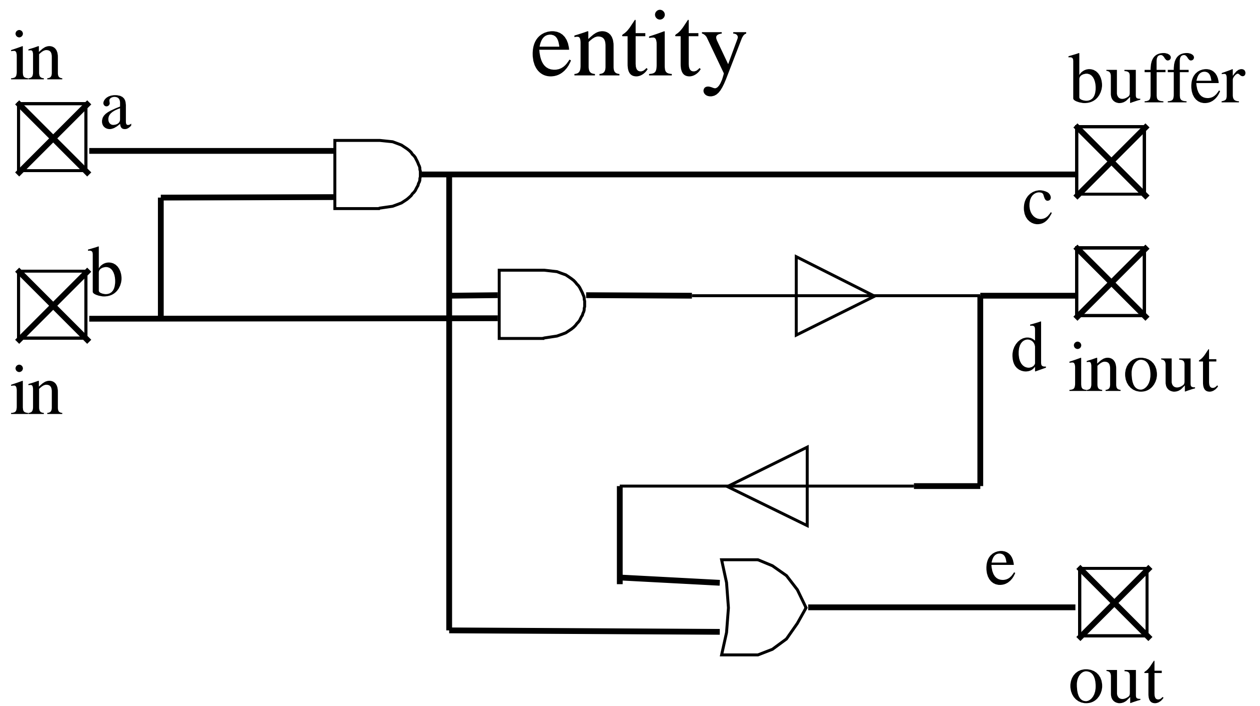
Current_time.seconds := 30;

Current_time.hours := 13;

2.5 Modeling Constructs

- VHDL inherited many modularity ideas form the DoD software language ADA
- hardware specifications are composed of five kinds of design units:
  - **Entities**
  - **Architectures**
  - **Configurations**
  - **Packages**
  - **Package bodies**
- design units are provided to the VHDL simulation and/or synthesis environment in source files
- Design units can also be included from libraries of pre-designed data types, signal types, signal type conversions, components etc.

2.5.1 Modeling Constructs: entity



Entity block **is**

Port (a, b: **in** bit;

c: **buffe**r bit;

d: **inout** bit;

e: **out** bit);

End entity block;

buffer can be used for all output signals

2.5.2 Modeling Constructs: Architecture Bodies

- Architecture_body <=
  - **Architecture** identifier **of** entity_name **is**
    - {block_declaration}
  - **Begin**
  - { concurrent_statement}
  - **End** [architecture][identifier];
  - Example:

```
Entity adder is
  Port (a: in word;
        b: in word;
        sum: out word);
  End entity adder;
```

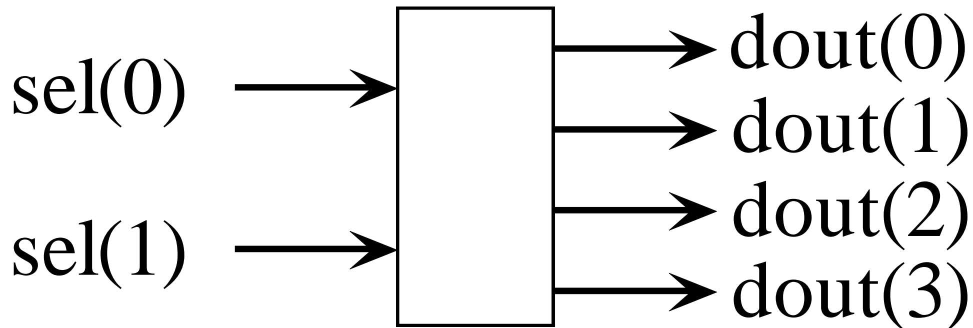
```
Architecture ad1 of adder is
 Begin
 Add_a_b: process(a,b) is
 Begin
     sum <= a+b;
 End process add_a_b;
 End architecture ad1;
```

- Signal declarations
  - Signal_declaration <=
  - Signal identifier {…} : subtype_indication [:= expression]

2.5.3 Two Main Levels of VHDL Specification

- 1) Behavior level:
  - What is the system supposed to do?
  - Components described using algorithms that do not necessarily reflect the actual hardware structure of likely implementations.
  - Signal don't necessary need to be binary values. Data types can be chosen to facilitate high-level description
- 2) Structure level:
  - What is the structure of an implementation?
  - Design specified using realizable components
  - Binary representation of data types and signals are used.

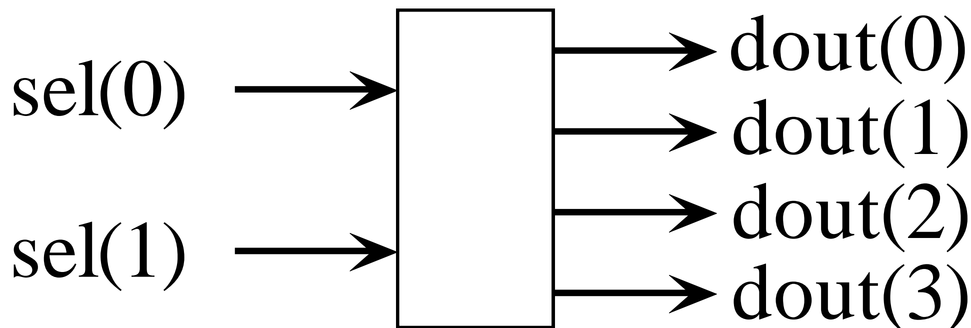
Example 2-to 4 Decoder



- VHDL entity for the decoder

```
Entity decoder is
  port ( sel : in bit_vector (1 downto 0);
         dout : out bit_vector (3 downto 0));
  constant delay : time := 5 ns;
 end entity decoder;
```

Behavior-level architecture in VHDL



- Behavior-level architecture in VHDL

```
Architecture behavior1 of decoder is
 begin
with sel  select
dout <=
   "0001" after delay when "00",
   "0010" after delay when "01",
   "0100" after delay when "10",
   "1000" after delay when "11",
 end behavior1 ;
```

Structure-level architecture

Architecture structure1 of decoder is

component and2 –pre-defined part type

Port (I1, I2 : **in** bit; O1 **out** bit);

End component;

component inverter –pre-defined part type

Port (I1 : **in** bit; O1 **out** bit);

End component;

Signal sel_bar: bit_vector (1 **downto** 0);

Begin

inv_0: inverter **port map** (I1=>sel(0), O1=>sel_bar(0));

inv_1: inverter **port map** (I1=>sel(1), O1=>sel_bar(1));

and_0:and2

port map (I1=>sel_bar(0), I2=>sel_bar(1), O1=>dout(0));

and_1:and2

port map (I1=>sel(0), I2=>sel_bar(1), O1=>dout(1));

and_2:and2

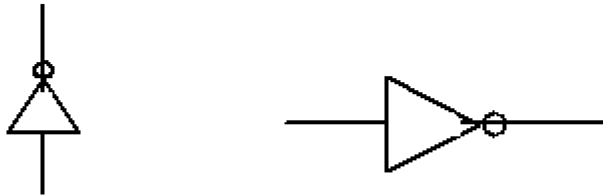
port map (I1=>sel_bar(0), I2=>sel(1), O1=>dout(2));

and_3:and2

port map (I1=>sel(0), I2=>sel(1), O1=>dout(3));

End structure1 ;

Structure-level schematic



sel(0)

sel(1)

dout(0)

dout(1)

dout(2)

dout(3)

2.5.4 Modeling Constructs

- Signal assignment
  - Signal_assignment_statement <=
  - [label:] name <= [delay] waveform;
  - Waveform <= (value_expression [**after** time_expression]) {…}
  - y <= a or b after 5 ns;

- Wait statement <=
  - [label:] **wait** [**on** signal name {…}]
  - [**until** boolean_expression]
  - [**for** time_expression];

2.5.4 Modeling Constructs (cont'd): signal attributes

- **S'delayed(T)**
- if T>0, then a signal is returned that is identical to S delayed by time T. If T=0 (or is absent), then S is returned delayed by time delta.
- **S'stable(T)**
- if T>0, then a signal is returned that has value TRUE if S has not changed for the past time T; at other times the signal has value FALSE. If T=0 (or is absent), then the signal will be FALSE during a simulation cycle when S changes values; otherwise the signal is TRUE.
- **S'quiet(T)**
- if T>0, then a signal is returned that has value TRUE if S has not been updated for the past time T; at other times the signal has value FALSE. If T=0 (or is absent), then the signal will be FALSE during a simulation cycle when S is updated; otherwise the signal is TRUE.

2.5.4 Modeling Constructs (cont'd): signal attributes

- **S'active(T)**
- Boolean that is true if signal S has been updated during the current simulation cycle
- **S'event**
- Boolean that is true if signal S has changed value during the current simulation cycle
- **S'LAST_EVENT**
- The amount of time elapsed since signal S last changed value.
- **S'LAST_ACTIVE**
- The amount of time elapsed since signal S was last updated.
- **S'LAST_VALUE**
- The value of signal S before the last time that signal S changed values.

2.5.4 Modeling Constructs (cont'd)

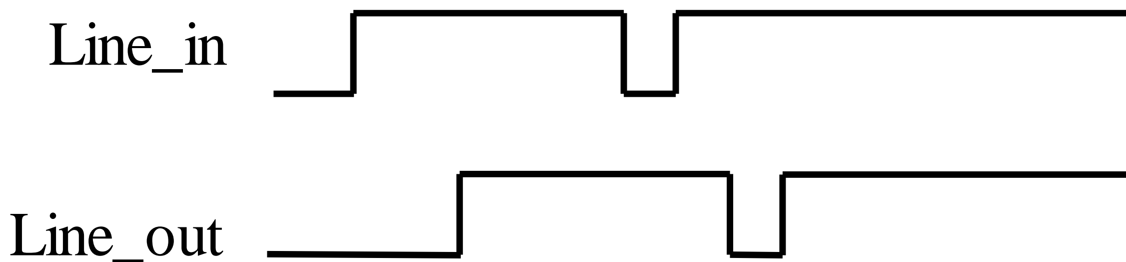
Delay_mechanism<=

transport | [**reject** time_expression]
inertial

Example for transport:

Line_out <= transport line_in after 3 ns;

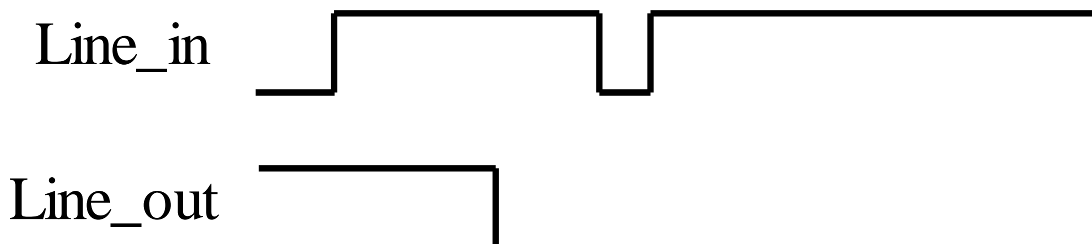
Remarks: the output is shift by the time delay



Example for inertial delay:

Line_out <= inertial not line_in after 3 ns;

- Remarks: if a signal would produce an output pulse shorter than the propagation delay, the the output pulse does not happen



2.5.4 Modeling Constructs (cont'd)

- Example for both inertial and reject
  - Line_out <= reject 2 ns inertial not line_in after 3 ns;
  - Remarks: if a signal would produce an output pulse shorter than the reject limit delay, the the output pulse does not happen
- Process statements <=
  - [process_label:]
  - **Process** [(signal_name{…})] [**is**]
  - { process_item}
  - **Begin**
  - {sequential_statement}
  - **End process** [process_label]

2.5.5 Modeling Concurrency

- In real digital hardware, components all operate at the same time and signals are updated in parallel.
- How to model concurrency/parallel in VHDL
- Components models are decomposed into **processes** that execute in parallel
- Different signals have values that change in parallel over time
- VHDL provides the ability to specify times in the future when signals will be updated.
- VHDL provides the ability to specify synchronization points, when the values of a group of signals are examined and/or updated for the same time instant.

2.5.5 Modeling Concurrency(cont’d)

- -VHDL processes can be used for concurrent statement

- Example :
  - Proc1: **process**(A,B) [is]
  - **Begin**
    - C<= A **or** B **after** 5 ns;
  - **End process;**
- **Another example:**
  - Proc2: **process**(A,B)
  - **Begin**
    - C<= A **or** B;
    - **Wait on** A, B;
  - **End process;**

2.5.5 Modeling Concurrency(cont’d)

- A VHDL process can be thought of as a sub-program that is called once at the beginning of the simulation
- All VHDL processes execute in parallel
- When the simulation starts, each process begins executing statements following the **begin** statement
- Execution is suspended when the next wait statement is encountered
- **Wait;** - suspends process forever
- **Wait on** signal_list;
- **Wait until** condition;
- **Wait for** time_value;
- Once the end process statement is encountered, execution returns to the statement following the begin statement.

2.5.5 Modeling Concurrency(cont'd): Concurrent Statements

- Sequence of Boolean equations:
  - F <= a nor b nor c;
  - D <= a and b and c;
  - E <= a nor b or c;
- When-else conditional signal assignment:

Architecture example **of** fsm **is**

…

With state **select**

X<= "0000" **when** s0|s1

"0010" **when** s2|s3;

Y **when** s4;

Z **when** others;

End example;

2.5.5 Modeling Concurrency(cont'd): Concurrent Statements

- Multiple assignment using Generate:

- g1: **for** j **in** 0 **to** 2 **generate**
  - a(j) <= b(j) or c(j);
  - End **generate** g1;

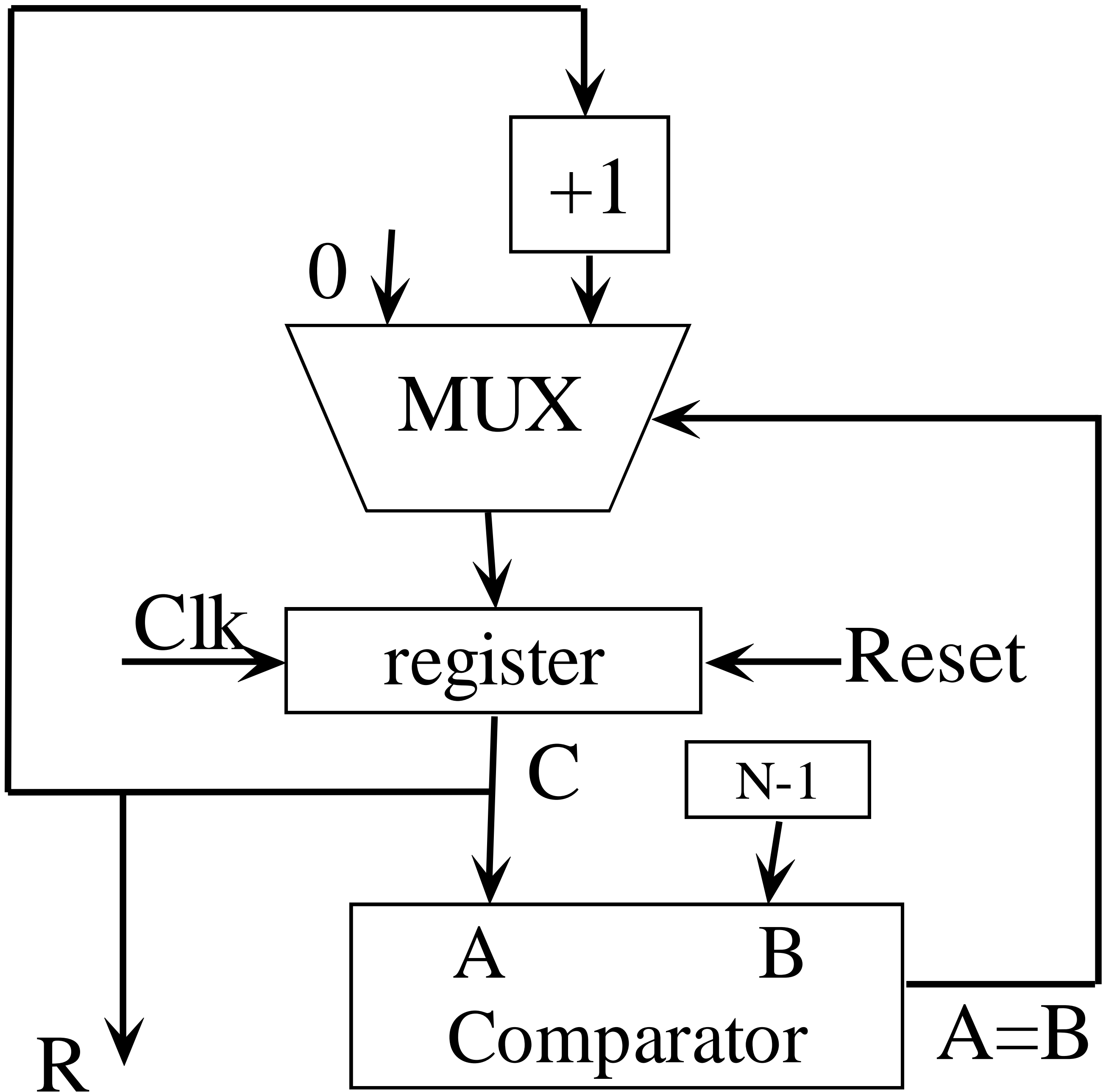
- g2: c(1) <=c(0) **and** a(1);
- For k **in** 2 **to** 20 **generate**
  - c(k) <= c(k-1) **and** a(k);
  - End **generate** g2;

- g3: For l **in** 0 **to** 8 **generate**
  - Reg1: register9 **port map** (clk, reset, enable, d_in(l), d_out(l));
  - End **generate** g3;

2.5.6 Example : Counter with asynch. reset

- **Entity** Counter **is**
  - **Generic** (N: Natural);
- **Port**(Clk: **in** bit;
- reset: **in** bit;
- R: **out** natural **range** 0 **to** N-1);
- **End** counter;
- **Architecture A**sync **of** counter **is**
  - **Signal** C: Natural **range** 0 **to** N-1;
- **Begin**
  - R<= C;
  - P_count : **process** (Clk, reset)
  - **begin**
  - **If** reset ='1' **then**
    - **C <=0; -- clear the counter**
    - **elsIf** clk ='1' and clk`event **then**
      - If C = N-1 then
      - C<=0; -- clear counter
    - **Else**
      - C<=C+1;
    - **End if;**
  - **End if;**
  - **End process** P_count;
- **End** Async

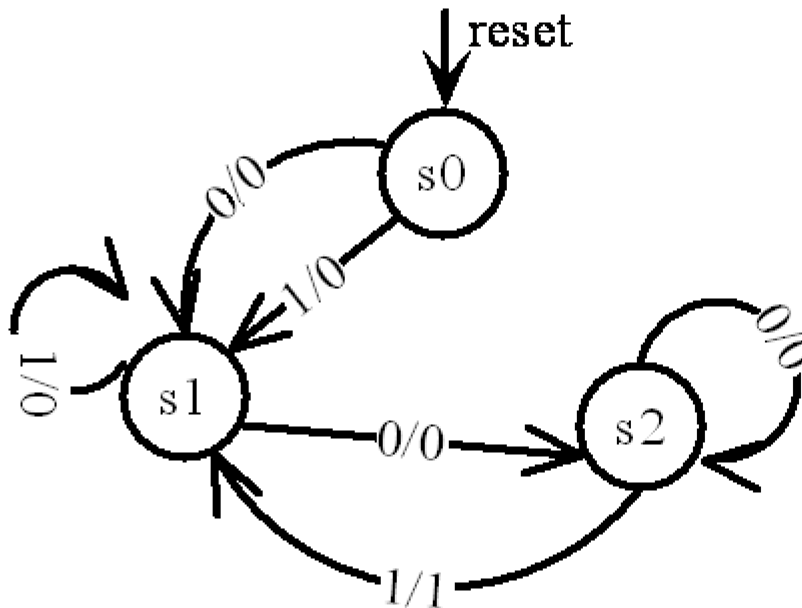
2.5.6 Example counter with asynch. Reset (cont'd)



2.5.6 Modeling Finite State Machine

- VHDL is easy to implement finite state machines
- When combined with logic synthesis, a hardware designer no longer needs to be concerned with the problems of state assignments, logic minimization, etc.
- Instead the designer can concentrate on high level behavior.

2.5.6 Modeling Finite State Machine : Example



Present	Next state		Output	
State	X=0	X=1	X=0	X=1
S0	S1	S1	0	0
S1	S2	S1	0	0
S2	S2	S1	0	1

Architecture of State Machine

- **Architecture** state_machine **of** example **is**
- **Type** stateType **is** (s0,s1,s2);
- **Signal** present_state,next_state:stateType;
- **Begin**
- Comb logic: **process**(present_state,x)
- **Begin**
- **Case** present_state **is**
- **When** s0 => output <=‘0’;
- Next_state <=s1;
- **When** s1 => output <=‘0’;
- **If** (x=‘1’) **then** Next_state <=s1;
- **Else** Next_state <=s2;
- **End if**;
- **When** s2 =>
- **If** (x=‘1’) **then** Next_state <=s1; Output <=‘1’;
- **Else** Next_state <=s2; output <=‘0’;
- **End if**;
- **End case**;
- **End process** comb_logic;

2.6 Subprograms & Packages & use clause (cont'd)

- Procedure encapsulates a collection of sequential statements that are executed for their effect
- Subprogram_body <=
  - **Procedure** identifier [(parameter_list)] **is**
  - **Begin**
  - {sequential_statement}
  - **End** [procedure] [identifier];
- Function encapsulates a collection of statement that compute a result
- Subprogram_body <=
  - [**pure** | **impure**]
  - **Function** identifier [(parameter_list)] **return** type_mark **is**
  - {subprogram_declarative_item}
  - **Begin**
  - {sequential_statement}
  - **End** [function] [identifier];

  - Return_statement <= [label:] return expression;

2.6 Subprograms & Packages & use clause (cont'd)

- Package provide an important way of organizing the data and subprogram declared in a model
- Package_declaration <=
- **package** identifier **is**
- {package_declarative_item}
- **End** [package] [identifier];

- Use clause allows us to make any name form a library or package directly visible
  - Use_clause <= Use selected_name {…};
  - Selected_name <=
  - Name.(identifier|character_literal|operator_symbol|all)

2.6.1 Procedures

- Example:
  - **Procedure** average_sample **is**
  - **Variable total:real := 0.0;**
  - **Begin**
  - **Assert** samples' length >0 **severity** failure;
  - **For** index **in** samples' range **loop**
  - Total :=total+sample(index);
  - **End loop;**
  - Average := total/real(samples' length);
  - **End procedure** average_samples;
- The action of a procedure are invoked by a procedure call statement
  - Procedure_call_statement <= [label:] procedure_name;
- Example:
  - Average_samples;

2.6.1 Procedures (cont'd)

- Return statement in a procedure
  - To handle exceptional conditions, the procedure may return in the middle of the procedure.
  - Return_statement <= [label:] **return**;

Procedure parameters

Interface_list <= ([**constant** | **variable** | **signal**] identifier {…}:[mode] subtype_indication [:=static_expression]) {;…}

mode <= **in** | **out** | **inout**

Example :

Type func_code **is** (add, substract);

Procedure do_arith_op (op: **in** func_code) **is**

  variable result: integer;

Begin

  case op **is**

   when add =>

      result := op1+op2;

   when subtract =>

      result :=op1-op2;

  end case;

End procedure do_arith_op;

2.6.2 Functions

- Example:
  - **Function** limit(value, min, max :integer) **return** integer **is**
  - **Begin**
    - If value > max then
      - **Return** max;
    - **Elsif** value < min then
      - **Return** min;
    - **Else**
      - **Return** value;
    - **End if;**
  - **End function** limit;
  - Pure and impure functions:

  Pure function: same parameter values for same results

  Impure function: same parameter values for possible different results.
- Overloading

2.6.2 Functions (cont'd): Visibility of Declarations

- **Architecture** arch **of** ent **is**
  - **Type** t **is**…;
  - **Signal** s:t;
  - **Procedure** p1(…) **is** - - p1 t s are visible global
    - **Variable** v1:t; -- v1 is visible only in procedure1
  - **Begin**
    - V1:=s;
  - **End Procedure** p1;
- **Begin** – arch
  - Proc1: **process is**
    - **Variable** v2:t; -- v2 is visible in proc1
    - **Procedure** p2(…) **is** --p2 is visible in proc1
      - **Variable** v3:t; --v3 is only visible in procedure2
    - **Begin**
      - P1(v2, v3…);
    - **End procedure** p2;
  - **Begin** –proc1
    - P2(V2,…);
  - **End process** proc1;
  - Proc2: **process is**
  - …
  - **Begin** –proc2
    - P1(…);
  - **End process** proc2;
- **End architecture** arch;

2.6.3 Packages

- Example :
  - **Package** cpu_type **is**
    - **Constant** word_size:positive := 16;
    - **Constant** address_size :positive :=24;
    - **Subtype** address **is** bit_vector(address_size-1 **downto** 0);
  - **End package** cpu_type;
- The cpu_type package has been analyzed and placed into the work library.
  - **Entity** address_decoder **is**
    - **Port** (addr : **in** work.cpu_types.address;
      - ……..);
    - **End entity** address_decoder;
- Remarks:
- Each package declaration that includes subprogram declarations or deferred constant declarations must have corresponding package body to fill in the missing details. However, if a package only include other kinds of declarations, such as types, signals, constant. No package body is necessary.

2.6.3 Packages (cont'd) : Package bodies

- Example :
  - **Package** some_arithmetic **is**
    - **Function** limit(value, min, max :integer) **return** integer;
    - **constant** word_size:positive := 16;
    - **Constant** address_size :positive :=24;
    - ……..
  - **End package** some_arithmetic;

  - **Package body** some_arithmetic **is**
    - **Function** limit(value, min, max :integer) **return** integer **is**
    - **Begin**
      - If value > max then
        - **Return** max;
      - **Elsif** value < min then
        - **Return** min;
      - **Else**
      - **Return** value;
      - **End if;**
    - **End function** limit;
    - …..
  - **End package body** some_arithmetic;

2.6.3 Use clause

- **Variable** Next_address: work.cpu_types.address;
- ….. -- tedious to
- Changes to
- **Use** work.cpu_types;
- **Variable** Next_address: cpu_types.address;
- …..
- Example:
  - **Library** ieee; **use** ieee.std_logic_1164.std_logic;
  - **Entity** logic_block is
    - **Port** (a, b: in std_logic;
      - Y,z: out std_logic);
  - **End entity** logic_Block;

2.7 Resolved Signals & Generic Constants

- Problem: Multiple output ports connecting one signal.
- **Type** tri_state_logic **is** (‘0’, ‘1’, ‘z’);
- **Type** tri_state_logic_array **is** array (integer range<>) of tri_state_logic;
  - **Function** resolve_tri_state_logic(value : **in** tri_state_logic_array) **return** tri_state_logic **is**
    - **Variable** result : tri_state_logic :=‘Z’;
  - **Begin**
    - **For** index **in** values' range **loop**
      - **If values(index) /= ‘z’ then**
        - **Result :=values(index);**
      - **End if;**
    - **End loop;**
    - **Return** result;
  - **End function** resolve_tri_state_logic;
- **Signal** s1: resolve_tri_state_logic tri_state_logic**;**
- **Subtype** resolved_logic **is** resolve_tri_state_logic tri_state_logic;
- **Signal** S2,S3: resolved_logic;

2.7.1 Resolved Signals (cont'd)

- IEEE std_logic_1164 resolved subtypes
- **Type** std_ulogic **is** ('U','X','0','1','Z','W','L','H','-');
- **Type** std_ulogic_vector **is** array (natural range<>) of std_ulogic;
- **Function** resolved(s:std_ulogic_vector) **return** std_ulogic;
- **Subtype** std_logic **is** resolved std_ulogic;
- **Type** std_logic_vector **is** array (natural range <>) of std_logic;

2.7.2 Generic Constants

- Generic: writing parameterized models
- Entity_declaration <=
  - **Entity** identifier **is**
    - [**generic** (generic_interface_list);]
    - [**port** (port_interface_list);]
    - {entity_declarative_item};
  - [**begin**
    - Concurrent_assertion_statement | passive_concurrent_procedure_call_statement | passiv_process_statement}]
  - **End** [**entity**] [identifier];

- A simple example
  - **Entity** and2 **is**
    - **Generic** (Tpd : time);
    - Port (a,b : **in** bit; y :**out** bit);
  - **End entity** and2;

  - **Architecture** simple of and2 **is**
  - **Begin**
    - And2_function:
    - Y<= a **and** b **after** Tpd;
  - **End architecture** simple;

2.7.2 Generic Constants (cont'd)

- A generic constant is given an actual value when the entity is used in a component instantiation statement.
- Component_instantiation_statement <=
  - Instantiation_label:
  - **Entity** entity_name [(architecture_identifier)]
  - [**generic map** (generic_association_list)]
  - [**port map**(port_association_list)];

  - Example to use and2 for component instantiation:
  - Gate1: **entity** work.and2(simple)
    - **Generic map**(Tpd => 2 ns)
    - **Port map** (a=>sig1,b=>sig2,y=>sig_out);
- For number of generic constants:
  - **Entity** control_unit **is**
    - **Generic** (Tpd_clk_out, tpw_clk : delay_length; debug: boolean:=false);
    - **Port** (clk : in bit; ready : in bit; control : out bit);
  - **End entity** control_unit;
  - Three ways to write a generic map:
  - **Generic map**(200ps, 1500 ps, false)
  - **Generic map**(tpd_clk_out=>200ps, tpw_clk=> 1500 ps)
  - **Generic map**(200ps, 1500 ps, debug => **open**) - - open means using the default value

2.7.2 Generic Constants (cont'd)

- Second use of generic constants is to parameterize their structure.
  - **Entity** reg **is**
  - **Generic** (width : positive);
  - **Port**(d: in bit_vector(0 **to** width –1);
  -         q: out bit_vector(0 **to** width –1);
  -         …);
  - **End entity** reg;

  - **Signal** in_data, out_data:bit_vector(0 **to** bus_size-1);
  - …
  - Ok_reg:**entity** work.reg
  - **Generic map**(width=>bus_size)
  - **Port map**(d=>in_data, q=> out_data,…);

2.8 Components and Configurations

- Component_declaration <=
- **Component** identifier [**is**]
- [**generic** (generic_interface_list);]
- [**port**(port_interface_list);]
- **End component** [identifeir];
- Example:
- **component** and2 –pre-defined part type
  -
  - **Port** (I1, I2 : **in** bit; O1 **out** bit);
- **End component**;
- Component_instantiation_statement <=
- Instantiation_label:
- [**component**] component_name
- [**generic map** (generic_association_list)]
- [**port map**(port_association_list)];

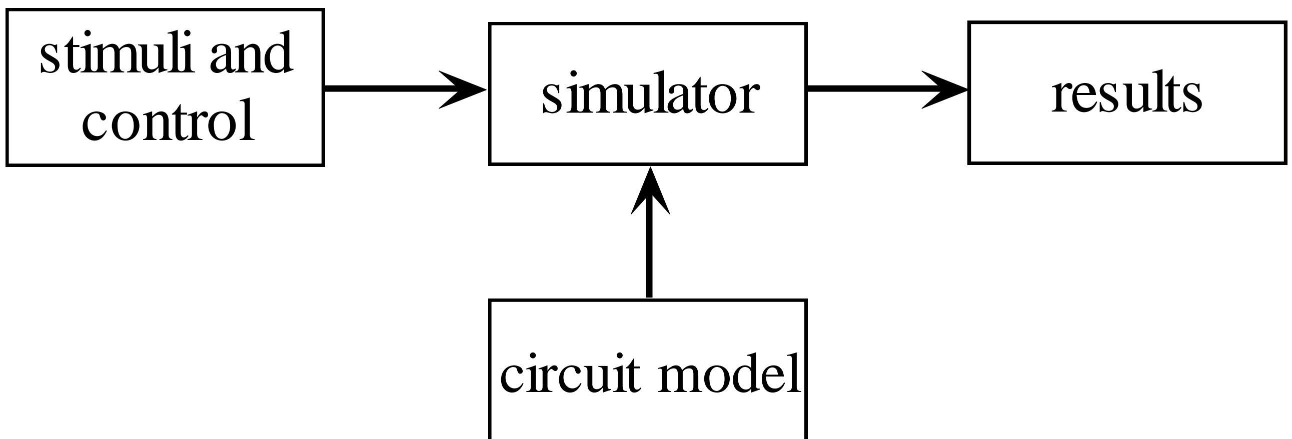
- Example see Architecture structure1 of decoder in slide 105.

2.8 Components and Configurations (cont’d)

- Packaging components:
- **Library** ieee; **use** ieee.std_logic_1164.all;
- **Package** serial_interface_defs **is**
- Subtype …
- Constant …
- **Component** serial_interface is
  - **Port**(…);
- **End component** serial_interface;
- **End package** serial_interface_defs;
- Entity declaration:
- **Library** ieee; **use** ieee.std_logic_1164.all;
- **Use** work.serial_interface_defs.**all**;
- **Entity** serial_interface is
  - **Port**(…);
- **End entity** serial_interface;
- An architecture body:
- **Library** ieee; **use** ieee.std_logic_1164.all;
- **Architecture** structure1 of micro controller is
  - **Use** work.serial_interface_defs.serial_interface;
- **Begin**
  - serial_a : **component** serial_interface
  - **Port** map(…);
- …

2.9: Logic Simulation

- A tool for design verification testing
- Ascertain a design perform its specified behavior (function and timing)
- Simulation process

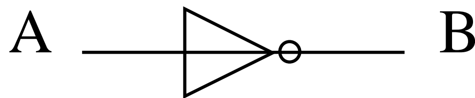


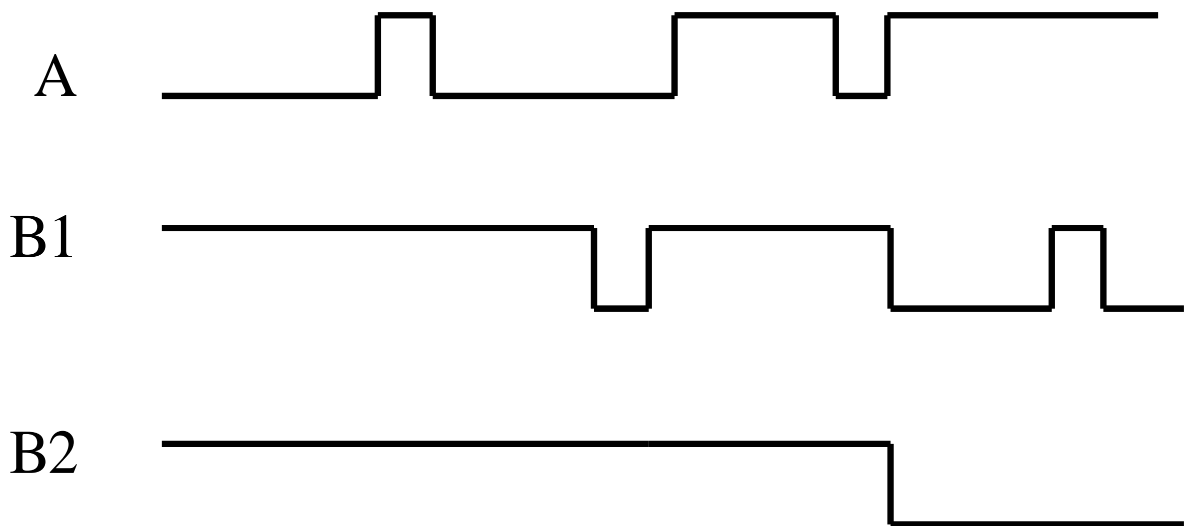
2.9.1 Design Verification: True Value Simulation

- Determine the output for certain inputs
- Find design errors, timing problems, etc.
- Limitations:
  - The set of design errors is not enumerable
  - No formal procedure to generate tests
  - A system passes the test is shown correct only with the applied test cases
  - Simplified circuit model
  - A circuit passes the simulation test may not work when being wired up

2.9.1 Design Verification: True Value Simulation (cont'd)

- Timing :delay models
  - Transport delays
  - Initial delays



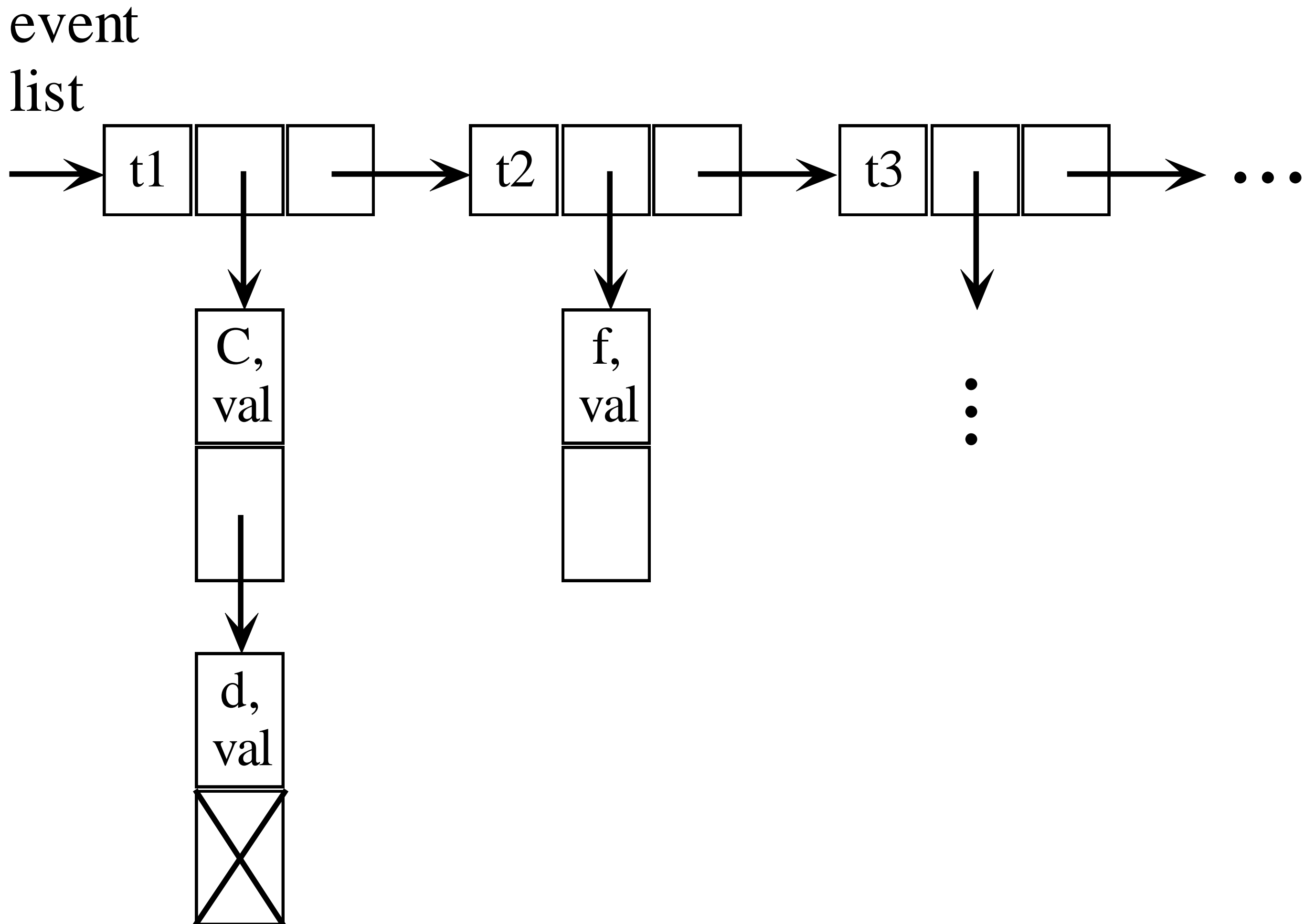


2.9.2 Event_driven Simulation

- General concept behind most simulation algorithms
- An event represents a change in the value of a signal line at some simulated time t
- If the value of a signal line x changes, then all gates having x as input are activated
- If the activated gates change their output values, new events are generated.

2.9.2 Event_driven Simulation (cont'd)

- An event list can be organized as a linked list stored in increasing time order



2.9.2 Event_driven Simulation (cont’d)

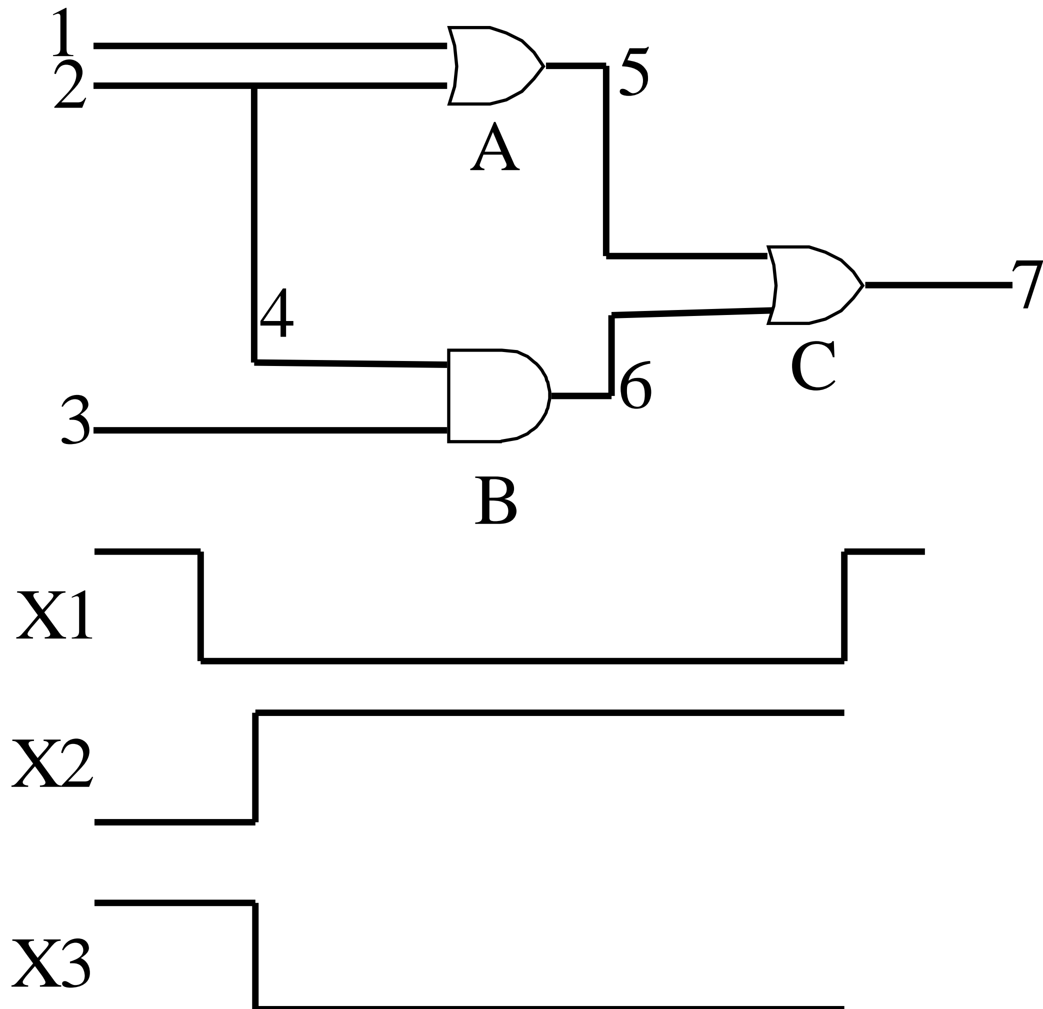
- Two-pass algorithm for gate-level event-driven simulation
- **While**(event_list is not empty) do begin
    - **For** (every event(x,val) at time t) do begin
        - Get the event (x,val)
        - **If**(current_val_of_x <>val) then begin
            - Current_val_of_x=val
            - **For** (each gate J on fanout list of x) do begin
                - Change the value of fanout line going into gate J
                - Add gate J to activated_gates
            - **End**
        - **End**
    - **End**
    - **For** (each gate J of activated_gates) do begin
        - Compute output value of gate J
        - **If**(output_value_gate_j <> last_scheduled_value_gate_j) then begin
        - Schedule (output_line_number_gate_J,output_value_gate_J) for time t+delay of gate_J
- Last scheduled_value_gate_J=output_value_gate_J
        - **End**
    - **End**
- **End**

2.9.2 Event_driven Simulation (cont'd)

- Pass1
- Get the entries from the event list associated with the current time t
- Determine the activated gates
- This is to avoid multiple evaluations of a gate that is activated by more than one event

- Pass2
- Compute the new output values of the activated gates
- Schedule their computed value in the event list
- The algorithm keeps track of the last-scheduled value of a gate so as to schedule only "true" events

2.9.2 Event_driven Simulation (cont'd): an Example

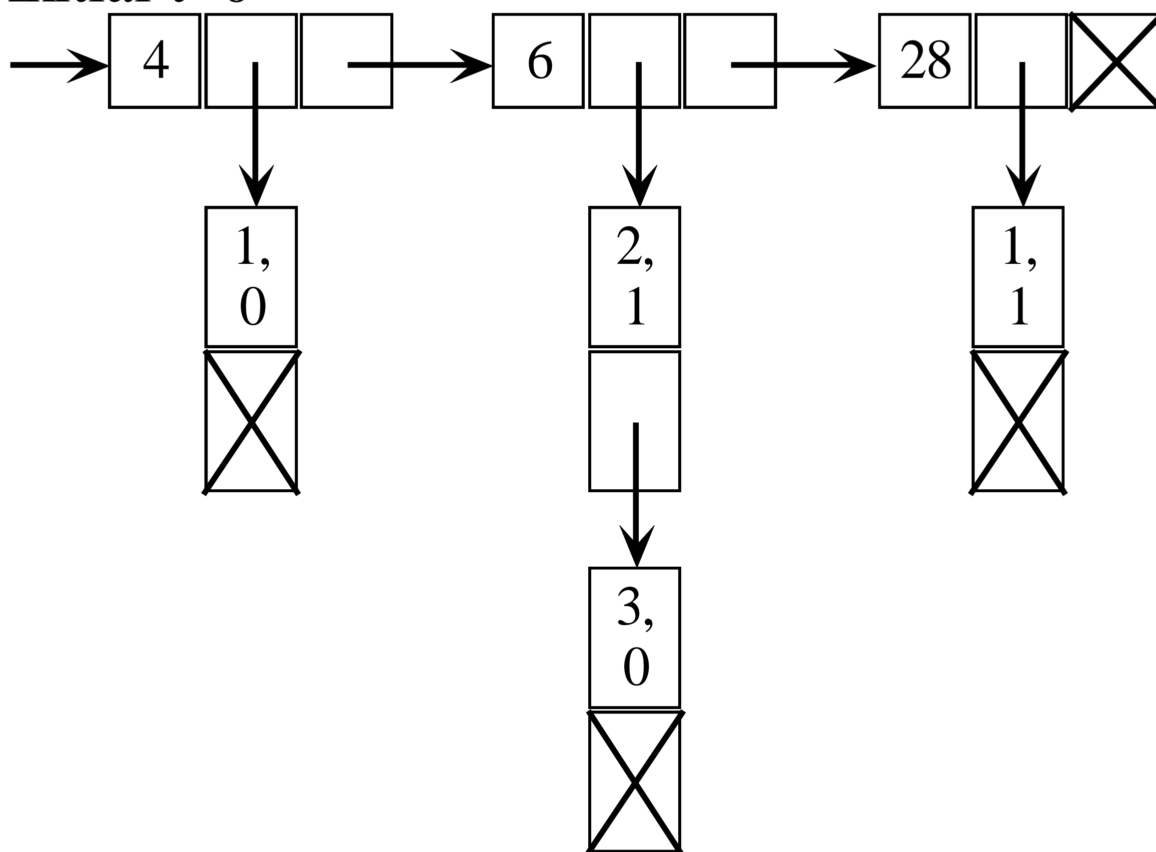


Gate	A	B	C
Propagation delay	10 ns	6 ns	10 ns

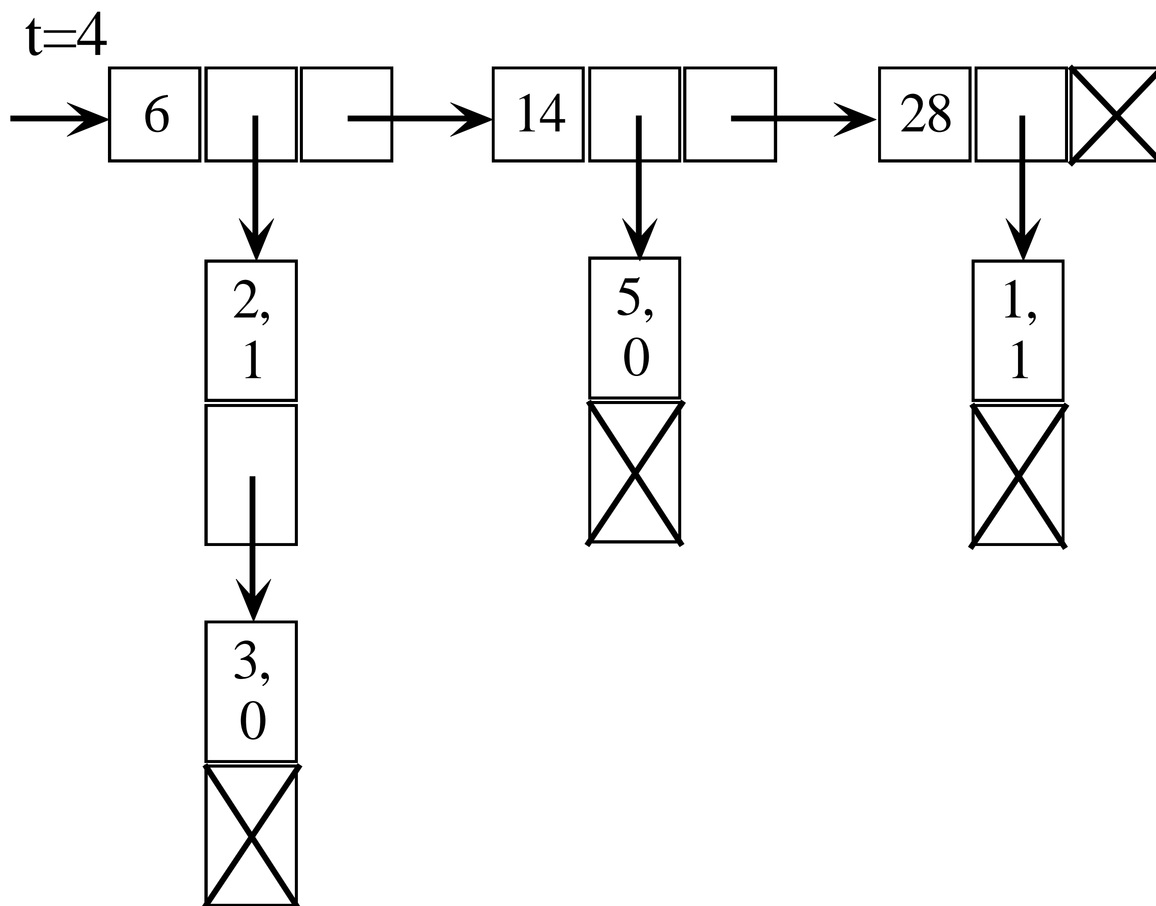
2.9.2 Event_driven Simulation (cont'd): an Example

	Phase 1							phase2			
	Line values							Gates affected			Scheduled events
Time	1	2	3	4	5	6	7	A	B	C	
Init	1	0	1	0	1	0	1				
4	0							X			(5,0) at 4+10
6		1	0	1				X	X		(5,1) at 6+10
14					0					X	(7,0) at 14+10
16					1					X	(7,1) at 16+10
24							0				
26							1				
28	1							X			

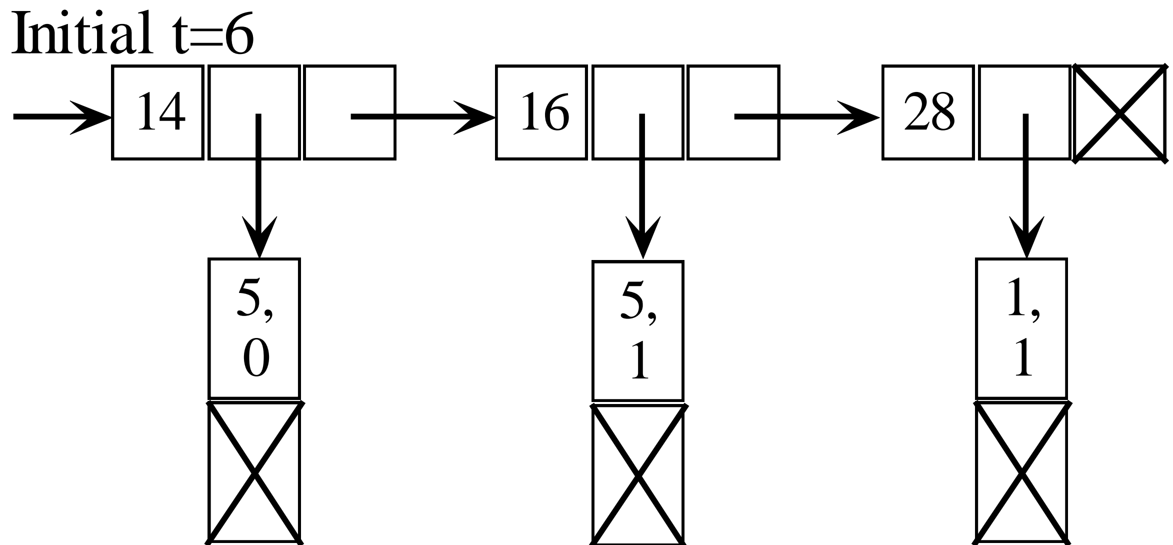
Event_driven Simulation (cont'd): an Example



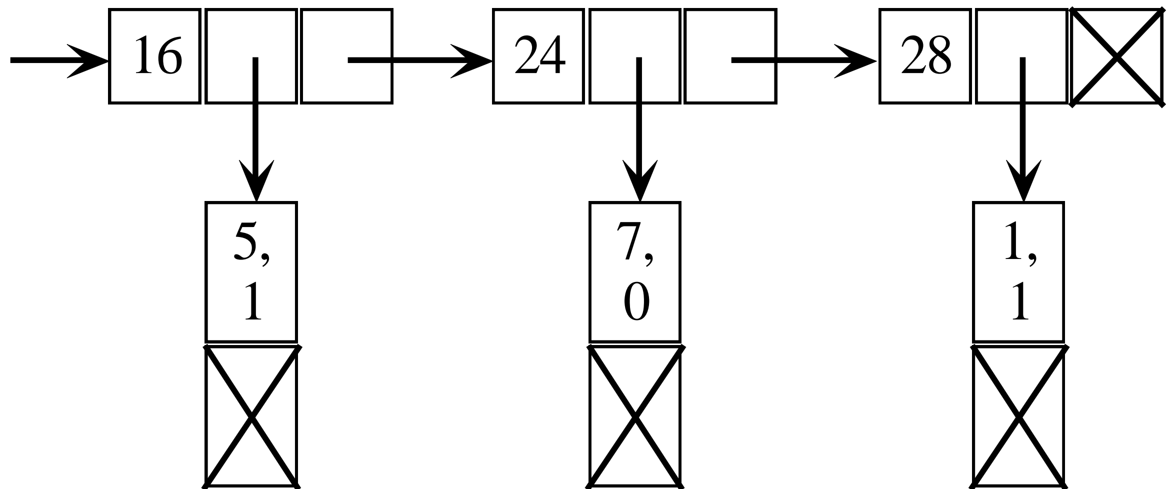
Event_driven Simulation (cont'd): an Example



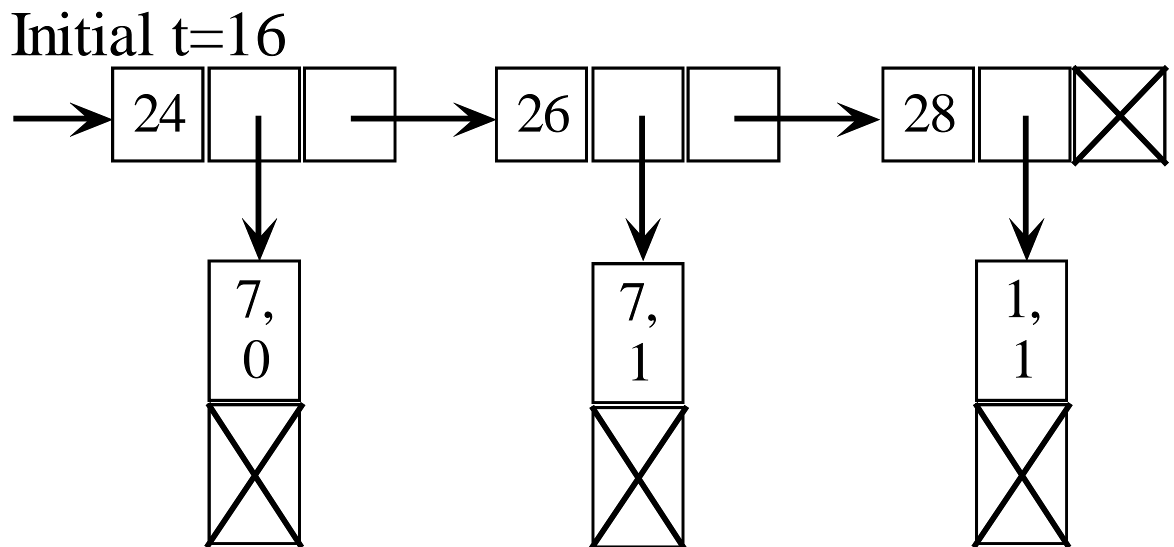
Event_driven Simulation (cont'd): an Example



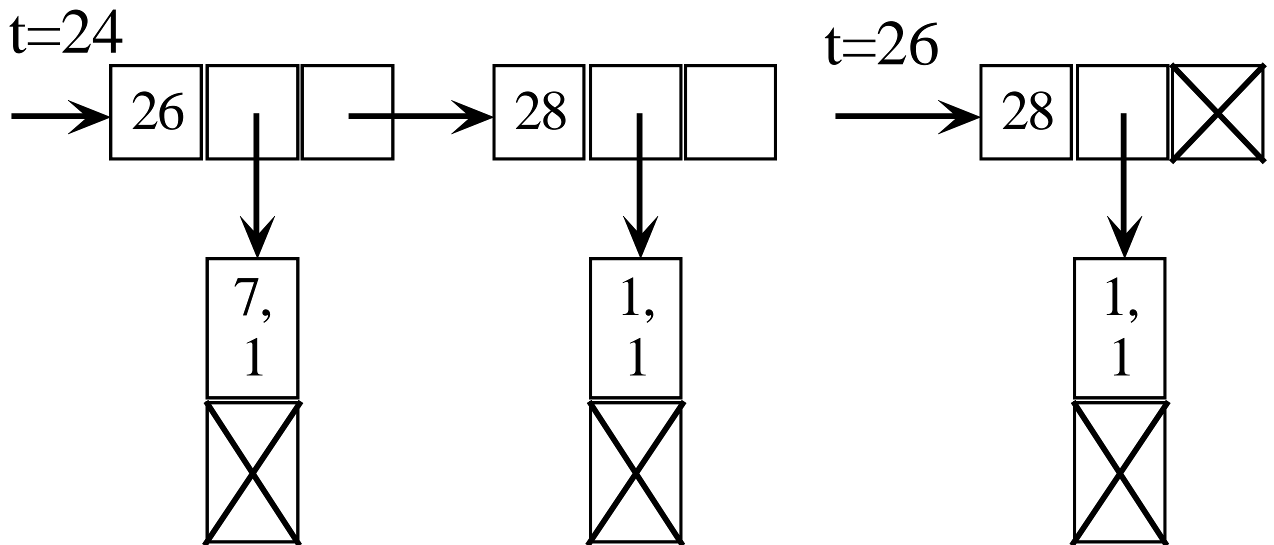
Event_driven Simulation (cont'd): an Example



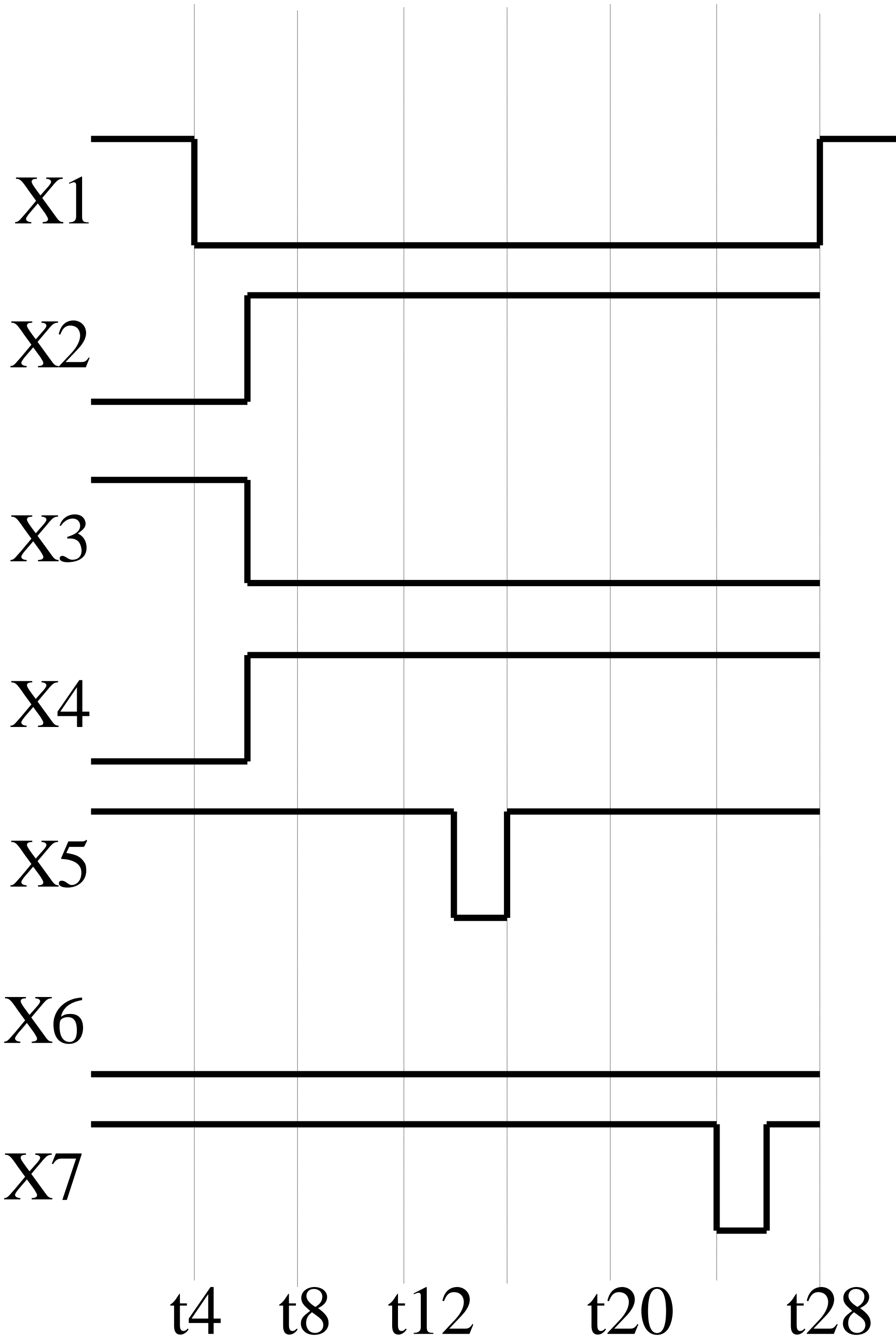
Event_driven Simulation (cont'd): an Example



Event_driven Simulation (cont'd): an Example



Event_driven Simulation (cont'd): Timing Diagram



2.9.2 Synthesis and Simulation

- **Simulation**
- model testing
- model debugging
- Find design errors,
- Find timing problems,

- **Synthesis**
- Reduction of a design description to a lower-level circuit representation.
- shorter design cycle
- Lower design cost
- Fewer design errors.
- Easier to determine available design trade-offs.

2.9.2 Predefined Environment

- The package STANDARD is always available
- **Package** STANDARD **is**
  - **Type** Boolean **is** (FALSE, TRUE);
  - **Type** BIT **is** (‘0’,’1’);
  - **Type** character is (ASCII characters);
  - **Type** severity_level **is** (note, warning, error, failure);
  - **Type** time **is** range implementation_defined
  - **Units** fs; ps=1000 fs; ns=1000 ps; us=1000ns;ms=1000us;sec=1000ms;min=60sec;hr=60 min;
  - **End units**
- Predefined numeric types
  - **Type** integer **is range** implementation_defined;
  - **Type** real **is range** implementation_defined;

2.9.2 Predefined Environment (cont'd): standard package

- **Function** Now **return** Time – function that returns current simulation time
- **Subtype** Natural **is** integer **range** 0 **to** integer' high;--numeric subtypes
- **Subtype** positive **is** integer **range** 1 **to** integer' high;
- **Type** string **is array** (**positive range<>)** **of** character;
- **Type** bit_vector **is array**(natural **range <>) of** bit;

- End STANDARD;

2.9.2 Predefined Environment (cont'd)

- Package TEXTIO is also always available
- **Package** TEXTIO **is**
  - **Type** Line **is** access string;
  - **Type** text **is** file of string;
  - **Type** side **is** (right,left);
  - **Subtype** width **is** natural;
  - **File** Input :text **is** in "STD_INPUT";
  - **File** output : text **is** out "STD_OUTPUT);
  - **Procedure** readline (F: **in** TEXT; L : **out** Line);
  - **Procedure** read (L: **inout** line; V : **out** Bit);
  - **Procedure** read (L: **inout** line; V : **out** Bit_vector);
  - **Procedure** read (L: **inout** line; V : **out** Boolean);

2.9.2 Predefined Environment(cont'd): TEXTIO package

- **Procedure** read (L: **inout** line; V : **out** character);
- **Procedure** read (L: **inout** line; V : **out** integer);
- **Procedure** read (L: **inout** line; V : **out** real);
- **Procedure** read (L: **inout** line; V : **out** string);
- **Procedure** read (L: **inout** line; V : **out** time);
- **Procedure** writeline (F: **out** text; L :**in** line);
- **Procedure** write (L:**inout** line; V : **in** bit; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** bit_vector; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** boolean; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** character; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** integer; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** real; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** string; justified : **in** side := right; field : **in** width :=0);
- **Procedure** write (L:**inout** line; V : **in** time; justified : **in** side := right; field : **in** width :=0);

- End textio;

2.9.2 Predefined Environment: Standard IEEE Library

- Package STD_logic _1164 is not part of the VHDL standard, but it is so widely used. To access the package, a VHDL program must include the following two lines at the beginning:
- Library ieee;
- Use ieee.std_logic_1164.all;
- Signals in this library have nine values
- **Type** std_logic **is** (
  - ‘U’, --uninitialized
  - ‘X’,--forcing unknown
  - ‘0’—forcing 0
  - ‘1’,--forcing 1
  - ‘z’,--high impedance
  - ‘w’,--weak unknown
  - ‘l’,--weak 0
  - ‘h’,--weak 1
  - ‘-’); -- don’t care

2.9.2 Predefined Environment: Standard IEEE Library (cont'd)

- The type STD_logic is provided with a resolution function that determines the final obtained when two or more buffers drive different values onto a signal
- The type STD_ULOGIC has the same nine signal values as STD_LOGIC, but without the resolution function.

- **Type** std_logic_vector **is**
- **Array** (**natural** range <>) **of** STD_logic;
- **Type** std_ulogic_vector **is**
- **Array** (**natural** range <>) **of** STD_ulogic;

2.9.2 Predefined Environment: Standard IEEE Library (cont'd)

- The standard IEEE library (cont'd)
  - **Function** To_bit (S: std_ulogic; Xmap : Bit := '0') **return** Bit;
  - **Function** To_bitvector (S: std_logic_vector; Xmap : Bit := '0') **return** Bit_vector;
  - **Function** To_bitvector (S: std_ulogic_vector; Xmap : Bit := '0') **return** Bit_vector;
  - **Function** To_stdulogic (B: bit) **return** std_ulogic;
  - **Function** To_stdlogicvector (B: bit_vector) **return** std_logic_vector;
  - **Function** To_stdulogic (B: std_ulogic_vector) **return** std_logic_vector;
  - **Function** To_stdulogic (B: bit_vector) **return** std_ulogic_vector;

III. Digital Design & Applications

3. Digital Design and Applications

- 3.1 Introduction to Digital System Design
- 3.2 Register-Transfer Level
- 3.3 Impediments to Synchronous Design
- 3.4 Variable Entered Maps
- 3.5 Design steps for a digital system
- 3.6 Digital Design Example

3.1.1 Problems for classical Sequential Design

- Classical sequential circuit design techniques could, in theory, be used in arbitrarily complex design problems.
- In practice, however, classical techniques are ineffective for all but the simplest problems.
- Reason:
  - The complexity of the design problem overwhelms the human designer's ability to find a correct solution
  - Using classical techniques leads to designs that are
    - Hard to understand, hard to modify, and hard to test

3.1.2 Software solutions to Digital System Design

•Many digital design problems are entirely solvable using custom software and standard microprocessor: examples ?

- **Advantages:**
- No need to design hardware
- Software is relatively easy to change/customize
- Complex features can be readily provided in software
- Can test software designs using emulators.

- Disadvantages
- Microprocessors can be overly complicated for many controller problems
- Custom hardware can provide better performance than general-purpose hardware
- Custom techniques are still required in custom or semi custom integrated circuits
- Custom designs can be protected using patent or IC mask laws

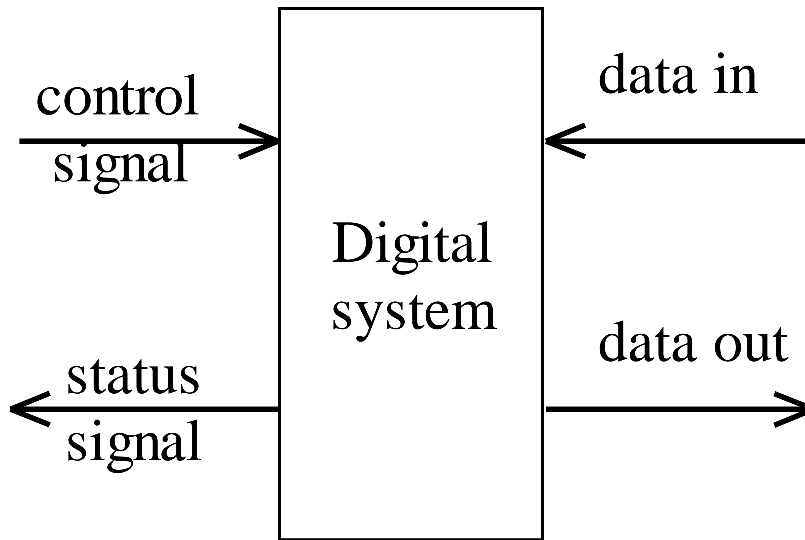
3.1.3 Basic Strategies in Digital System Design

- Top-down design
  - Manage design complexity.
  - Postpone commitment to any particular hardware
  - Use iterative refinement to gradually converge on a natural solution
  - Decompose a module into loosely interacting sub modules
  - Design using high-level building blocks
- Conservative/safe design techniques
  - Use synchronous hardware wherever possible
  - Convert asynchronous inputs to synchronous inputs
  - Use a robust system-wide clocking strategy
  - Make design static if possible(ie. Correct operation is independent of clock speed)

3.1.3 Basic Strategies in Digital System Design (cont'd)

- Conservative/safe design techniques
  - Provide a single stepping mode
  - Make design testable by construction
  - Avoid obscure design tricks
- Document the design thoroughly
  - Requirement, (user-oriented), specification (designer-oriented)
  - Reasons for designs decisions
  - List relevant standards
  - Propose test plans
  - Develop maintenance procedures
  - Consider manufacturability issues

3.1.4 Signals at External Interface



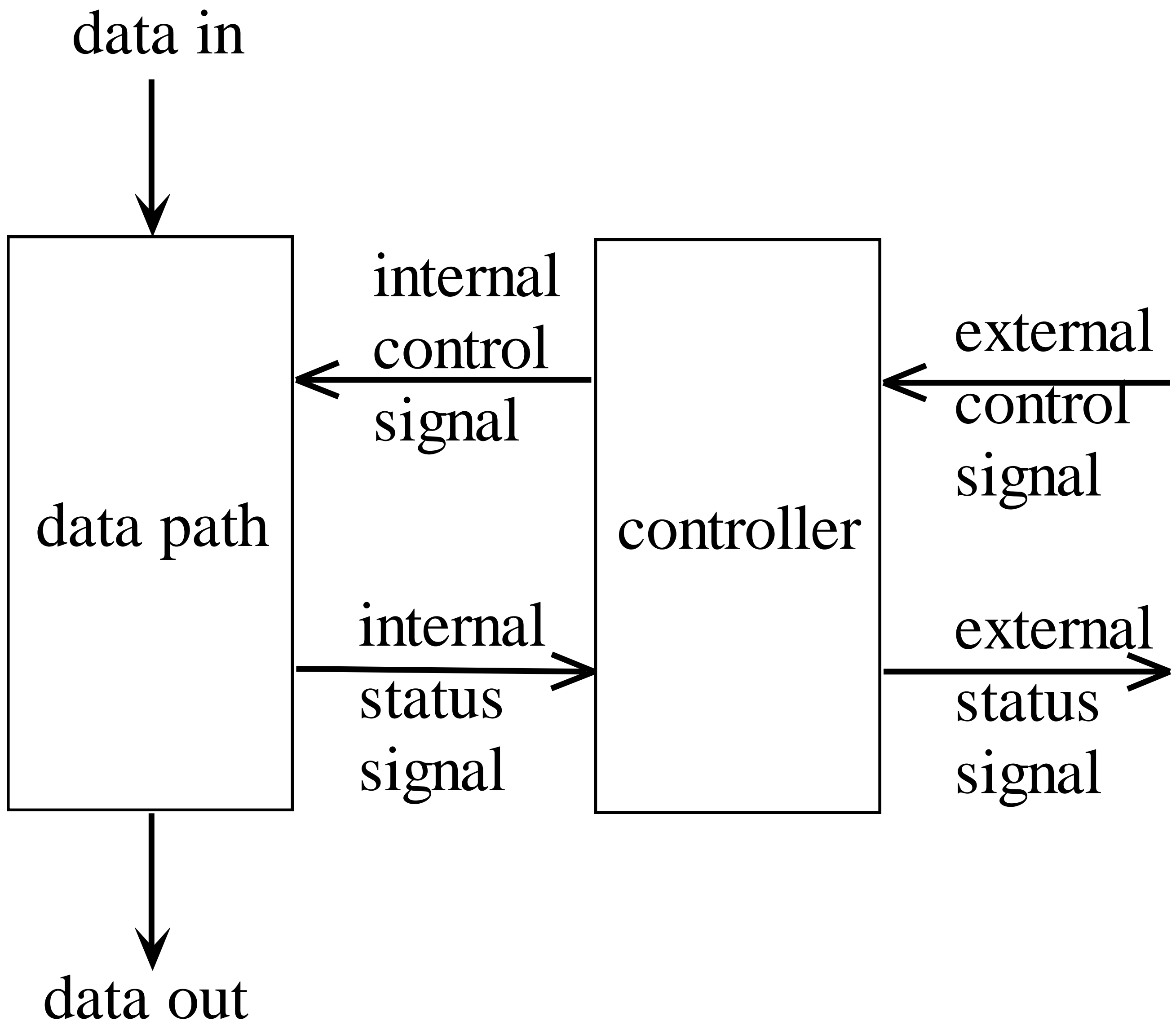
- System level control signals:
  - Highest level commands
  - External user is shielded form internal details
- System level status signals
  - Simplified high-level status information is provided to the external user
- Data in
  - Analog signals are typical converted to digital processing
- Data out
  - Analog signals may need to be reconstructed from their internal digital representation

3.1.5 Highest Level of System Architecture

- Control path
- Circuit that control algorithm by which operators are applied to the data
- State control and data operation sequencing are emphasized
- Typical control units:
  - Registers
  - Next state logic
  - Output logic
  - Control+status line

- Data path
- Circuits that directly store and transform the data
- Bit parallelism and regular structure are emphasized
- Typical data path elements:
- Registers
- Multiplexes
- Shift/adders/ALUs
- Counters
- buses

3.1.5 Highest Level of System Architecture (cont'd)



3.2 Register Transfer Level

- A convenient conceptual level intermediate between the system level and the gate level
  - RTL assumes a set of hardware constructs are defined in FPGA hardware and library elements
  - HDL code is mapped to these constructs
  - Describe the operation of synchronous system
  - Combine the control-flow state machine with means for defining and operating multi-bit registers
- Typical RTL constructs:
- Combinational logic
  - Arbitrary functions (random logic, ROMs, PALs, PLAs)
  - Multiplexers
  - Demultiplexers/decoders
  - Comparators
  - Arithmetic./logic circuits (ALUs, adders, subtractors)

3.2 Register Transfer Level (cont'd)

- Sequential Logic
  - Latches, flip-flops
  - Registers, shift registers
  - Counters, LFSRs
  - RAMs

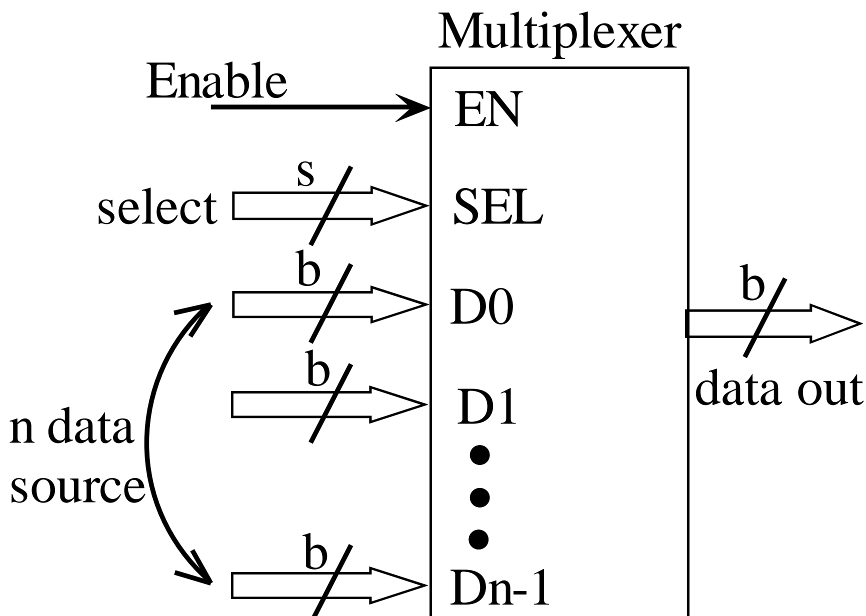
- Interconnect:
  - Buses
  - Wires
  - Buffers
  - Tri-state able buffers
  - Bi-directional transceivers

3.2.1 Multiplexers (Mux)

- Multiplexer is a digital switch
  - It connects data from one of n sources to its output.
  - A logic equation:

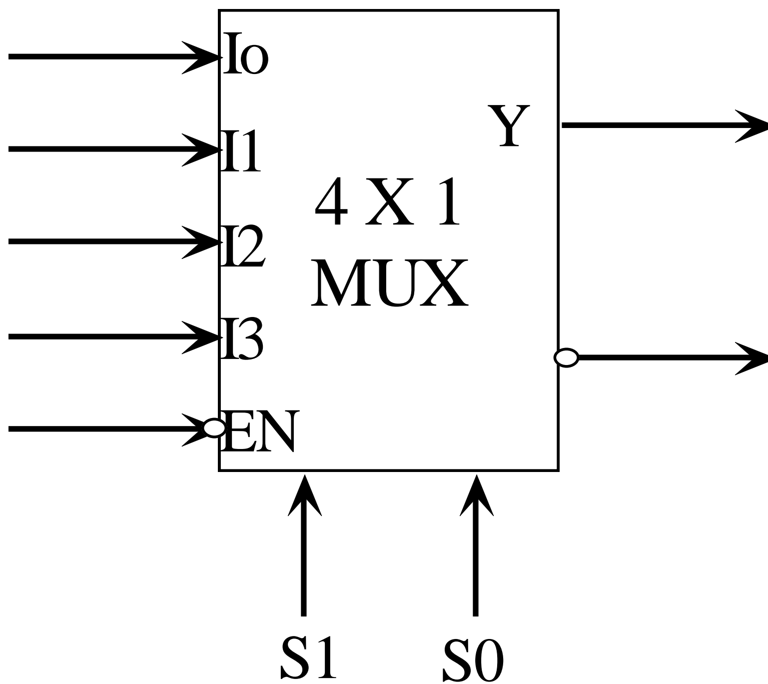
$$iY = \sum_{j=0}^{n-1} EN \bullet M_j \bullet iD_j$$

  - Summation symbol represents a logical sum of product terms
  - iY is a particular output bit (1 <= i< =b)
  - iDj is input bit I of source j (0<=j<=n-1)
  - Mj represents minterm j of the s select inputs
  - ? The relationship between S and n



3.2.1 Multiplexers (cont’d)

- Example 4 to 1 MUX
- How to implement?



EN	S1	S0	Y
1	x	x	0
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3

3.2.1 Multiplexers in VHDL (cont'd)

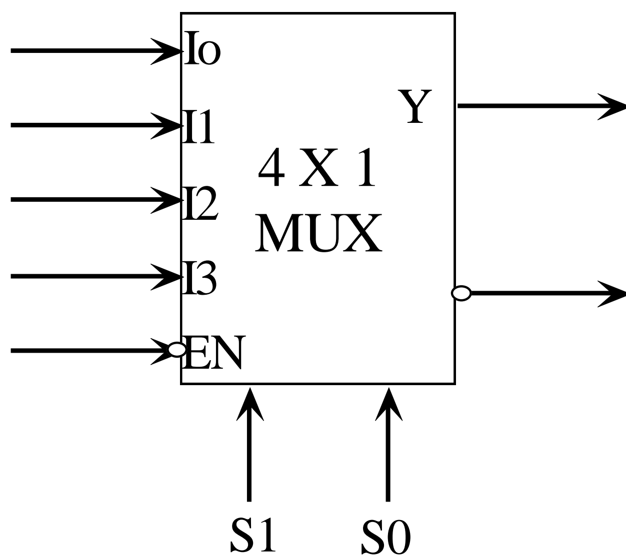
- Mutiplexers are very easy to describe in VHDL.
- Example for 4 in 1 bit MUX:
- **Entity** mux4in1b **is**
  - **Port** (S : **in** std_logic_vector (1 **downto** 0);
  - I0, I1, I2, I3 : **in** std_logic;
  - Y: **out** std_logic
  -);
- **End** mux4in1b;
- **Architecture** mux4_1b **of** mux4in1b **is**
  - **Begin**
    - **with** S **select** Y<=
    - I0 **when** "00",
    - I1 **when** "01",
    - I2 **when** "10",
    - I3 **when** "11",
- **End** mux4in1b;

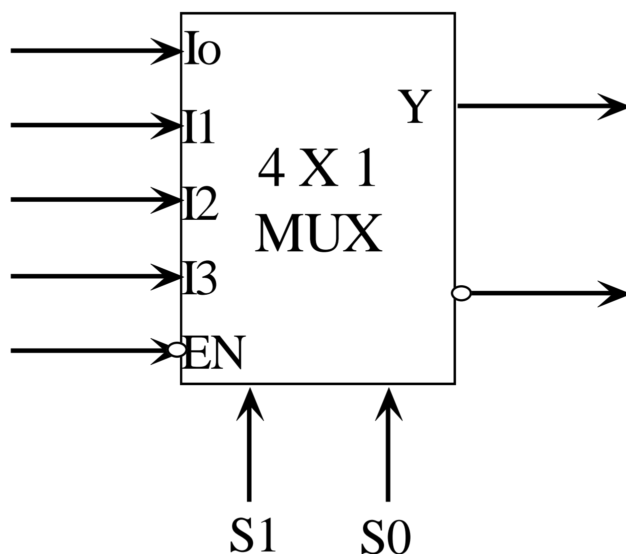
3.2.1 Expanding Multiplexers

- Expand 4 input 1 bit output multiplexer to 4 input 8 bit output multiplexer
- **Entity** mux4in8b **is**
  - **Port** (S : **in** std_logic_vector (1 **downto** 0);
  - I0, I1, I2, I3 : **in** std_logic_vector (1 to 8) ;
  - Y: **out** std_logic_vector (1 to 8)
  -);
- **End** mux4in8b;
- **Architecture** mux4in_8b **of** mux4in8b **is**
  - **Begin**
    - **with** S **select** Y<=
    - I0 **when** “00”,
    - I1 **when** “01”,
    - I2 **when** “10”,
    - I3 **when** “11”,
- **End** mux4in_8b;

3.2.1 Expanding Multiplexers (cont'd)

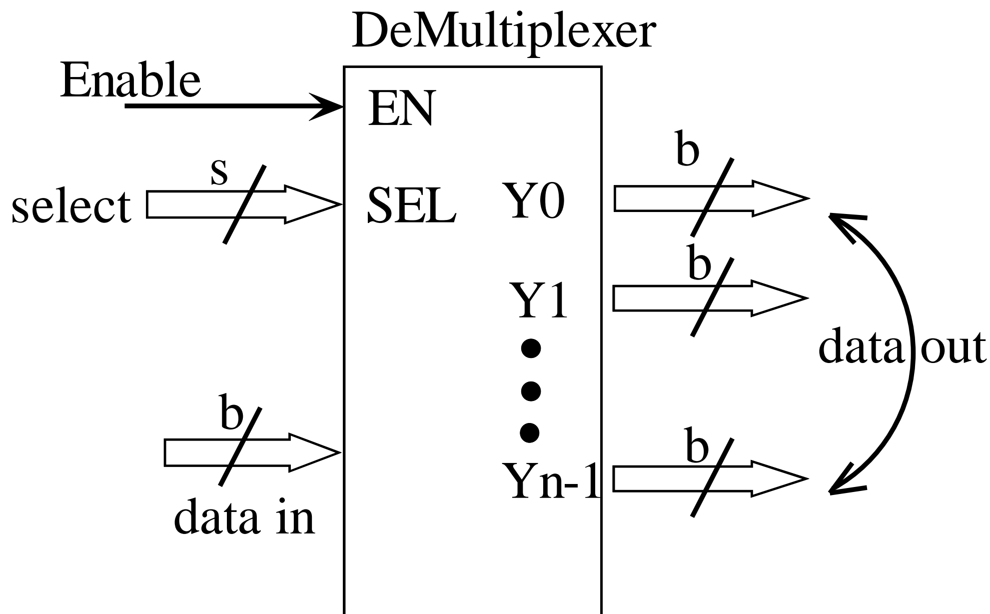
- Expand 4 input 1 bit output multiplexer to 8 input 1 bit output multiplexer
- How to implement it in VHDL?

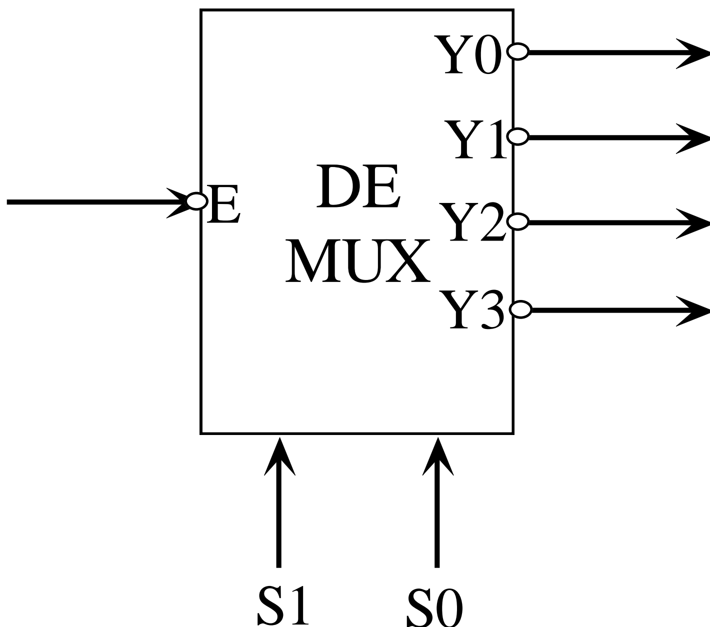




3.2.2 DeMultiplexers

- Used to direct data to one of two or more possible destinations





S1	S0	Y0	Y1	Y2	Y3
0	0	E	1	1	1
0	1	1	E	1	1
1	0	1	1	E	1
1	1	1	1	1	E

3.2.2 DeMultiplexers in VHDL

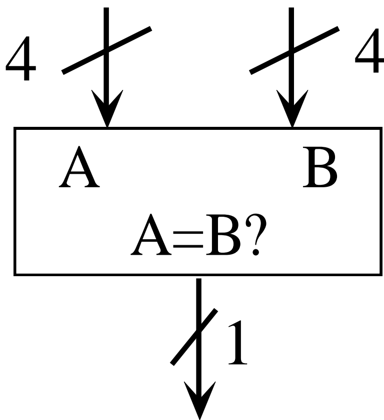
- Example for demux:
- **Entity** Demux **is**
  - **Port** (S : **in** std_logic_vector (1 **downto** 0);
  - E : **in** std_logic;
  - Y0, Y1, Y2, Y3: **out** std_logic
  -);
- **End** Demux;
- **Architecture** Demux **of** demux **is**
  - **Begin**

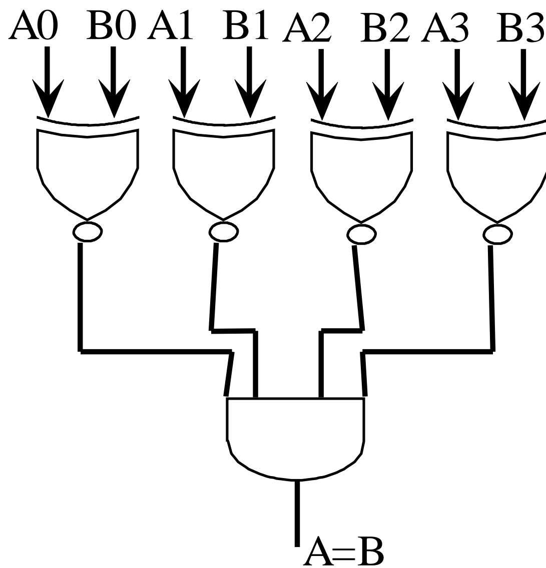
```
case S is
When “00” =>
        Y0<=E ;
When “01” =>
        Y1<=E ;
When “10” =>
        Y2<=E ;
When “11” =>
        Y3<=E ;
End case ;
```

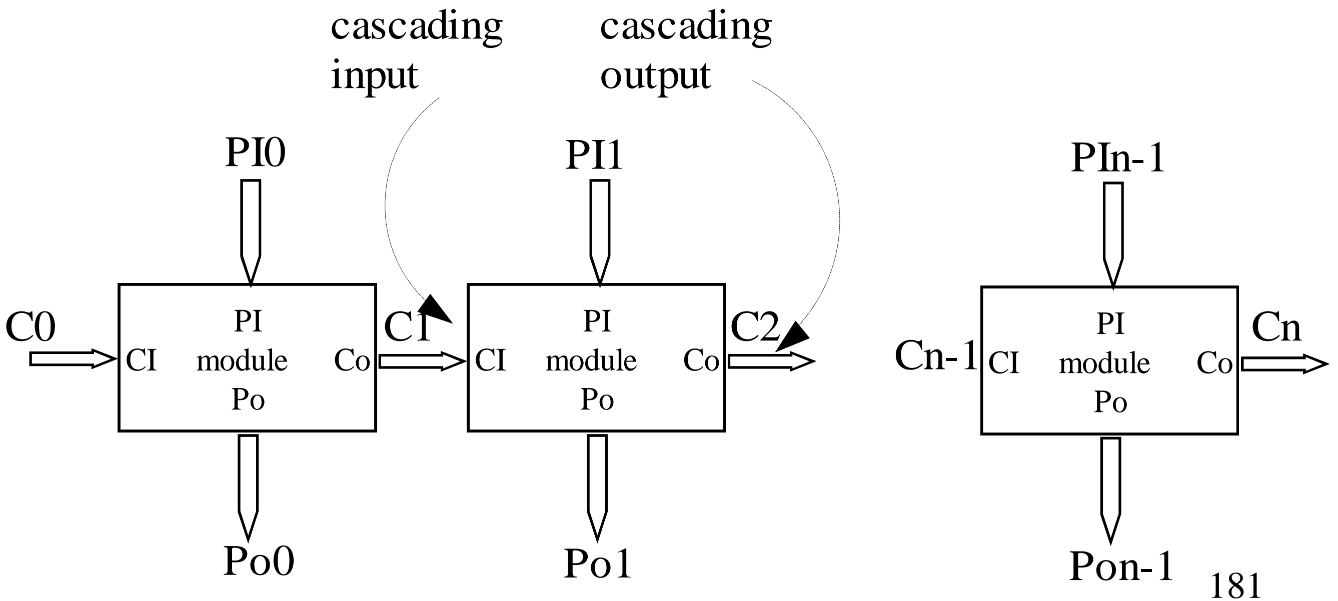
- **End** demux;

3.2.3 Comparators

- Comparing two binary words for equality







3.2.3 Comparators in VHDL

- Example of four bit equality comparator
- **entity** eqcomp4 **is**
- **port** (A0, B0, A1, B1, A2, B2, A3, B3 : **in** Std_logic; equals: **out** std_logic);
- **end** eqcomp4;
- **architecture** structure1 **of** eqcomp4 **is**
- **Signal** O0, O1, O2,O3 : Std_logic**;**
- **begin**
- O0 <= A0 xor B0;
- O1 <= A1 xor B1;
- O2 <= A2 xor B2;
- O3 <= A3 xor B3;
- Equals <= O0 and O1 and O2 and O3;
- **end** structure1;

- How to implement cascade comparator in VHDL ?

3.2.4 Binary Adders

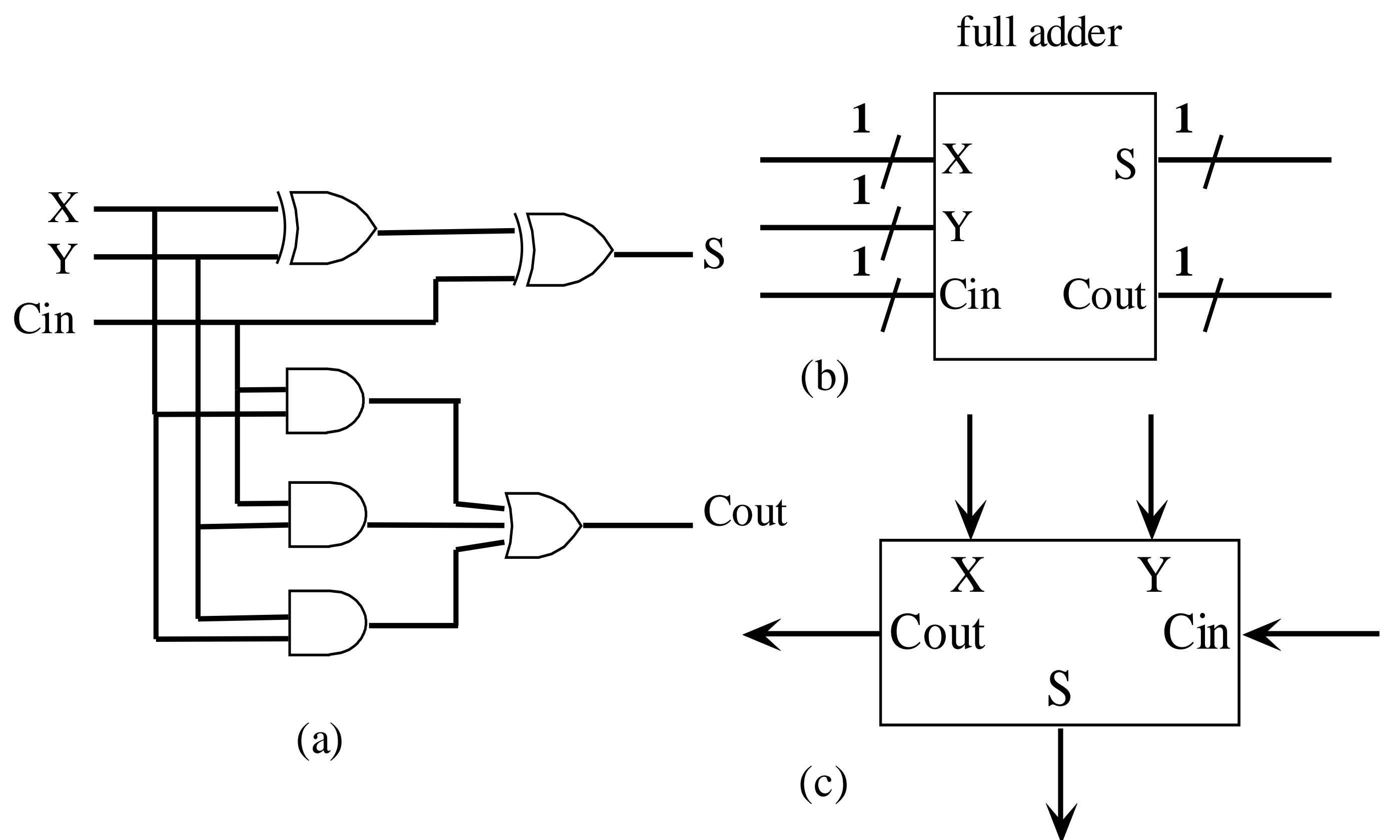
- Half adders: adds two 1 bit operands X and Y, producing a 2-bit sum. The lower-order bit of the sum may be named HS (half sum), and the higher-order bit may be named CO (carry out)
  - HS= X xor Y=XY'+X'Y
  - CO=XY
- Full adders: to add operands with more than one bit, we must provide for carries between bit positions. The building block for this operation called full adder.
- Besides the added-bit inputs X and Y, a full adder has a carry-bit input, Cin.
- Out put is named S (sum) and Cout (carry out)
  - S= X xor Y xor Cin
  - Cout=XY+XCin+YCin

3.2.4 Binary Adders (cont'd)

(a) One possible circuit that performs the full adder equations

(b) The corresponding logic symbol

(c) symbol for cascaded full adders



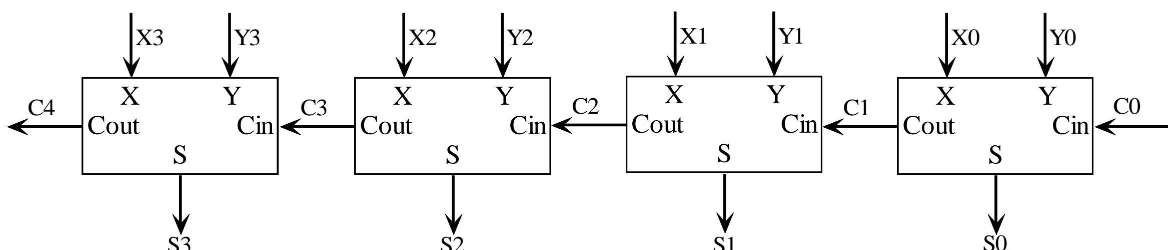
(a)

(b)

(c)

3.2.4 Binary Adders (cont'd)

- Entity for a full adder
  - **entity** adder **is**
  - **port** (X, Y, Cin : **in** Std_logic;
  - S, Cout : **out** Std_logic);
  - **end** adder;
- Ripple adder : two binary words, each with n bits, can be added using a ripple adder– cascade of n full-adder stages, each of which handles one bit.
- Ripple adder is slow, why ?
- Some other adders ?



3.2.4 Binary Adders in VHDL

- **entity** rippleadder **is**
- **port** (X, Y: **in** Std_logic_vector(0 to 3);
- S : **out** std_logic_vector(0 to 3);
- Cout : **out** Std_logic);
- **end** rippleadder;

Architecture structure1 of rippleadder is

component adder –pre-defined part type

port (X, Y, Cin : **in** Std_logic;

- S, Cout : **out** Std_logic);

End component;

Signal Cout0cin1, cout1cin2, cout2cin3 : in std_logic;

Begin

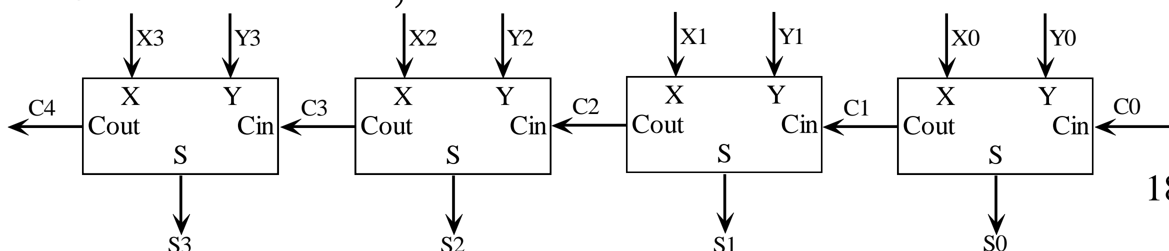
adder_0: adder **port map** (X=>X(0), Y=>Y(0),Cin =>low, S=> s(0), Cout => Cout0cin1);

adder_1: adder **port map** (X=>X(1), Y=>Y(1),Cin =>Cout0cin1, S=> s(1), Cout => Cout1cin2);

adder_1: adder **port map** (X=>X(0), Y=>Y(0),Cin =>Cout1cin2, S=> s(2), Cout => Cout2cin3);

adder_1: adder **port map** (X=>X(0), Y=>Y(0),Cin =>Cout2cin3, S=> s(3), Cout => Cout);

End structure1 ;



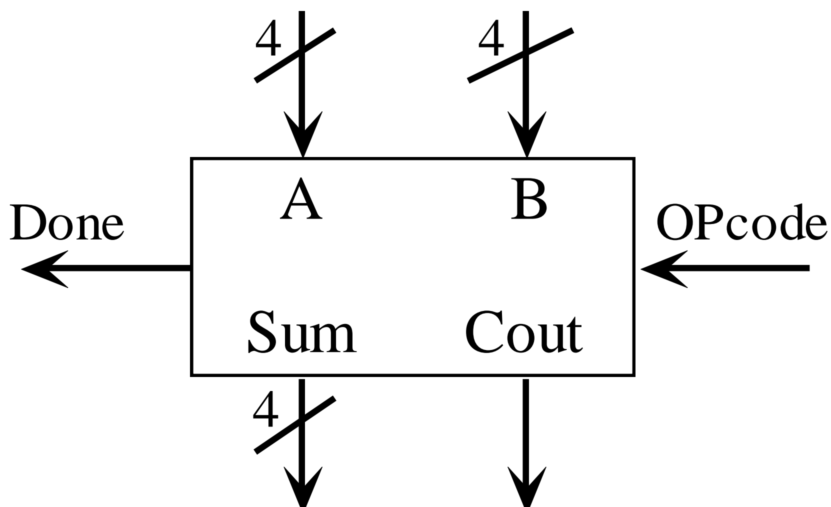
3.2.4 A Design Example: Recursive Adder/Subtractor

- Objective: Implement an adder and subtractor using VHDL.
- Design steps

- Define Specifications
- Data Path Design
- Control Path Design
- Simulation
- Hardware Implementation
- Testing

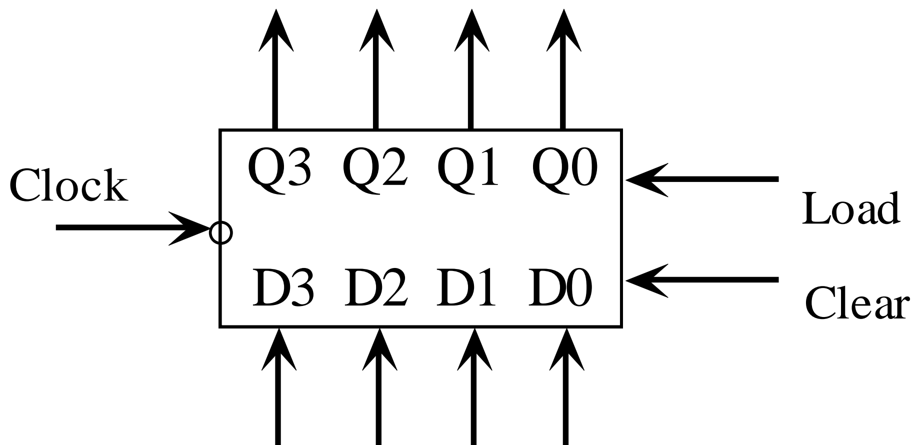
3.2.4 Define Specifications

- 4-bit Switch Input A
- 4-bit Switch Input B
- 1 bit switch for Opcode.
- LED output 4 LED for Sum, one for
- Cout, one done signal
- Restricted building blocks:
  - An inverter, 2 2N-N mux, three registers and an adderN.

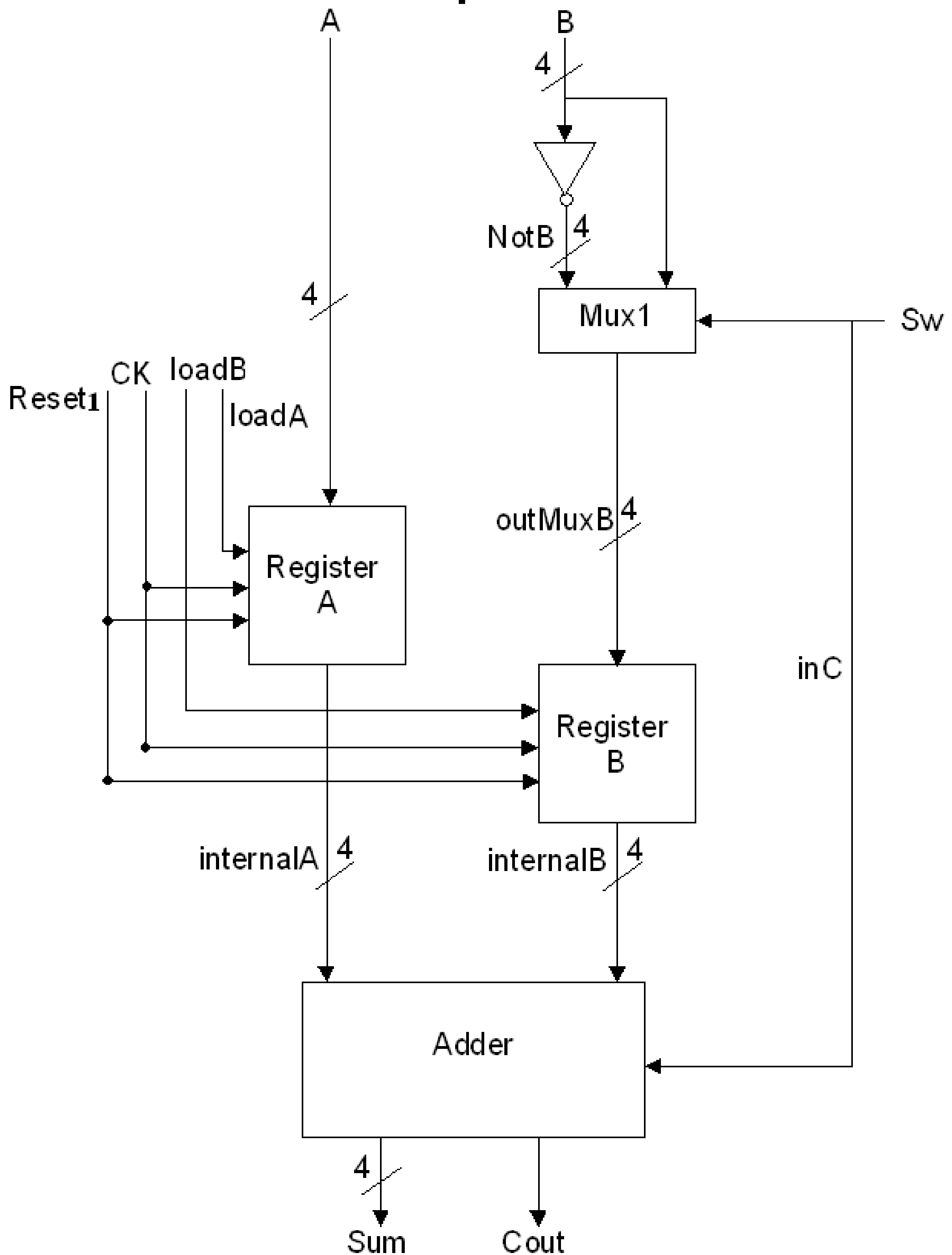


3.2.4 Define Specifications

- Opcode 0, A+B
  - example 0101+0111
- Opcode 1, A-B=A+NotB+1
  - Example 1100-0011=?
- Adder implementation
  - Ripple adder
  - fast adder
- Register ? Synchronize the add and subtract procedure



3.2.4 Example : Data Path



3.2.4 Example : Data Path Entity

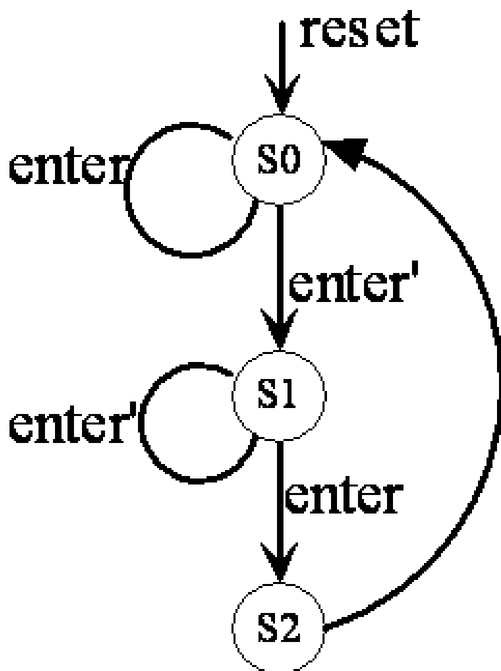
- **Entity** DataPath **is**
  - **port** (A, B **in** Std_logic_vector(3 downto 0);
  - loadA, loadB : **in** std_logic;
  - CK, reset1 ,sw : **in** std_logic;
  - Sum : out std_logic_vector(3 downto 0);
  - Cout : out std_logic);
- **end Entity** DataPath;

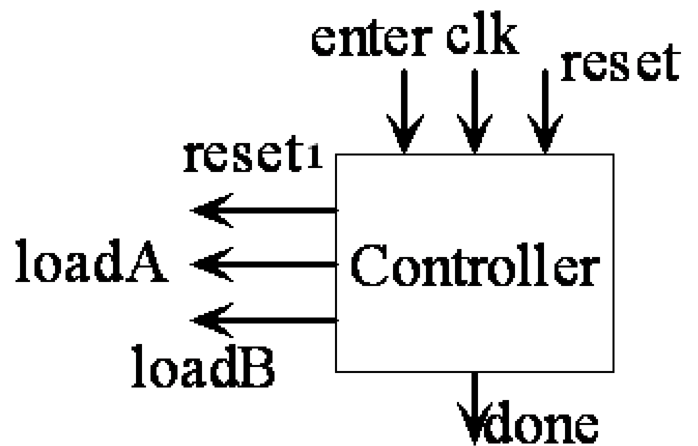
- **Architecture** struture1 **of** DataPath **is**
  - **Signal** NotB : std_logic_vector(3 downto 0);
  - **Signal OutMuxB** : std_logic_vector(3 downto 0);
  - **Signal InternalA, InternalB :** std_logic_vector(3 downto 0);
  - Signal inC: std_logic;
  - Begin
  - …
- **End architecture** structure1;

3.2.4 Example : Control Path

- Control Signals in Data Path
- Sw Control the mode
- Reset, reset the output active low
- CK, clock synchronize the calculation of Sum and sub
- Load A, Load B: allow the input of register to be loaded into the register.
- Done : High when the calculation is done, low when the reset is activated.

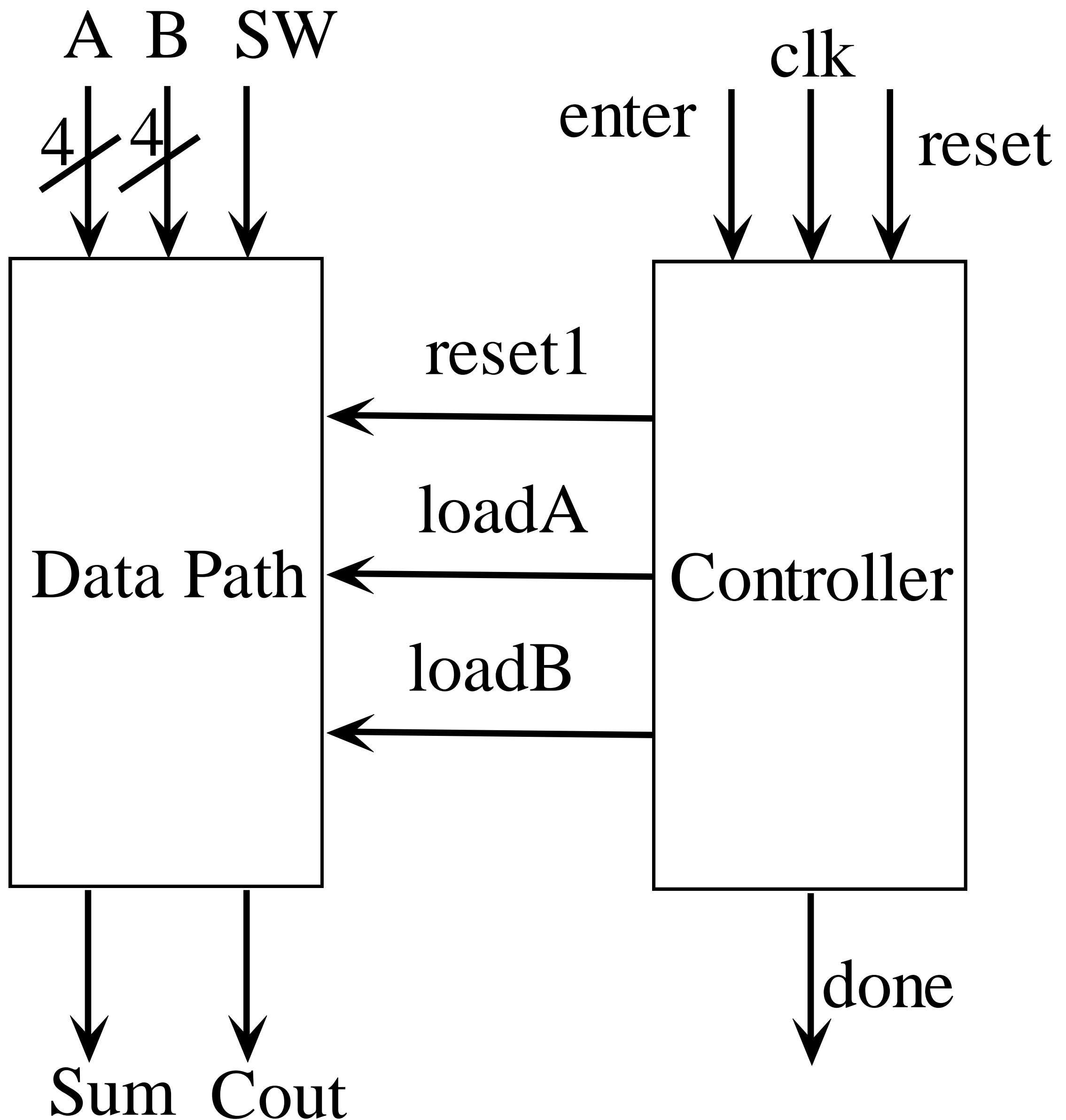
3.2.4 Example : Control Path





	next state		output			
	enter=0	enter=1	loadA	loadB	reset1	done
s0	s1	s0	0	0	1	0
s1	s2	s1	1	1	0	0
s2	s0	s0	0	0	1	1

3.2.4 Example : Final Design

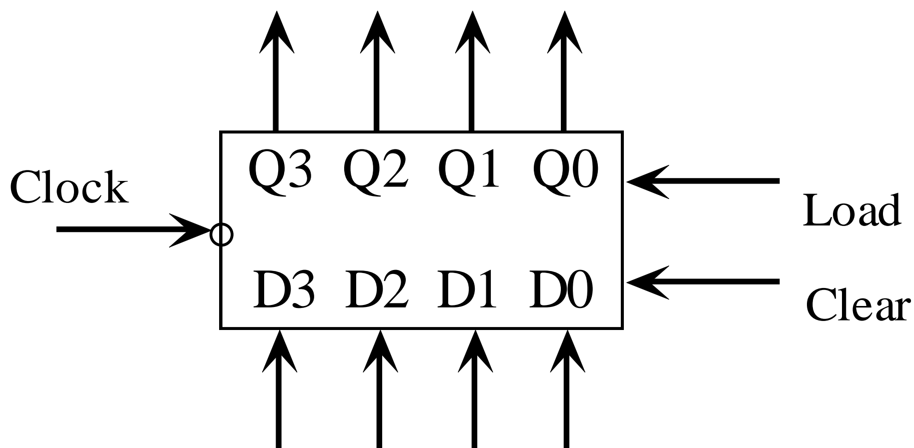


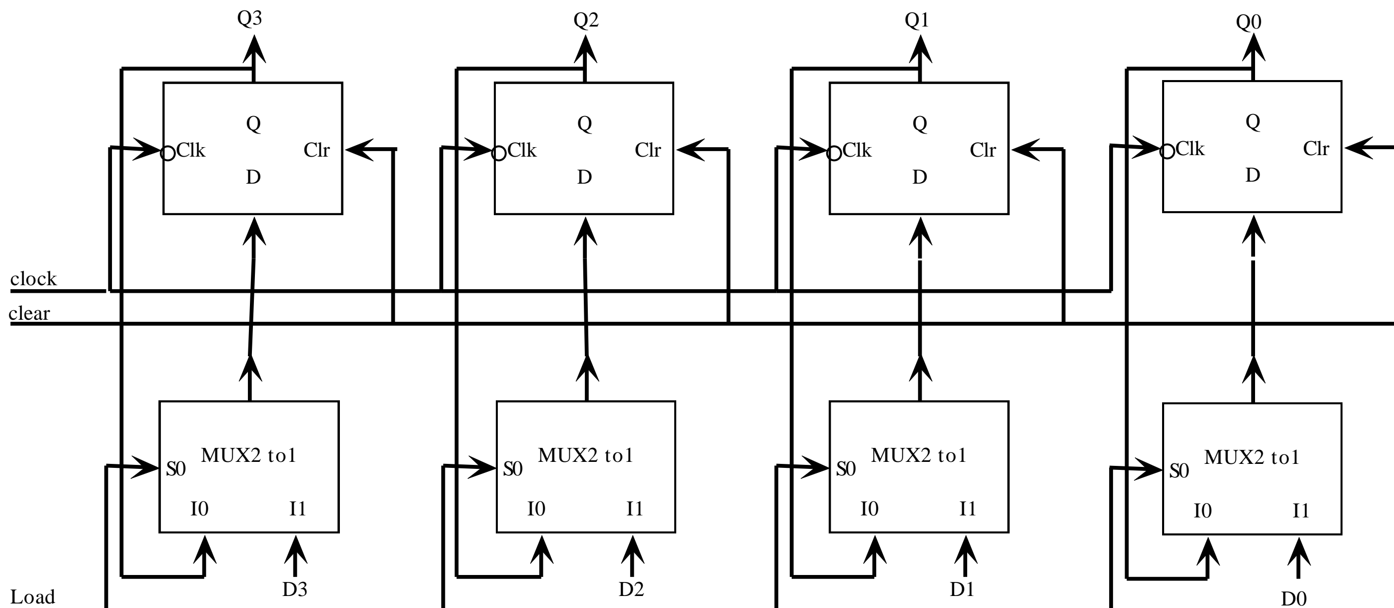
3.2.4 Hardware Implementation

- Wiring
- Pin assignment
- Programming
- Testing

3.2.5 Shift Register

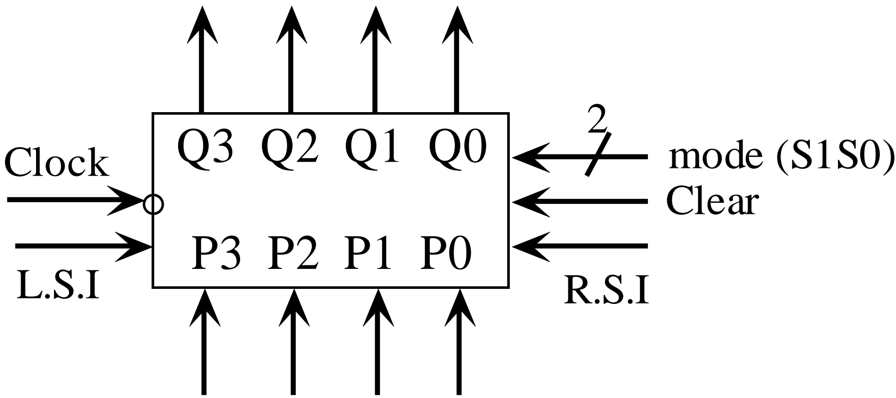
- A shift Register is an n-bit register with a provision for shifting its stored data by one bit position at each tick of the clock. Following is a 4-bit register:



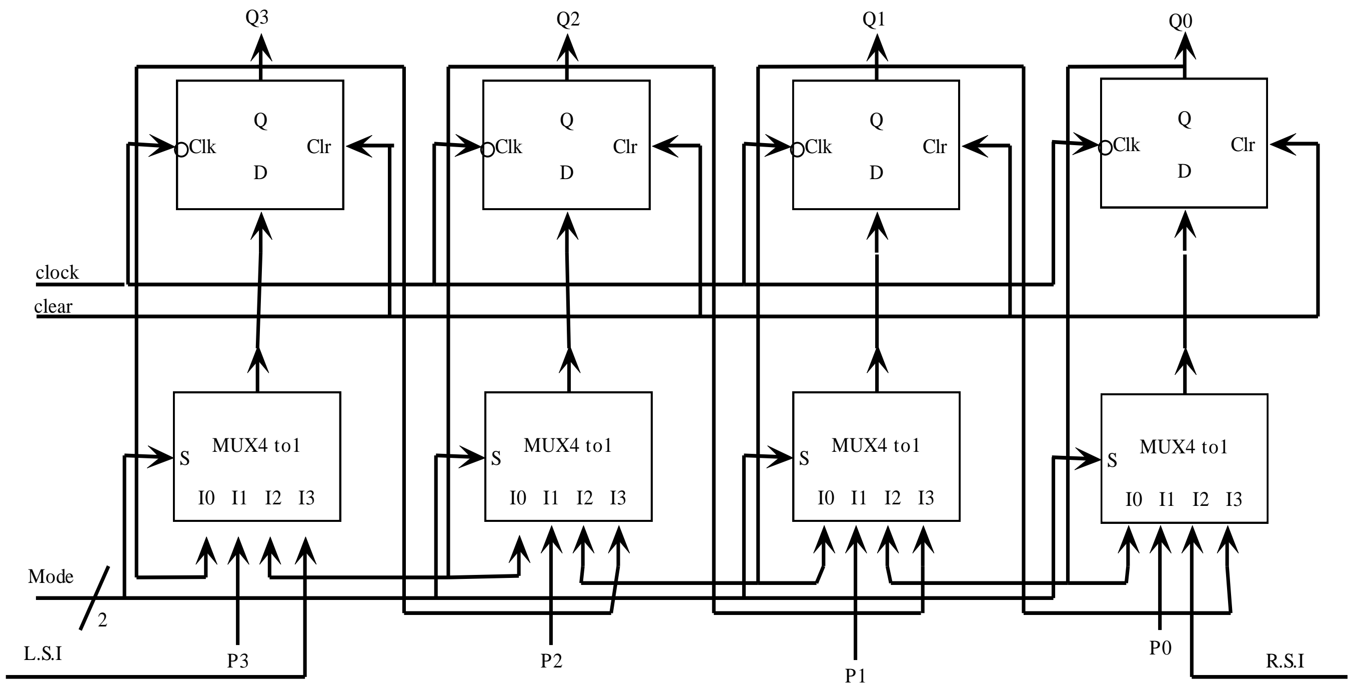


3.2.5 Shift Register (cont'd)

- Multi-mode 4 bit shift register



S1	S0	Mode
0	0	Hold
0	1	Load
1	0	Shift left
1	1	shift right

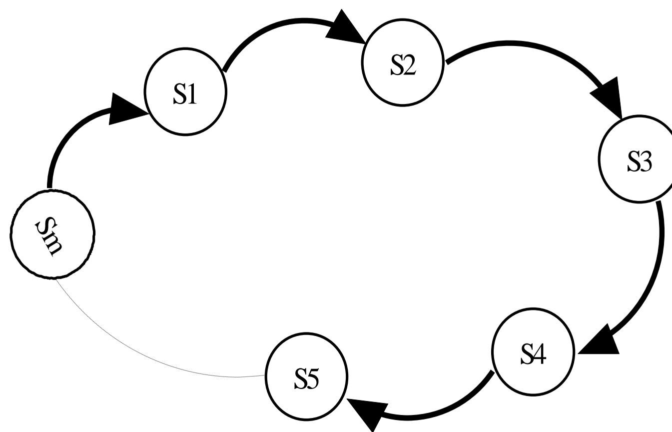


3.2.5 Shift Register in VHDL

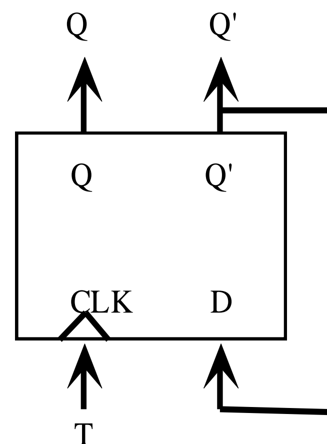
- Entity Vshftreg is
  - Port(
  - Clk, clr,rin,lin : in std_logic;
  - S: in std_logic_vector (2 downto 0); --function select
  - D: in std_logic_vector(7 downto 0); --data in
  - Q: out std_logic_vector (7 downto 0) –data out
  -);
- End entity;
- Architecture vshftreg_arch of vshreg is
  - Signal Iq: std_logic_vector (7 downto 0);
- Begin
- Process(clk,clr,iq)
- begin
  - If(clr='1') then Iq <= (others=>'0');--asynchronous clear
  - Elsif (clk'event and clk='1') then
  - Case conv_integer(s) is
    - **When** 0 => null; --hold
    - **When** 1=> iq <=D; --load
    - **When** 2 =>iq <=rin & iq(7 downto 1); --shift right
    - **When** 3 => iq <= iq(6 downto 0) & lin; --shift left
    - **When** 4=> iq <=iq(0) & iq(7 downto 1); --circular right
    - **When** 5 => iq <= iq(6 downto 0) & Iq(7); --circular left
    - **When** 6 => iq <=iq(7) & iq(7 downto 1); --shift arith right
    - **When** 7 => iq <=iq(6 downto 0) & '0'; --shift arith left
    - **When** others => null;
  - **End case;**
  - **End if;**
  - Q<=iq;
- End process;
- End vshftreg_arch;

3.2.6 Counters

- Counter is generally used for any a clocked sequential circuit whose state diagram contains a single cycle.
  - The modulus of a counter is the number of states in the cycle
  - Counter with m states is called a modulo-m counter of a divide-by-m counter

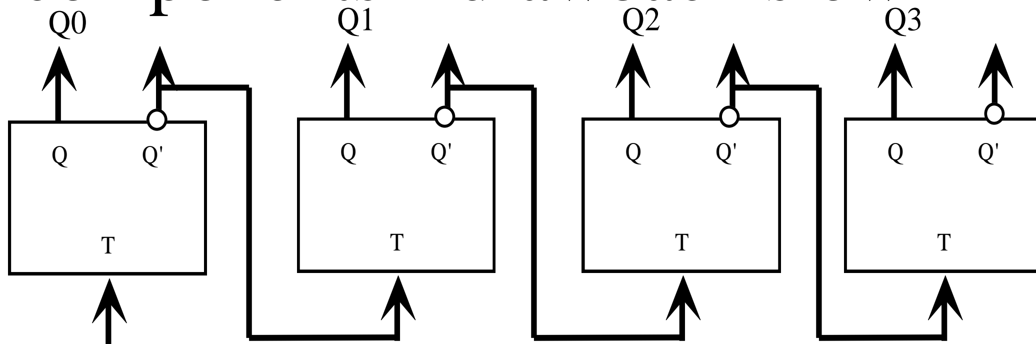


Use D flip flop to construct a T Flip flop

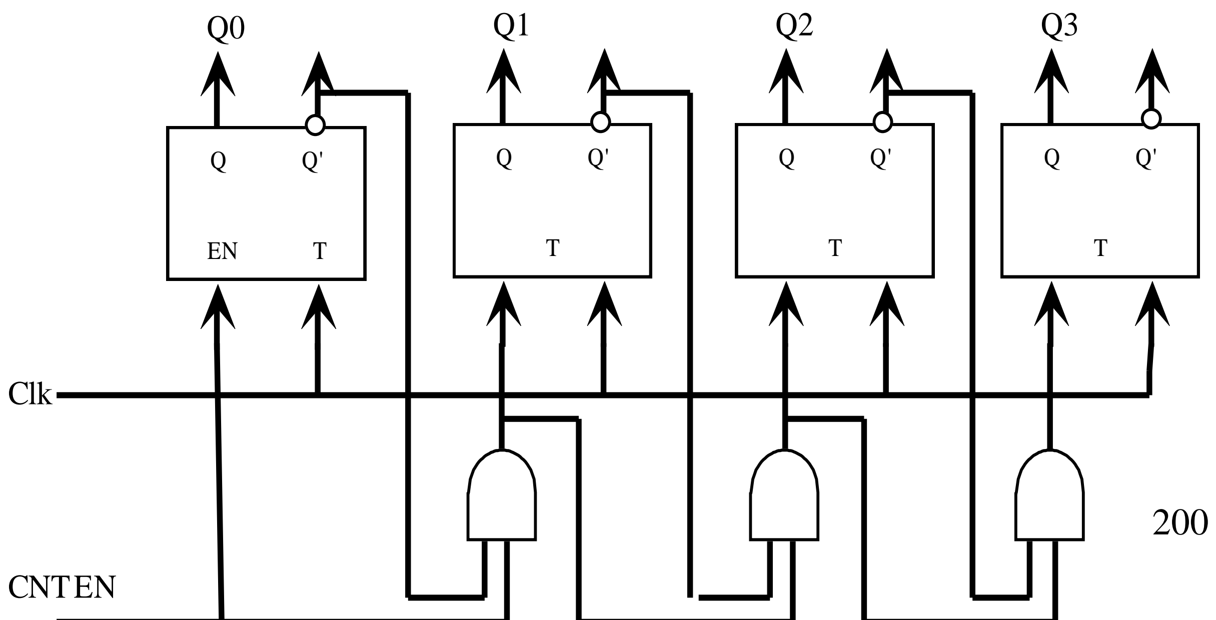


3.2.6 Counters (cont'd)

- Ripple counters : can be constructed with just n flips –flops and no other components—drawback slow



- Synchronous counters: connect the inputs to the same common Clk signal.
- Clock period should > propagation delay.—improvement: synch parallel counter

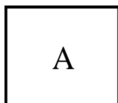


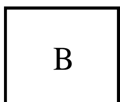
3.2.6 Counters (cont’d)

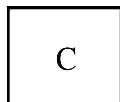
- LFSR: linear Feedback shift register counters.
  - Many shift register counters have far less than the maximum of 2^n normal states (n-bit).
  - LFSR can have $2^n - 1$ states.
  - LFSR is called maximum-length sequence generator.
  - Based on finite field theory Evariste Galois (1811-1832)

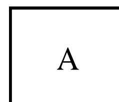
3.2.7 Buses

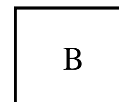
- Bus is a collection of two or more related signals lines. They are used to move data around within a system and among systems.
  - Bus are drawn with a double or heavy line
  - A slash and a number may indicate how many individual signal lines are contained in a bus
  - Size may be denoted in the bus name (e.g. inbus[31..0].
  - Why we need a bus?

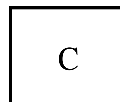












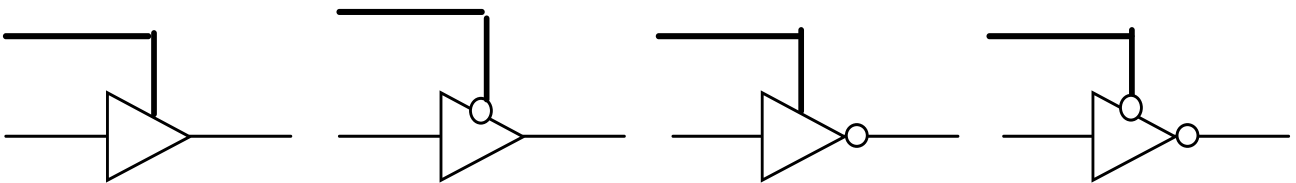
3.2.7 Buses (cont'd)

- Direct connection
- Advantages:
  - High bandwidth
  - No sharing required
- Disadvantages:
  - Number of wires increases rapidly as each new system is added

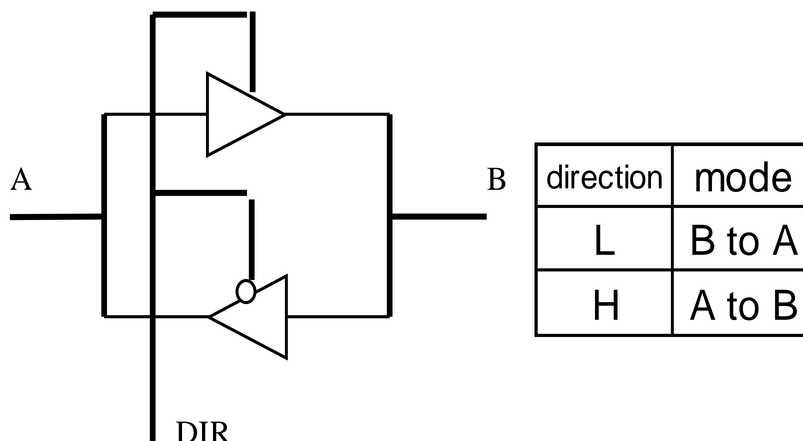
- Bus connection
- Advantages:
  - Less wiring
  - Easy & cheap to add a new system
- Disadvantages:
- Bus becomes a bottle neck
- Need control circuitry to prevent bus contention

3.2.8 Three State Buffers

- Three state buffer or three state driver.
  - Three states 0, 1 or Hi-Z
  - Various three-state buffers
    - A) non-inverting, active high enable
    - B) non inverting, active-low enable
    - C) inverting, active-high enable
    - D) inverting, active-low enable



Bi-directional transceivers

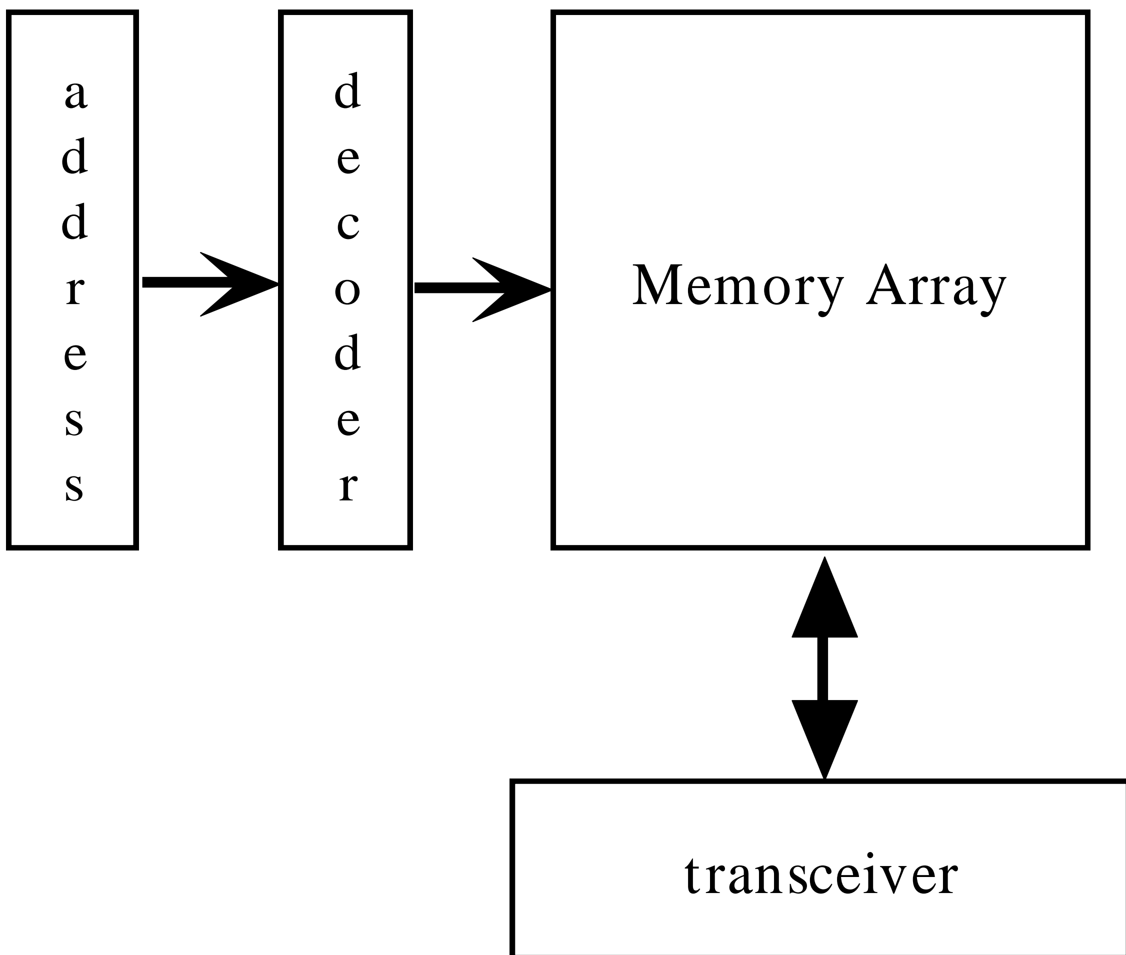


direction	mode
L	B to A
H	A to B

3.2.9 RAM

- RAM: random access memories, which means that the time it takes to read or write a bit of memory is independent of the bits location in the RAM.
- SRAM (static RAM): once a word is written at a location, it remains stored as long as power is applied to the chip. Unless what ?
- DRAM (dynamic RAM): the data stored at each location must be refreshed periodically by reading it and then writing it back again. Why ?

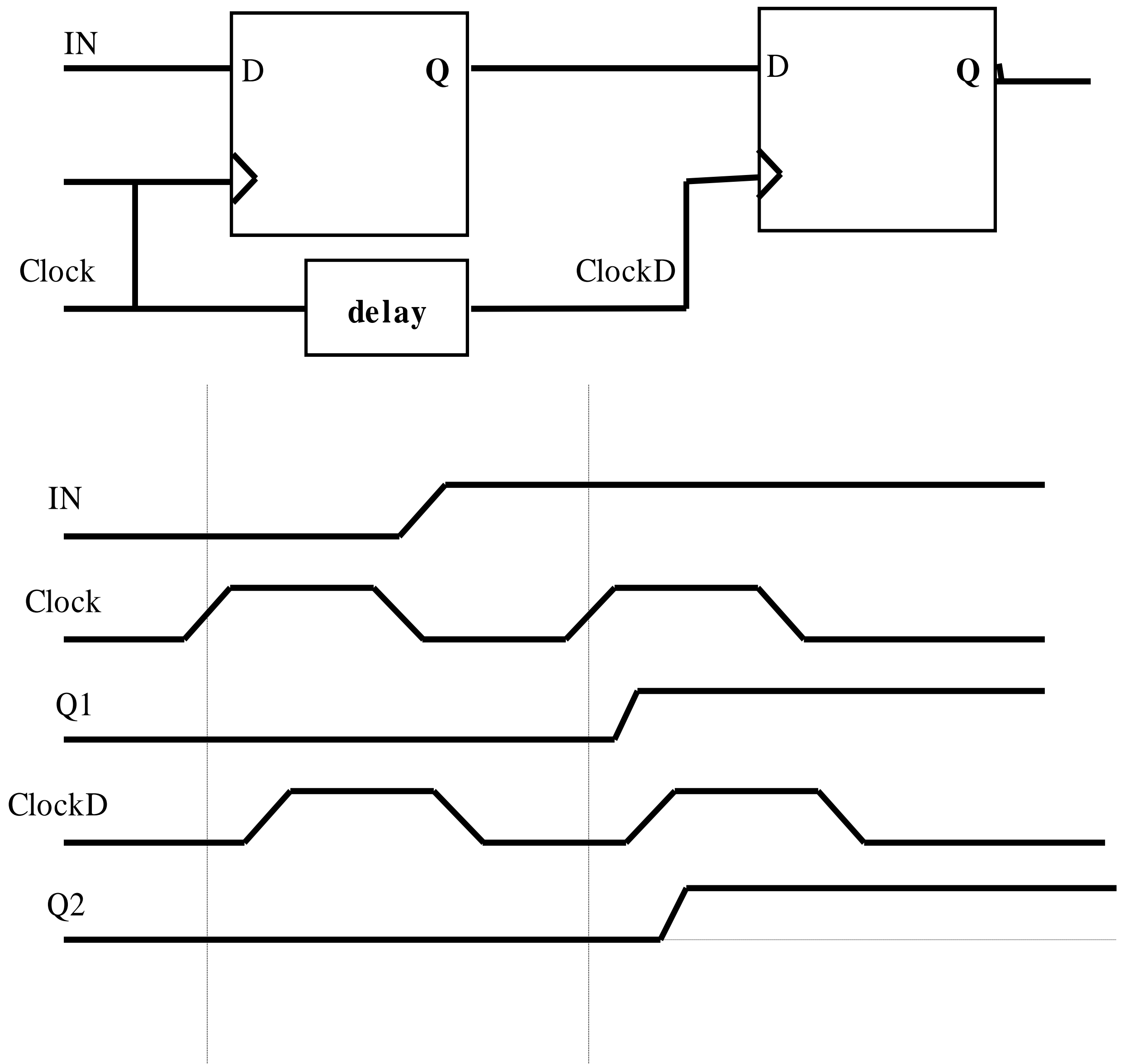
3.2.9 RAM (cont'd) : a Simplified Block Diagram of RAM



3.3 Impediments to Synchronous Design

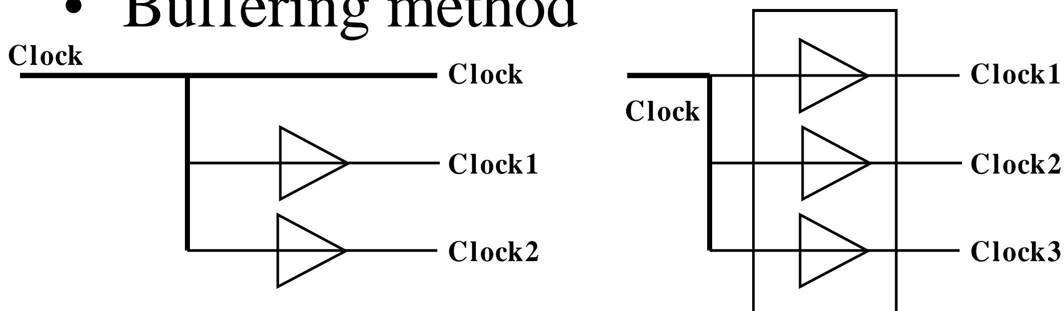
- Synchronous approach is the most straightforward and reliable method of digital system design, however…
- Synchronous systems using edge-triggered flip-flops work properly only if all flip-flops see the triggering clock at the same time.
- Clock skew: the situation when the clock signal arrives at different flip-flops at different times
  - Caused by unequal clock propagation times
  - Clock skew may cause flip-flops to load transient input signals.

3.3.1 Clock Skew : Example of Clock Skew



3.3.1 Clock Skew (cont'd)

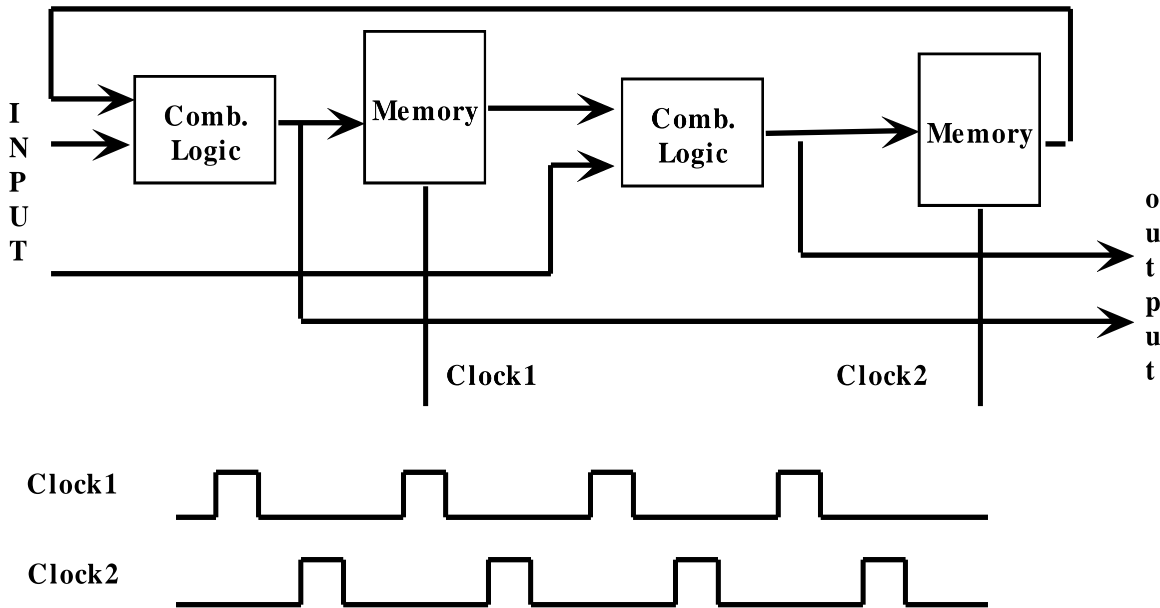
- Clock skew are caused by
  - Buffering method



  - DC and AC loading
  - Signals on PCB are auto routed by CAD
  - Some wire maybe slower than other
- To control this problem, many high performance systems and VLSI chips use a two-phase latch design

3.3.1 Two Phase Clocking

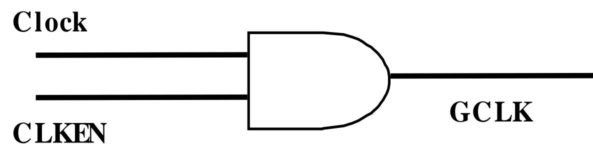
- Popular in custom CMOS ICs
- Standard scheme at IBM

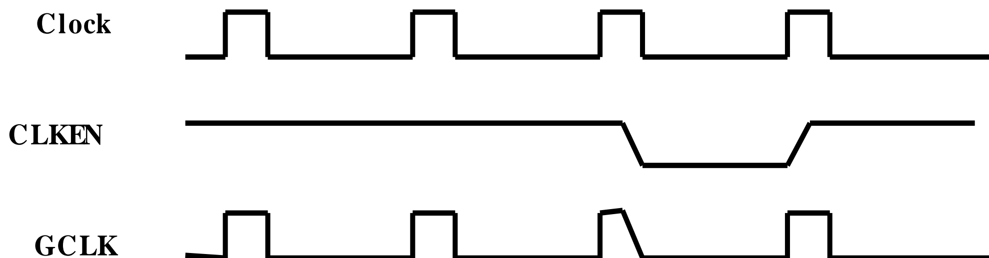


- Advantages:
  - Simple memory elements
  - Essential hazards avoided
  - Clock skew problems avoided
- Disadvantages
  - Two separate clock signals are required
  - Non-overlapping condition must be guaranteed to ensure correct operation

3.3.2 Gating the Clock

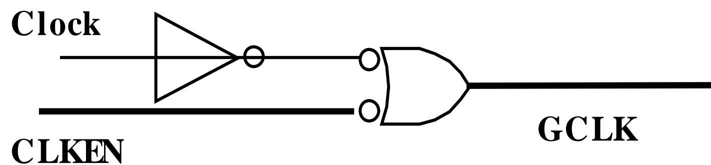
- If CLKEN is a state machine output or other signal produced by a register clocked by clock, the CLKEN changes some time after clock has already gone high. This produces glitches and false clocking of the registers controlled by GCLK
- AND gate delays gives GCLK excessive clock skew, which cause problems.

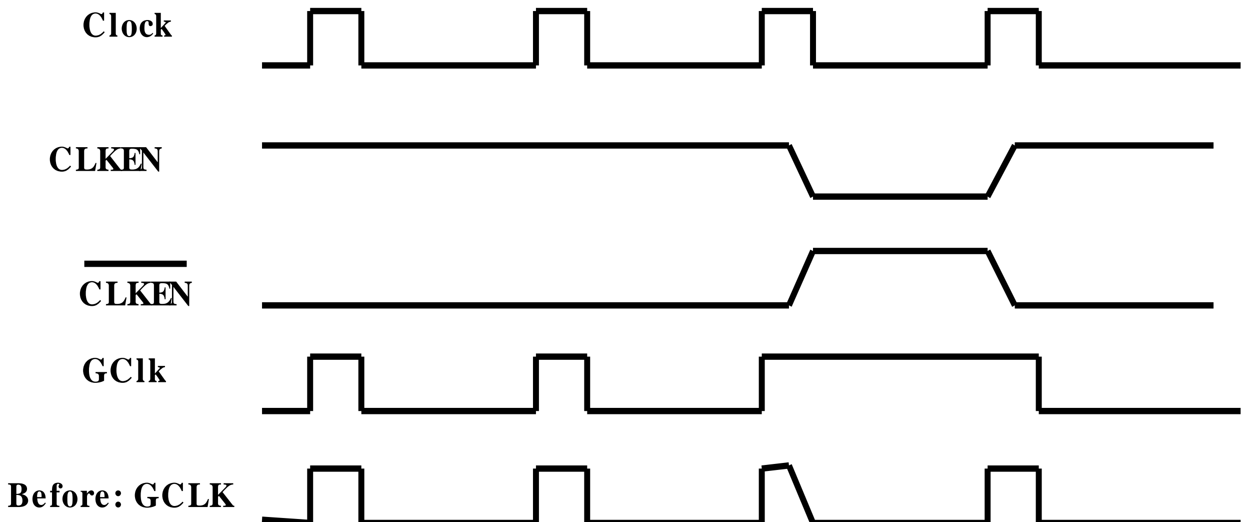




3.3.2 Gating the Clock (Cont’d)

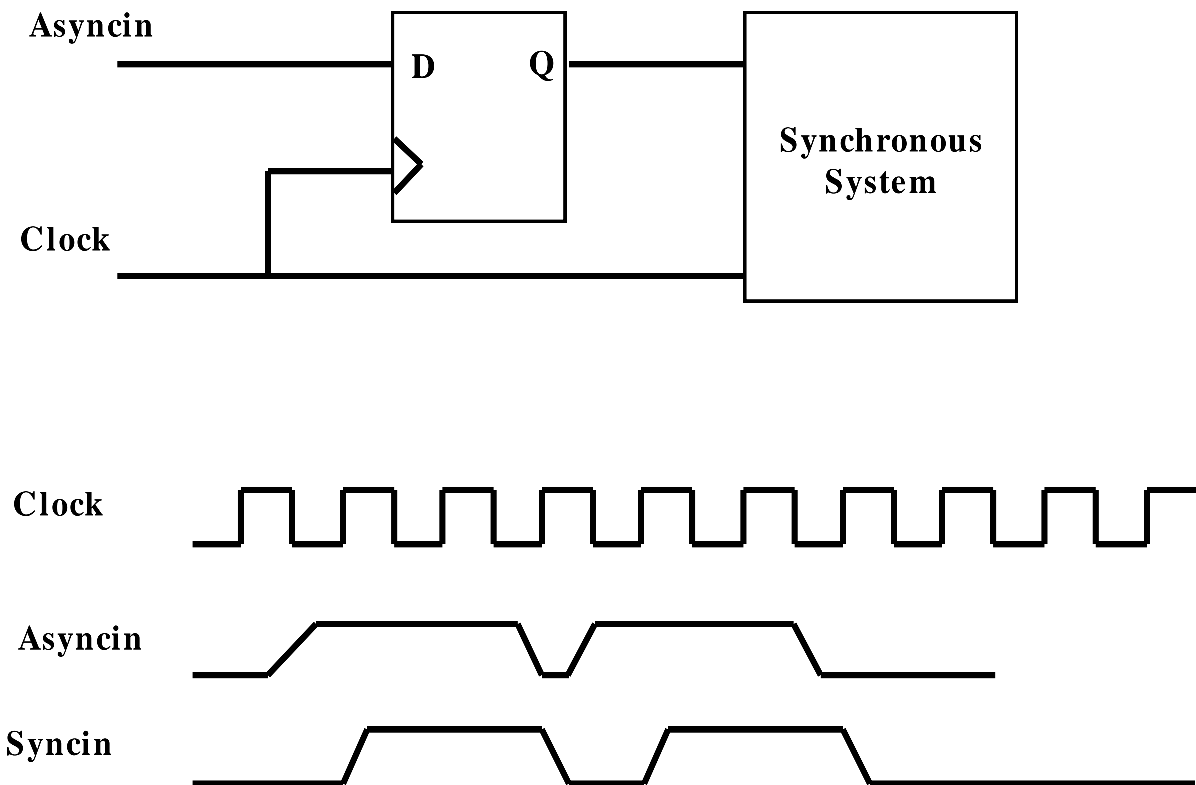
- An acceptable way to gate the clock





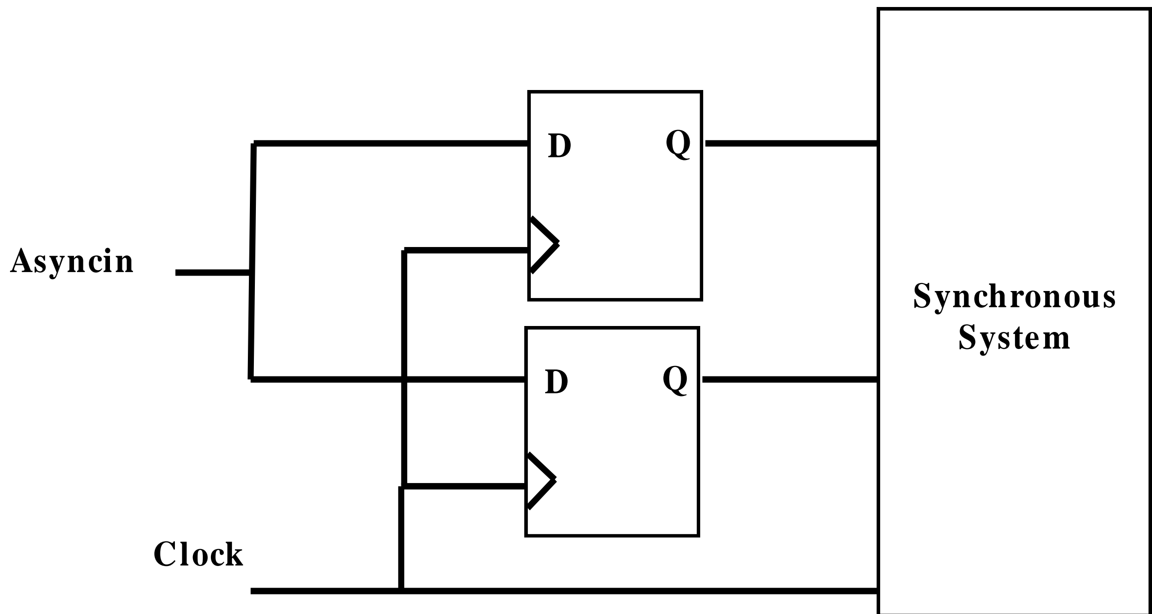
3.3.3 Asynchronous Input

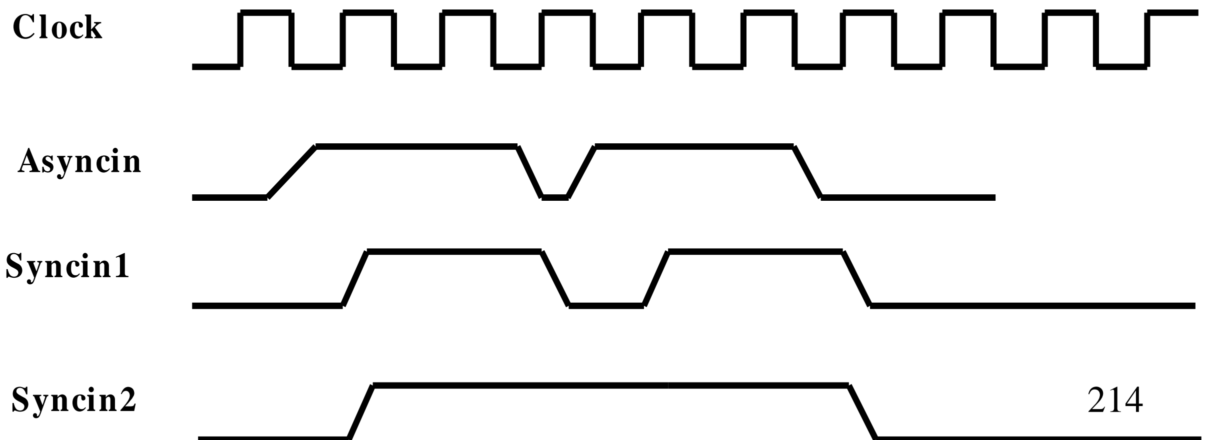
- There are always asynchronous inputs
  - Key input (very low frequency)
  - Interrupts
  - Status flags
- Solution synchronizer:



3.3.3 Asynchronous Input (cont'd)

- It is essential for asynchronous inputs to be synchronized at only one place





3.4 Variable Entered Maps (VEM)

- Add the power to K-map method
- Reduce the work to plot and read maps
- A technique to reduce the map size.

- Regular K-maps entries
- 1
- 0
- Don't care

- VEM entries
- 1
- 0
- Don't care
- Boolean variables
- expressions

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	Φ
1	1	1	Φ

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

A	B	C	F	F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	c'
0	1	1	1	c
1	0	0	1	c'
1	0	1	0	0
1	1	0	Φ	c'o
1	1	1	Φ	co

B \ A	0	1
0	0	c'
1	1	Φ

3.4.1 Variable Entered Maps (cont'd)

- VEM are most effective when a function depends strongly on <= 4 inputs and depends only weakly on the remaining inputs.
- Map Entered variable (MEV) : a variable that appears in a box in a VEM

A	B	C	F
0	0	0	f0
0	0	1	f1
0	1	0	f2
0	1	1	f3
1	0	0	f4
1	0	1	f5
1	1	0	f6
1	1	1	f7

C \ AB	00	01	11	10
0	f0	f2	f6	f4
1	f1	f3	f7	f5

A	B	C F
0	0	c'f0+cf1
0	0	
0	1	c'f2+cf3
0	1	
1	0	c'f4+cf5
1	0	
1	1	c'f6+cf7
1	1	

B \ A	0	1
0	c'f0 +cf1	c'f4+ cf5
1	c'f2 +cf3	c'f6+ cf7

3.4.2 Plotting the Map

- How to plot a VEM from a truth table
  - Select the MEV
  - Partition the truth table so that the non-MEVs have the same values in each partition

A	B	C	F1	F2	F3	F4
0	0	0	0	1	0	1
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	1	1
1	0	0	1	0	0	1
1	0	1	1	0	1	0
1	1	0	0	1	1	0
1	1	1	0	1	0	Φ

B \ A	0	1
0	c	1
1	c'	0

B \ A	0	1
0	c'	0
1	c	1

B \ A	0	1
0	0	c
1	1	c'

B \ A	0	1
0	1	c'
1	c	cΦ

3.4.2 Plotting the Map(cont'd)

- How to plot a VEM from an equation
  - Rearrange the function into S.O.P form
  - Identify the most dependant variable
  - Factor out the minterms in the identified variables
  - Draw out an map
  - Fill in the VEM
  - Example :

$$F = BCEFH + \overline{B}CDEH + \overline{B}\,\overline{C}\,\overline{E}H$$
$$+ A\overline{B}CE\overline{H} + \overline{B}\,\overline{C}E + B\overline{E}H$$
$$+ B\overline{C}E\overline{G}\,\overline{H} + A\overline{B}C\overline{E}\,\overline{H}$$

Don't Cares

$\overline{B}C\overline{E}H$ $\quad B\overline{C}\,\overline{E}\,\overline{G}\,\overline{H}$ $\quad B\overline{C}E\overline{G}H$

3.4.3 Reading Theory

- Step1: first image that all 1 entries in the map are replaced by the map entered variables Ored with its complement. 1=D+D'
  - First loop all single MEV entries that will not loop with another identical MEV in an adjacent cell or with a 1 or don't care (island).
  - Loop all MEV's that will loop into duals only with another identical MEV in an adjacent cell
  - Loop all MEV's that will loop into a dual only with a 1.
  - Loop all MEV's that will loop into a dual only with a don't care.
  - Any MEV that will loop two ways with another identical MEV, 1 or don't care but won't loop into a quad, leave until later
  - Continue looping in similar fasion for quads and groups of eight until every MEV has been looped at least once

3.4.3 Reading Theory (cont'd)

- Step2: once all single MEV entries have been covered, transform the map according to the following transformations:
  - A) Replace the MEV and MEV' with 0.
  - B) 0 to 0, don't care to don't care
  - C) 1 : two possible transformations:
    - 1 if not completed covered
    - Don't care if completed covered, I.e. looped with both the MEV and MEV'

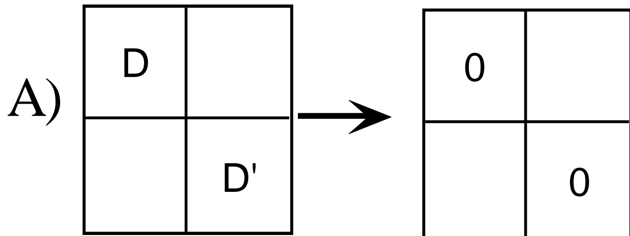
$$D) \quad MEV\Phi \; and \; \overline{MEV}\Phi \rightarrow 0$$

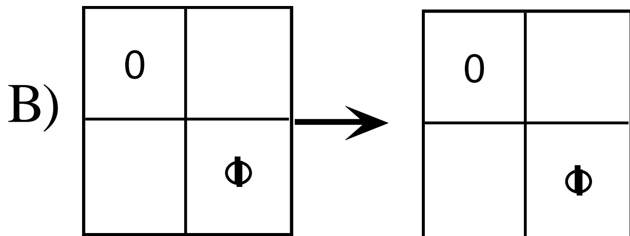
$$E) \quad (MEV + \overline{MEV}\Phi) \; and \; (\overline{MEV} + MEV\Phi)$$

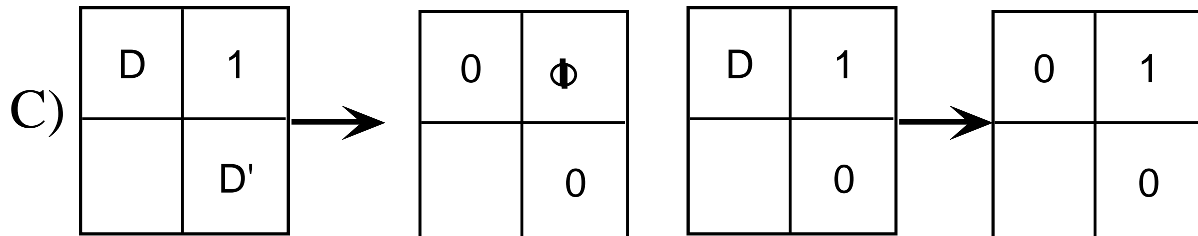
$$\begin{cases} 1 & if\ not\ covered\ at\ all\ or\ if \\ & just\ the\ \Phi\ is\ covered \\ \Phi & if\ completed\ covered\ or \\ & if\ just\ the\ necessary\ term \\ & is\ covered \end{cases}$$

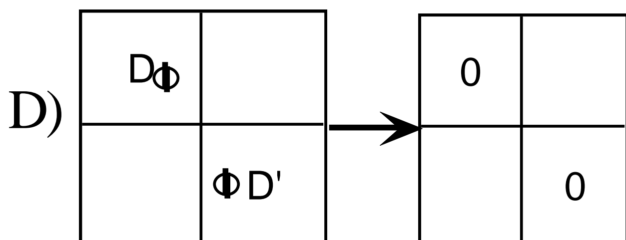
- Step3: OR together the terms from steps 1 and 2

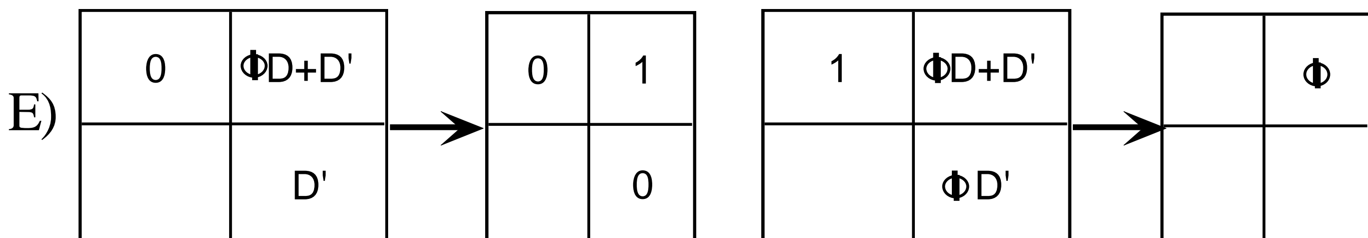
3.4.3 Reading Theory (cont’d)











3.4.3 Reading Theory (cont'd): example1

given

B \ A	0	1
0	C	0
1	1	C

B \ A	0	1
0	C	0
1	C+C'	C

step1

B \ A	0	1
0	C	0
1	C+C'	C

step2

B \ A	0	1
0	0	0
1	1	0

$$F = \overline{A}C + BC + \overline{A}B$$

3.4.3 Reading Theory (cont'd): example2

A	B	C	D	F1	F2
0	0	0	0	Φ	0
0	0	1	1	Φ	0
0	0	1	0	1	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	Φ	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	Φ
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	Φ
1	1	1	0	0	1
1	1	1	1	Φ	0

C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

$$F1 = B\overline{C}D + \overline{A}\,\overline{D} + A\overline{B}C$$

$$F2 = \overline{A}BD + C\overline{D} + \overline{A}C$$

3.5 Design Steps for Digital System Design

1. Define the system specifications
2. Develop a rough design for the system
3. Develop a detailed design for the data path
4. Develop a detailed specification for the controller
5. Complete the design of the controller
6. Finalize the design
7. Simulation and hardware implementation

3.5.1 Step1:Define the System Specifications

- Define the purpose of the system
  - Input, output
  - System-level
- Define the system's operation
  - Algorithms
  - flowchart
- Define the operational constrains
  - Cost
  - Speed
  - Size
  - Power requirement
  - Reliability
  - Upgrade ability
  - Marketing plans
  - Other considerations

3.5.2 Step2 : Develop a Rough Design for the System

- Objectives
  - Define the control relationships within system
  - Define basic sequential behavior
  - Identify functional units in data path
  - Choose signal names

- Graphical illustrations
  - Block diagram
  - flowchart

3.5.3 Step3: Develop a Detailed Design for the Data Path

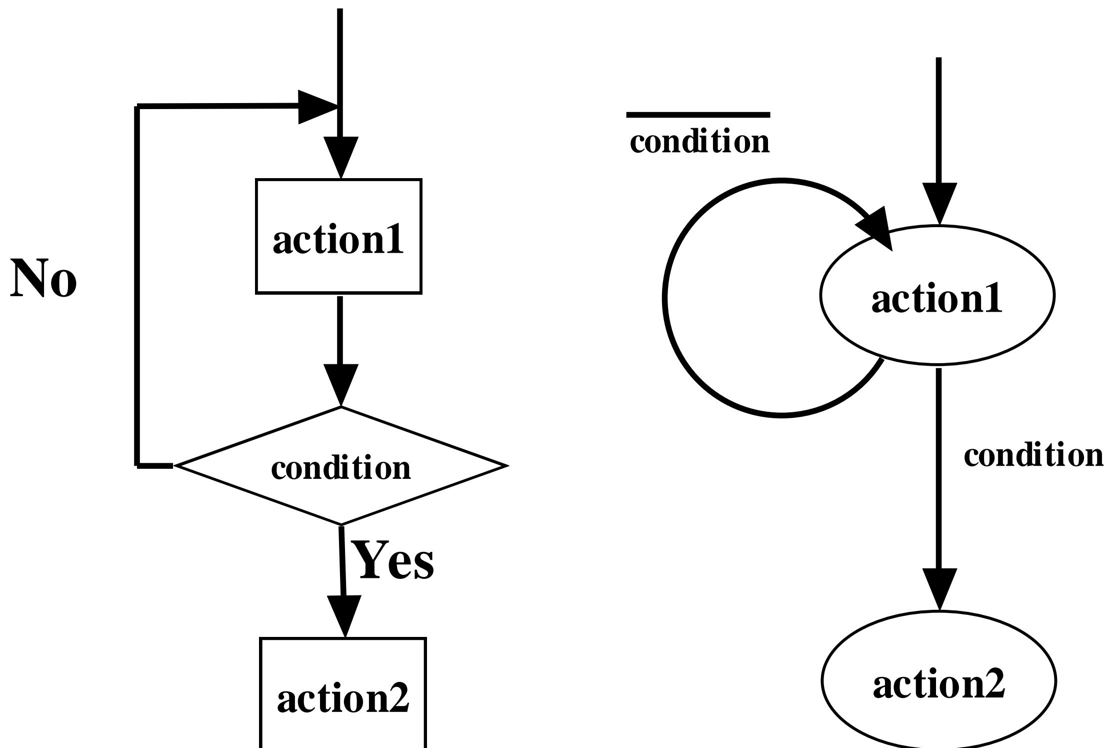
- Objectives
  - Fully define the data path
- Documentation aids
  - Detailed timing diagram
  - Detailed flowchart
  - Detailed functional partial partition

3.5.4 Step4 : Develop a Detailed Specification for the Controller

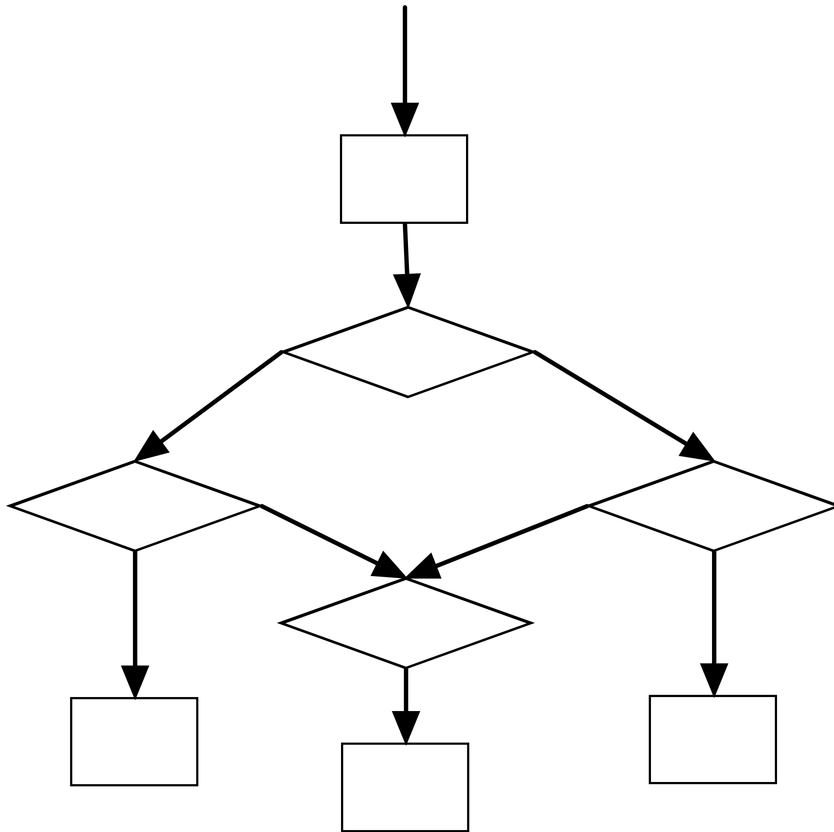
- Objectives:
  - Fully specify the controller behavior
- The operation of the system controller is completed defined by the detailed flowchart
- The controller is a synchronous sequential machine
- It should be expressed as a state diagram.
- Translate the flow diagram to a state diagram

3.5.4 Step4 : Rules for Converting Flow Chart to State Diagram

- Rule1 : Action block in flow diagram→ state in the state diagram

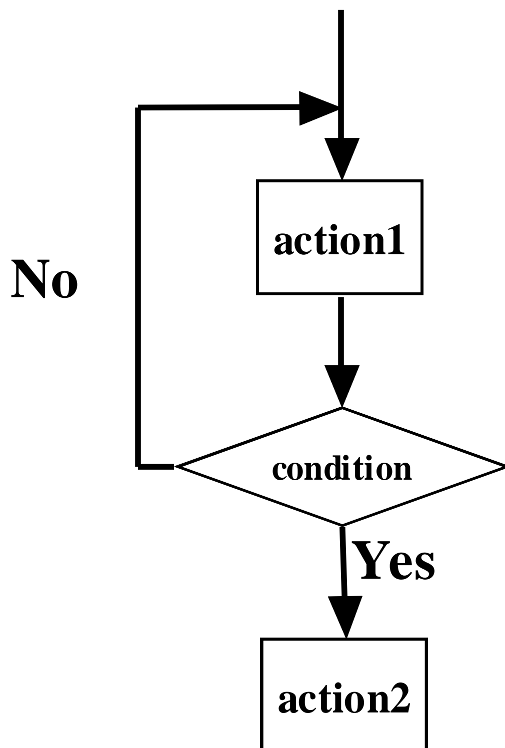


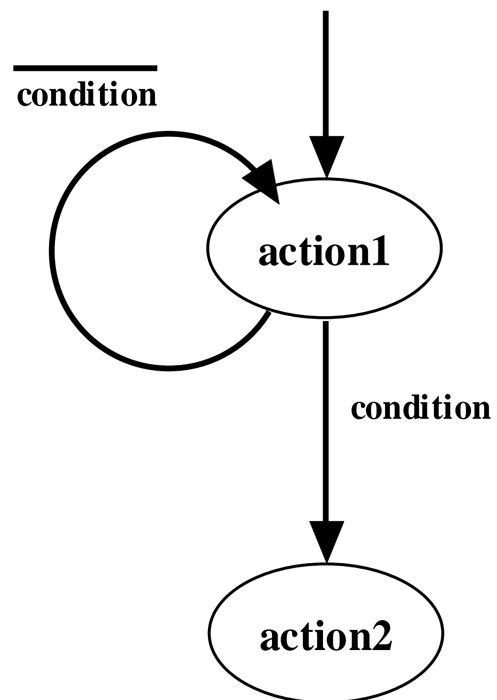
Example



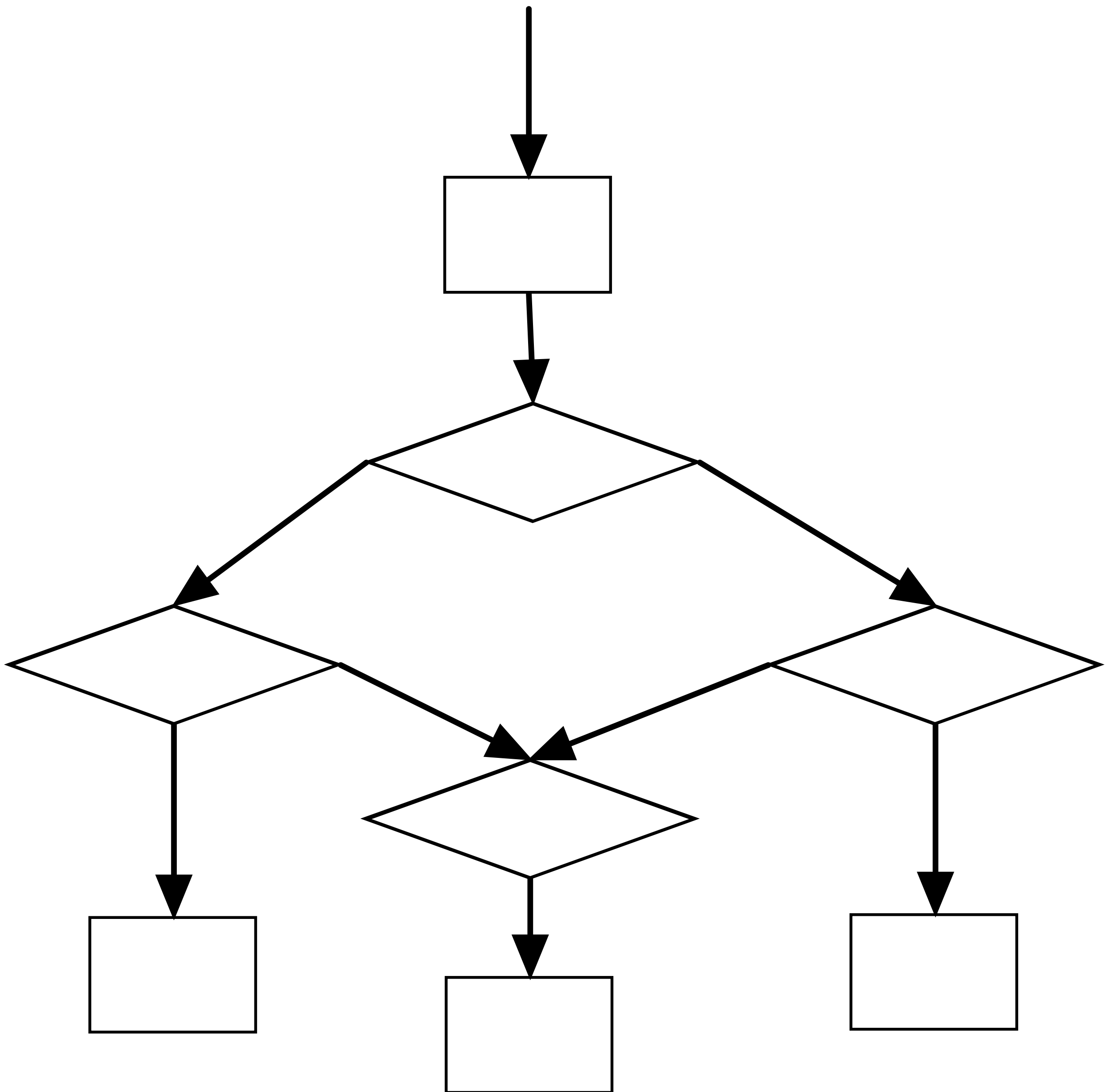
3.5.4 Step4 : Rules for Converting Flow Chart to State Diagram(cont’d)

- Rule2: the branching conditions for a state are derived by tracing through all possible decision paths from the given action block to all possible other action blocks



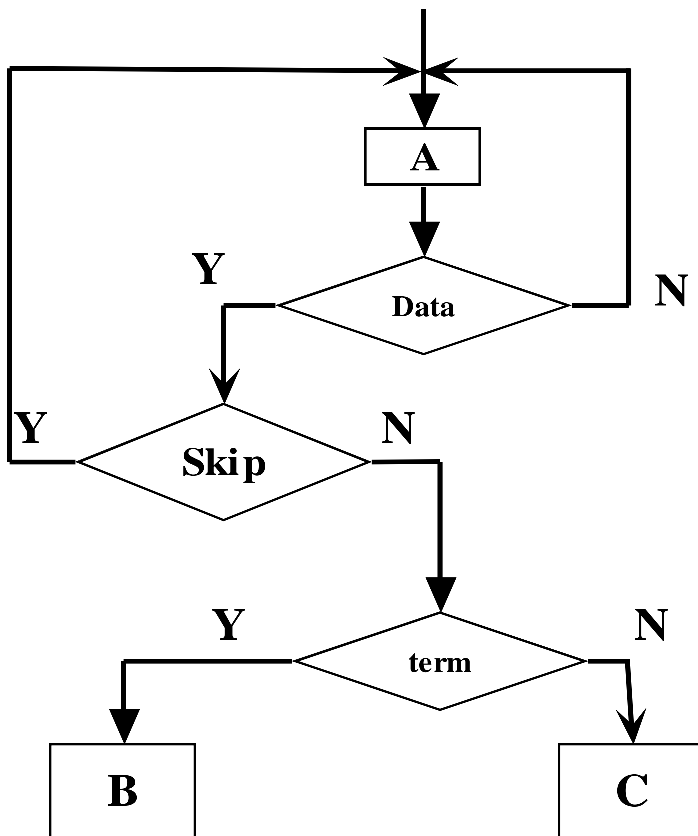


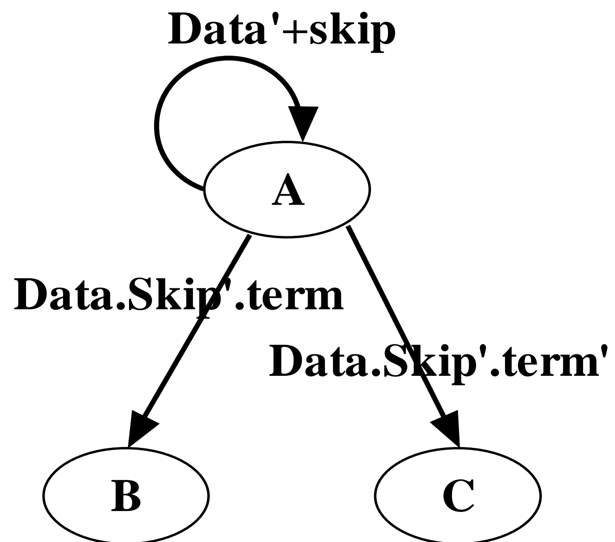
Example



3.5.4 Step4 : Rules for Converting Flow Chart to State Diagram(cont'd)

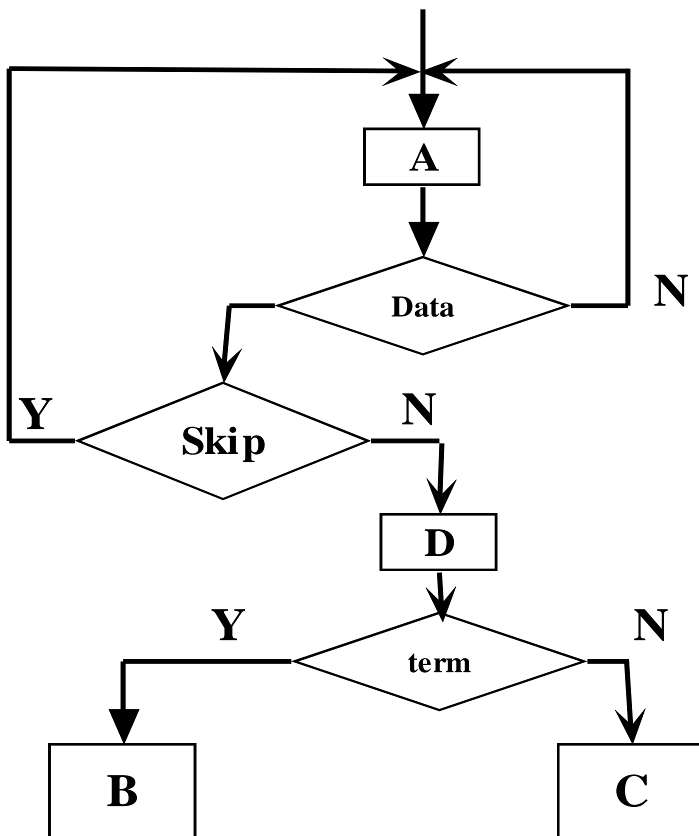
- Rule3: Avoid making branching decisions on more than one asynchronous variable at a time

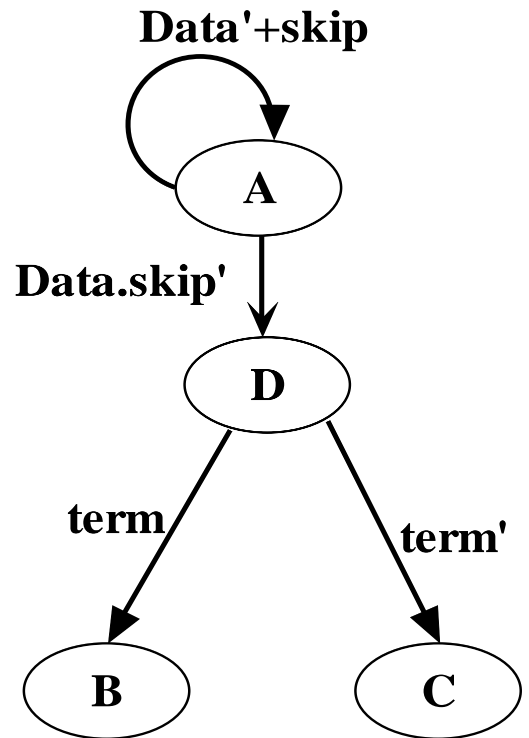




3.5.4 Step4 : Rules for Converting Flow Chart to State Diagram(cont'd)

- Rule3: Avoid making branching decisions on more than one asynchronous variable at a time

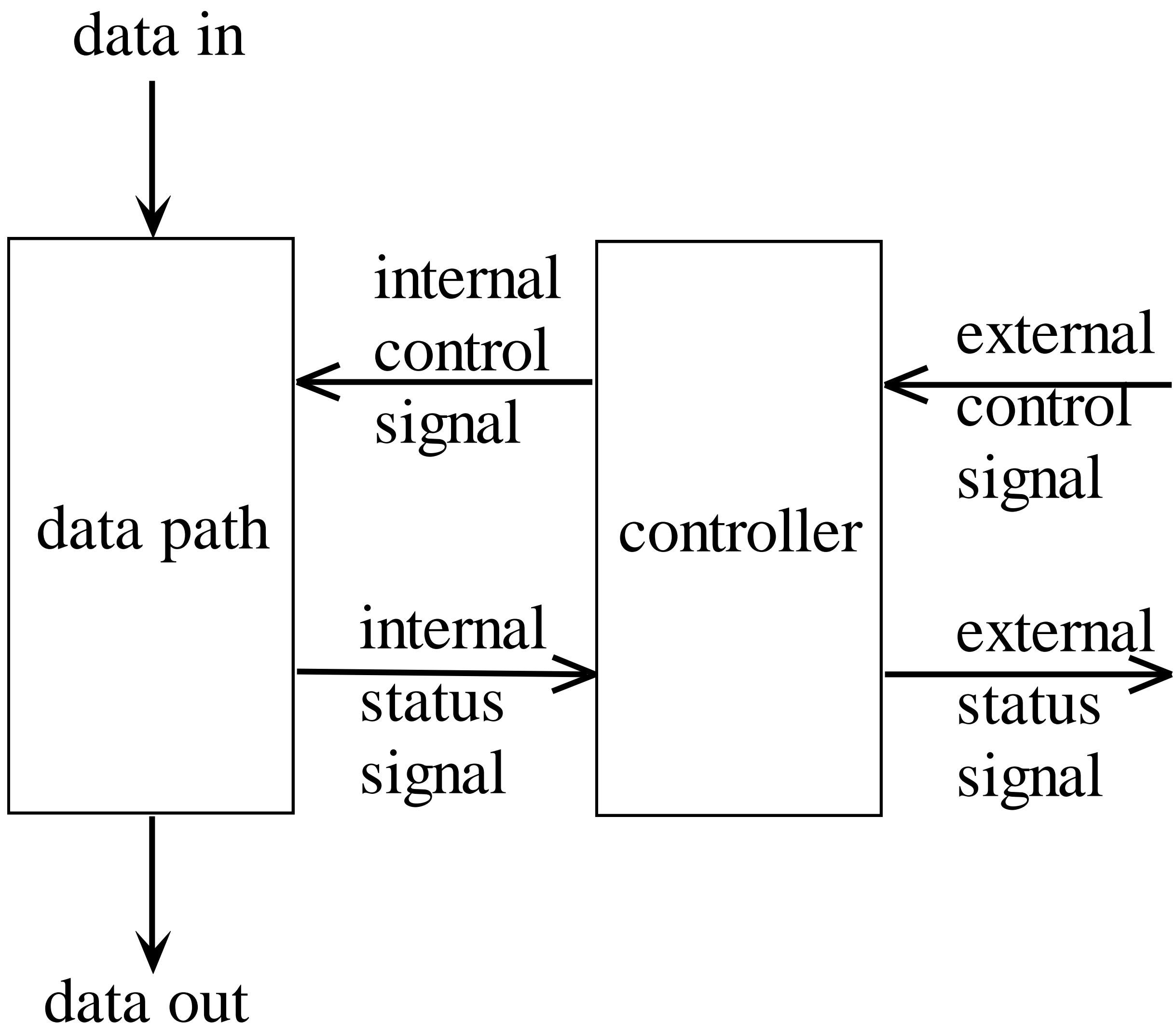




3.5.5 Step5: Complete the Design of the Controller

- Complications:
  - Short/brief input pulses
  - Asynchronous inputs
  - Avoid glitches in the outputs
  - Debug and testability features
- Design steps:
  - Select a controller architecture
    - Deal with synchronous problems
    - Select a clock frequency
  - Find a suitable state assignment
    - Use a state map
  - Implement the next state maps
    - Plot next state maps
  - Design the output decoder
    - Plot output maps (if necessary)
    - Produce an output list

3.5.6 Step6 : Finalize the Design
Controller Data Path Interface



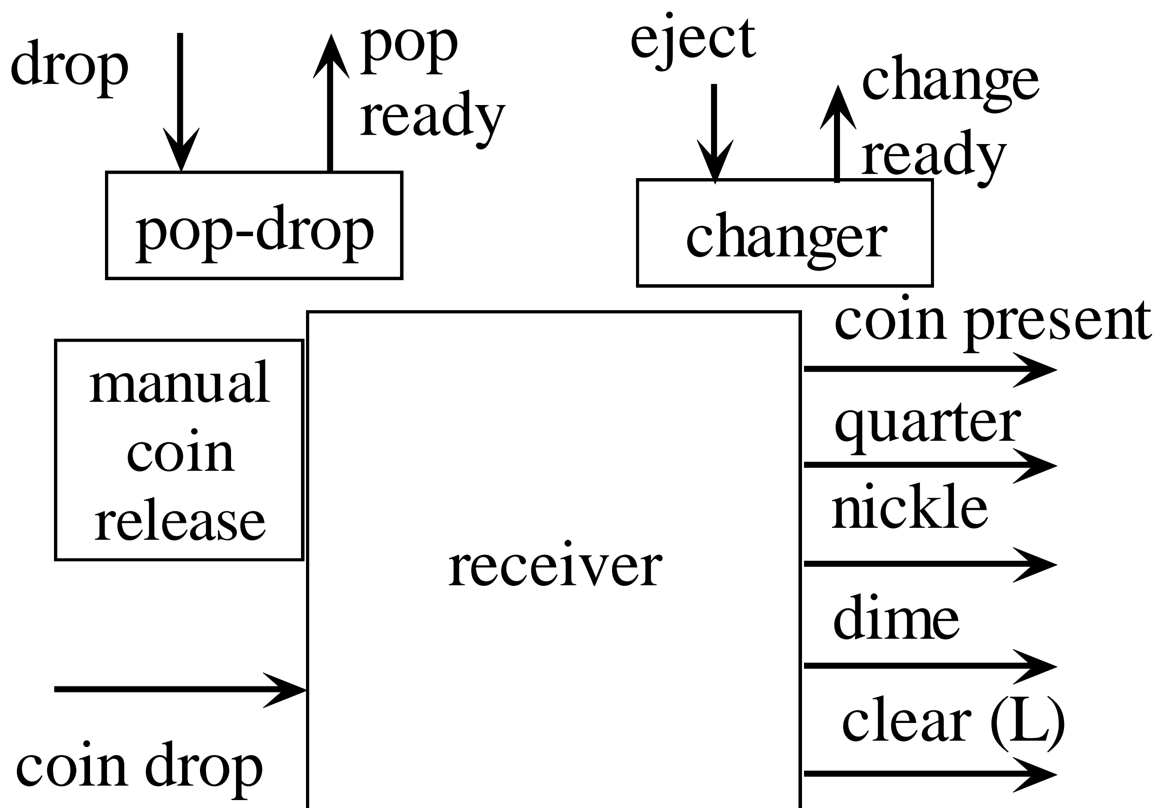
3.5.7 Step7: Simulation and Hardware Implementation

3.6 An Example: pop machine controller

- Requirement:
- Pop machine capable of automatically dispensing soda pop at 75 cents per can and make proper change for coin sequences comprising nickels, dimes, quarters. The new machine use existing inventory including coin receiver, coin changer, and pop drop mechanism. These three given subsystems are to be controlled by a newly designed digital controller.

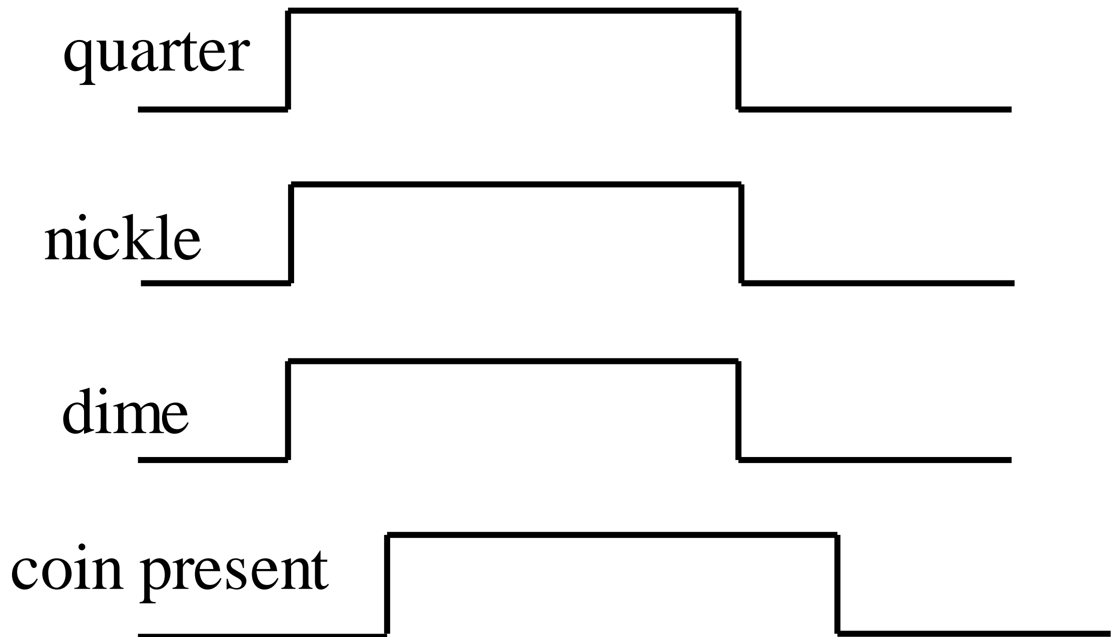
3.6.1 Specs

- Constrains:
  - Must operate out of doors
  - Must operate in an electrically noisy environment
  - Mean time between failures > 2 months
  - Hardware constraints : coin receiver, coin changer and pop-drop mechanism have already been chosen.

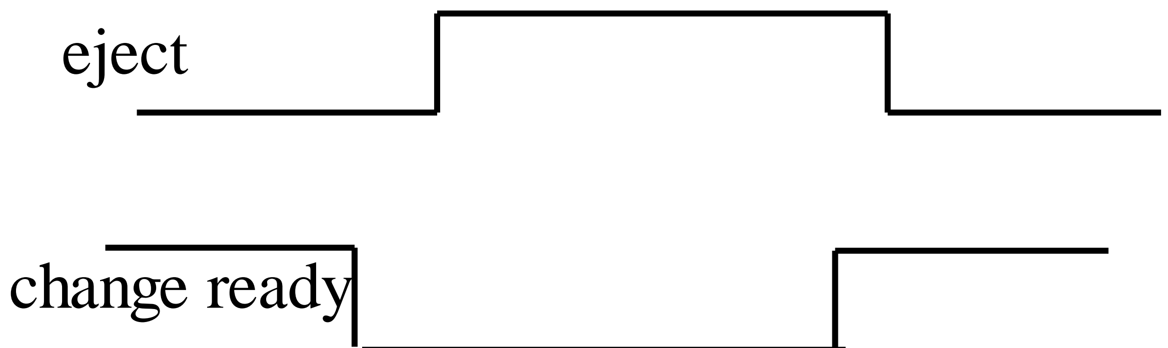


3.6.1 Specs (cont'd)

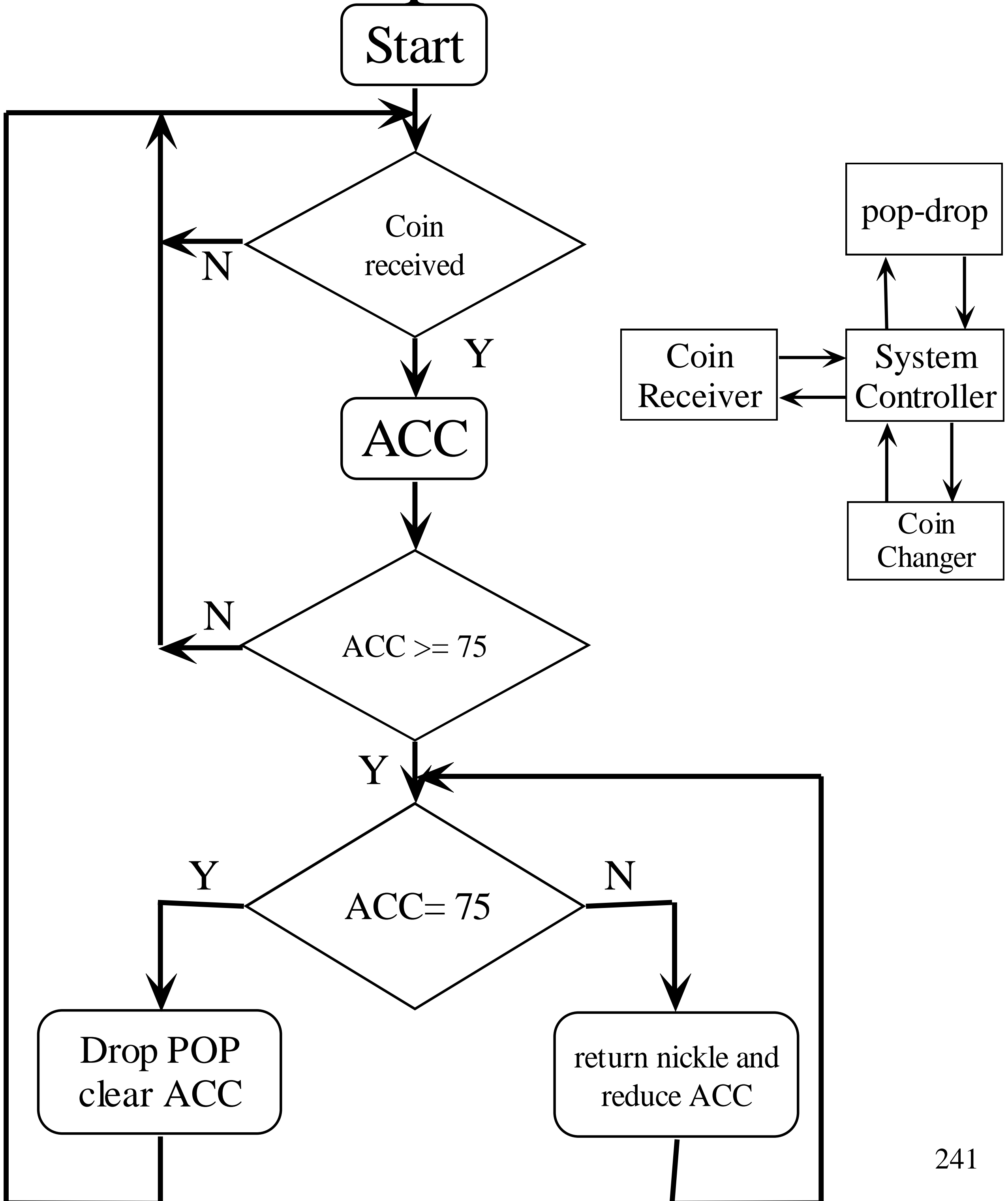
- Time Specs for the coin receiver



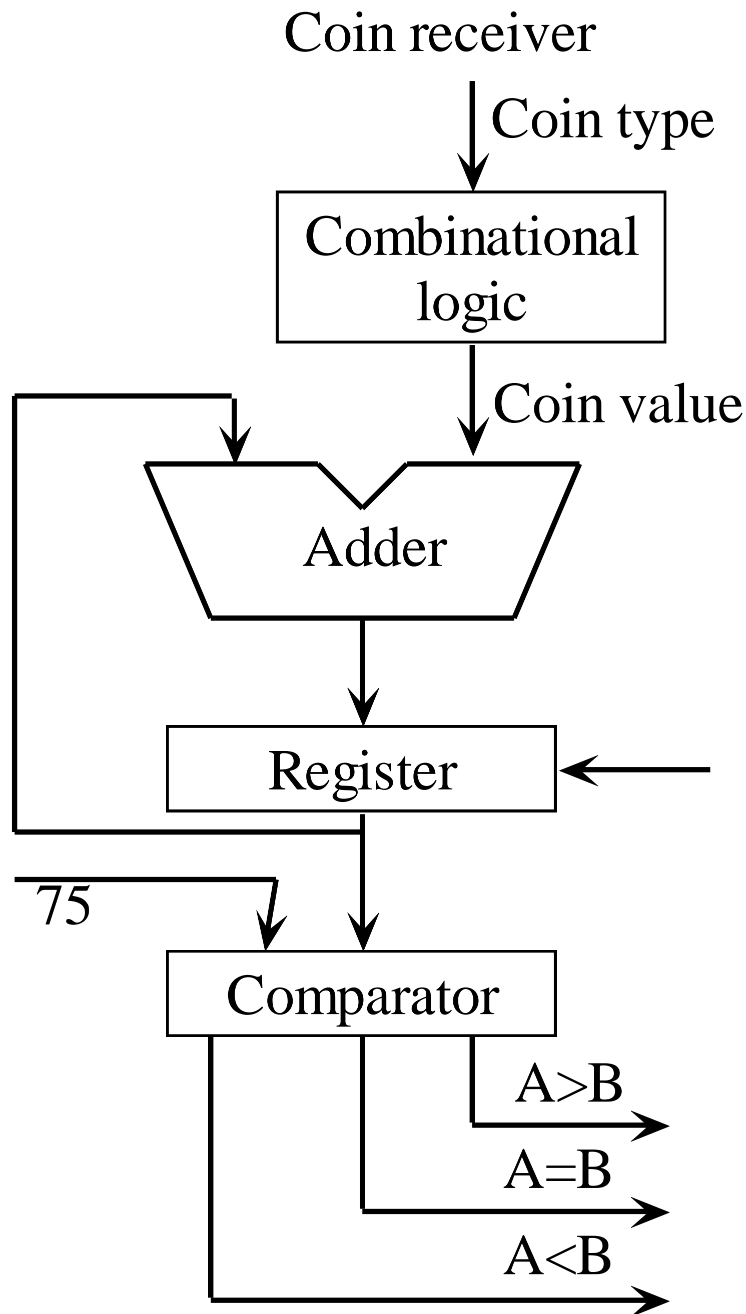
- Time Specs for the coin changer



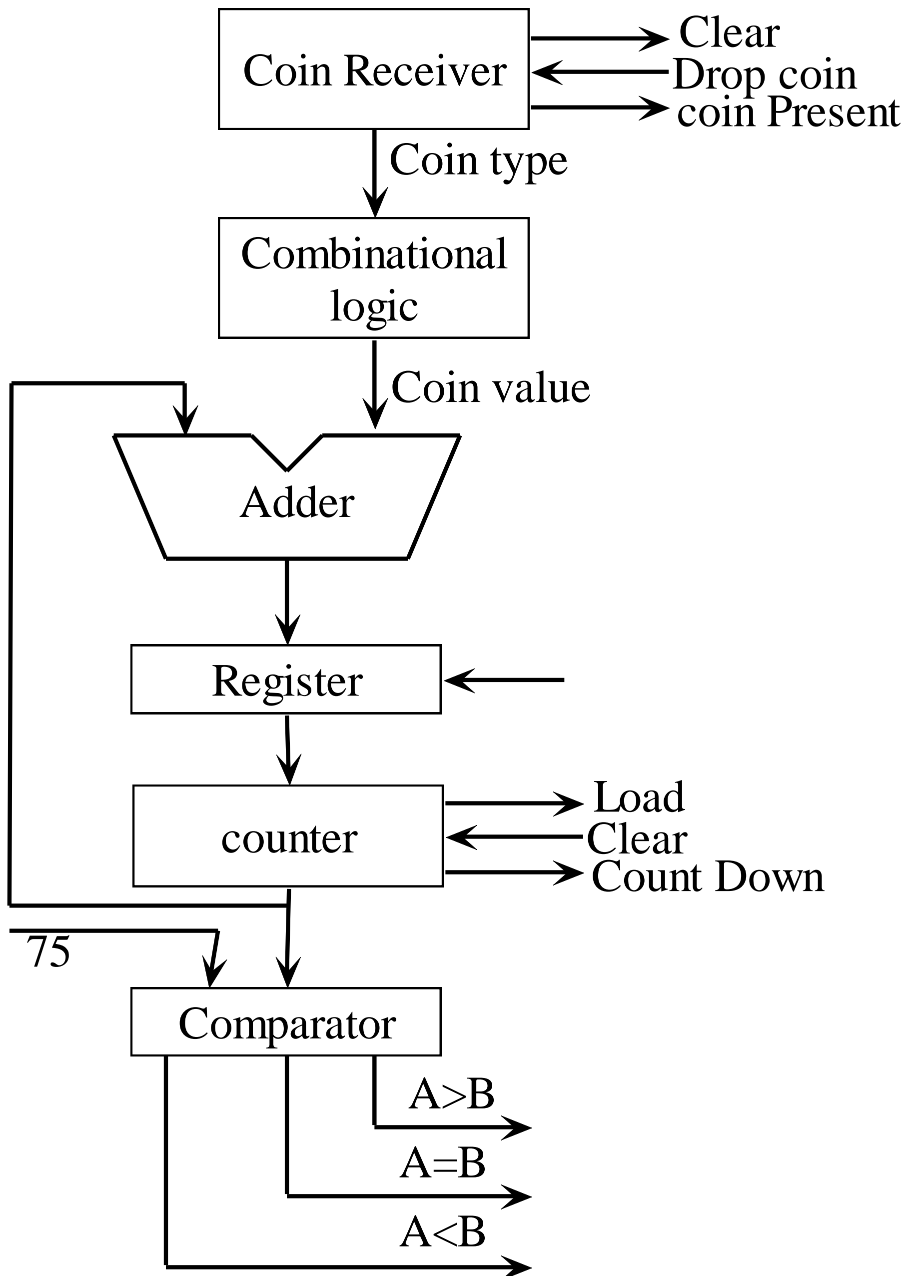
3.6.2 Roughly Define System Operation



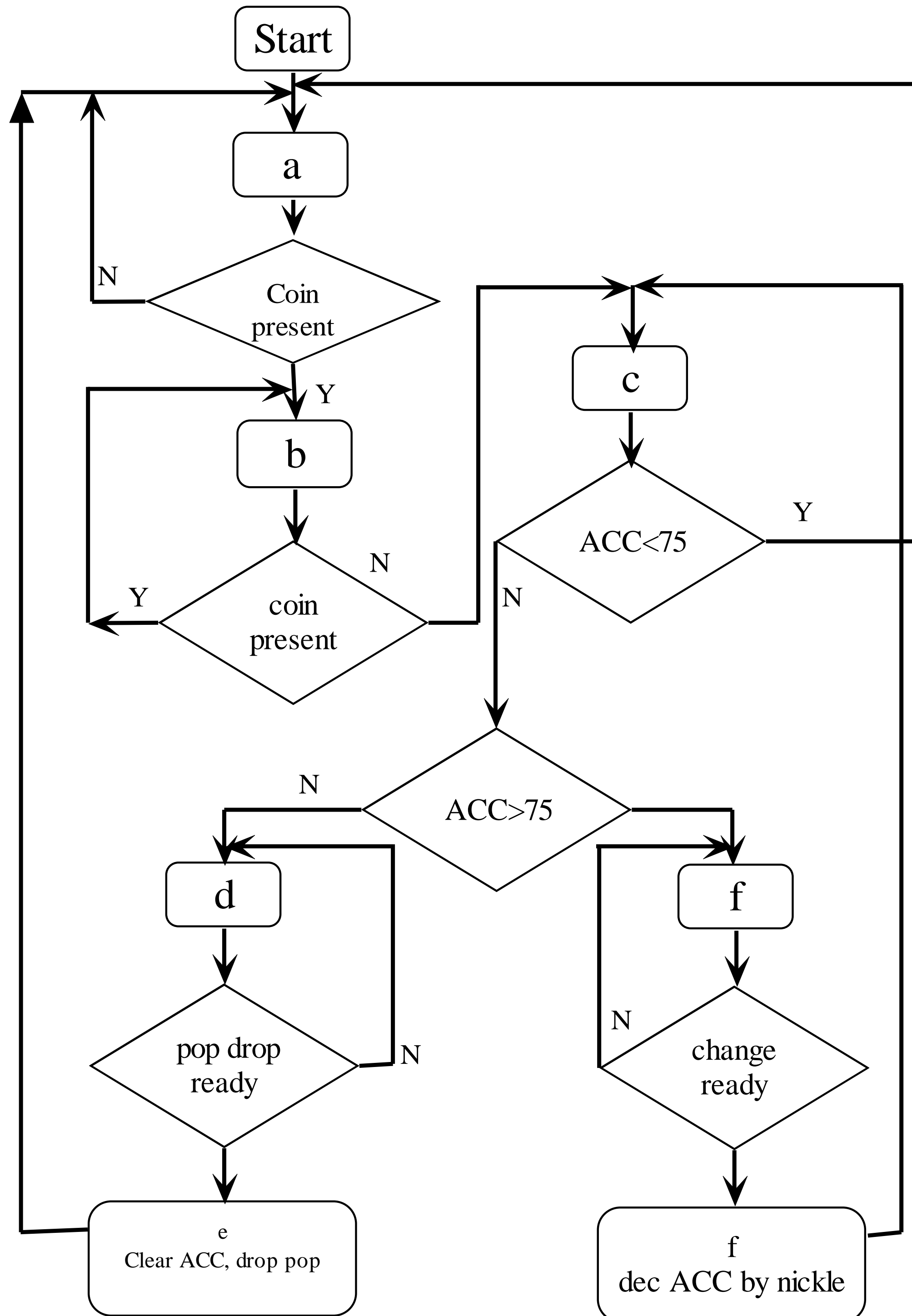
3.6.2 Roughly Block Diagram for the Data Path



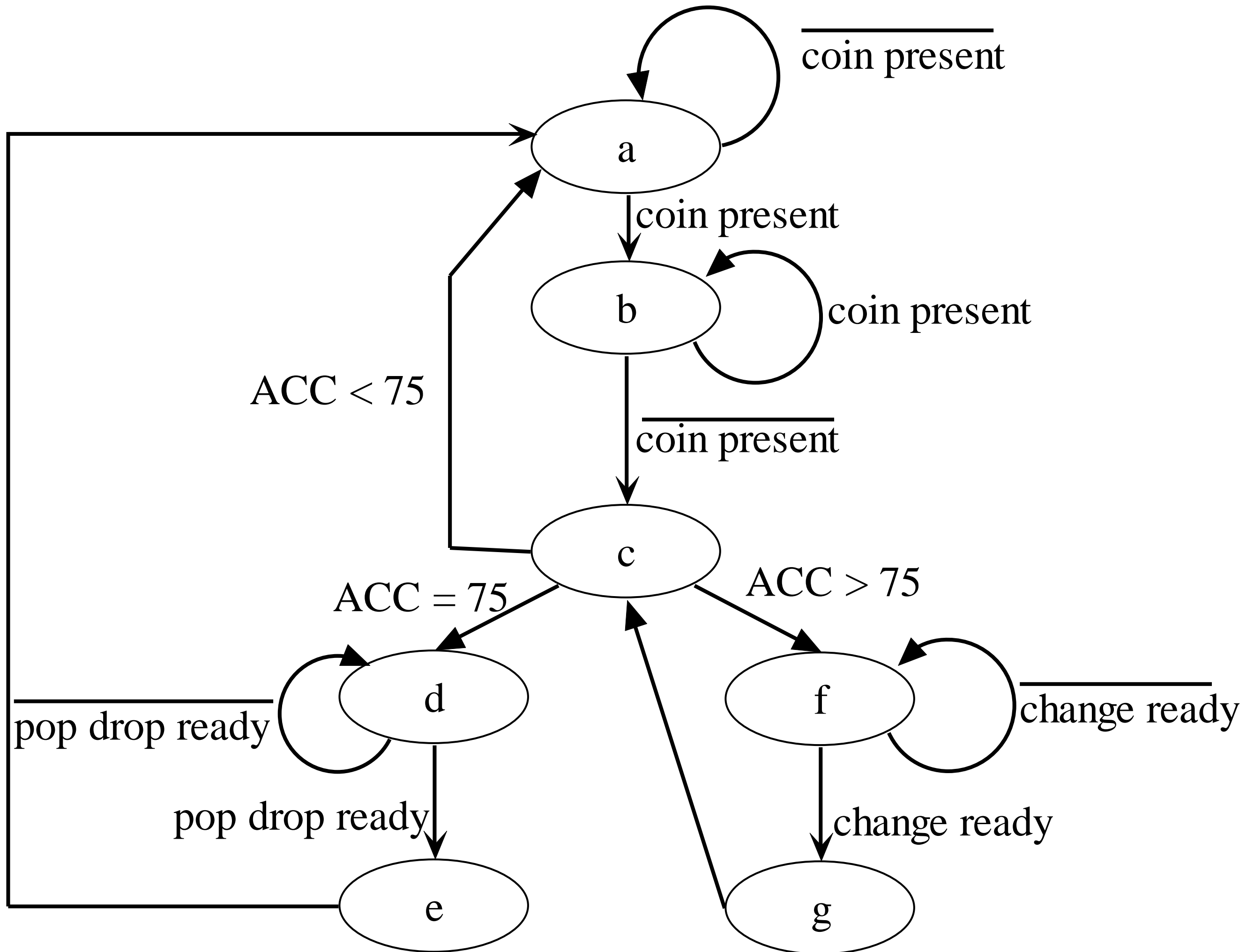
3.6.3 Refined Data Path



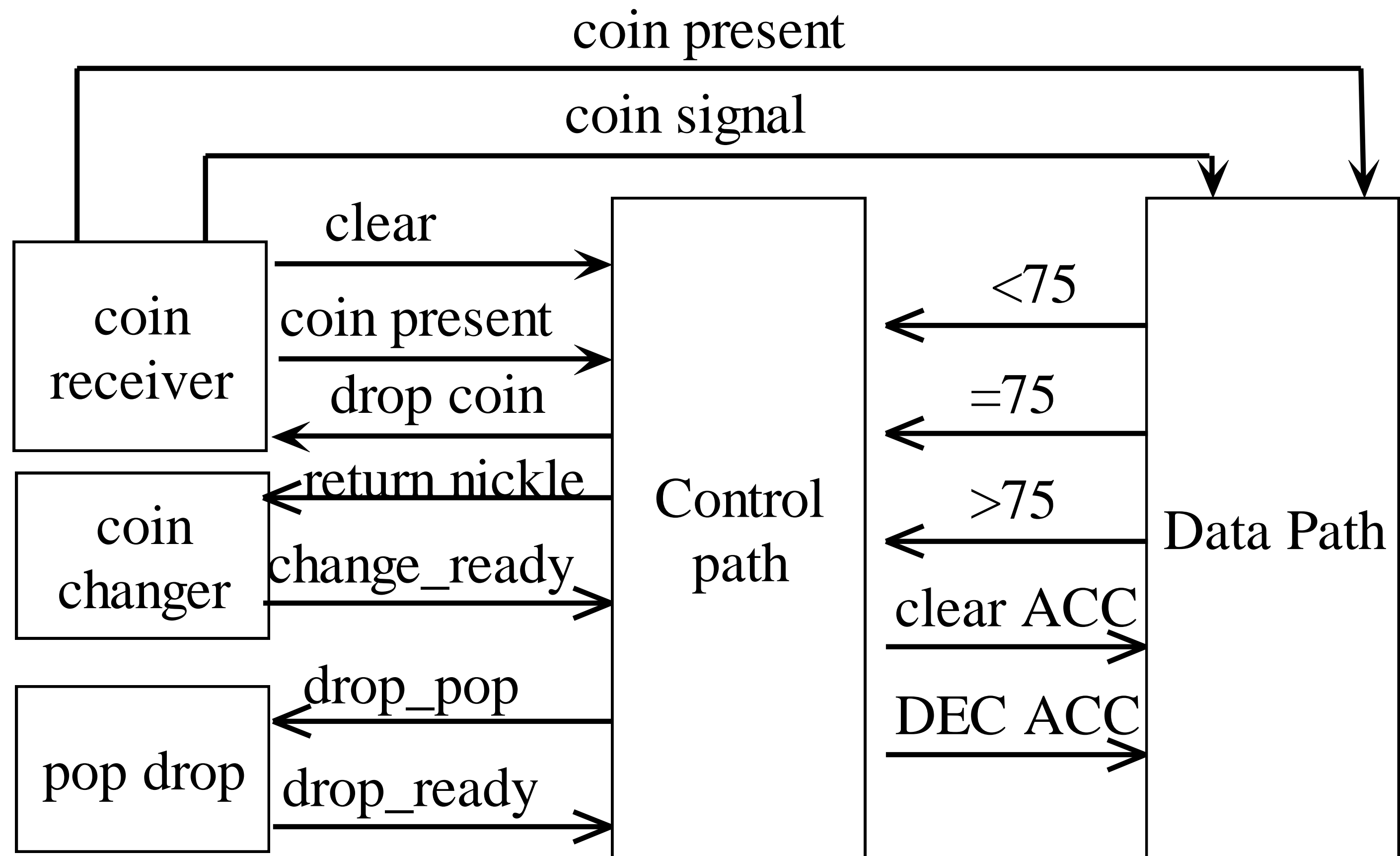
3.6.4 Develop a detailed specification for the controller



3.6.5 Complete the Design of Controller

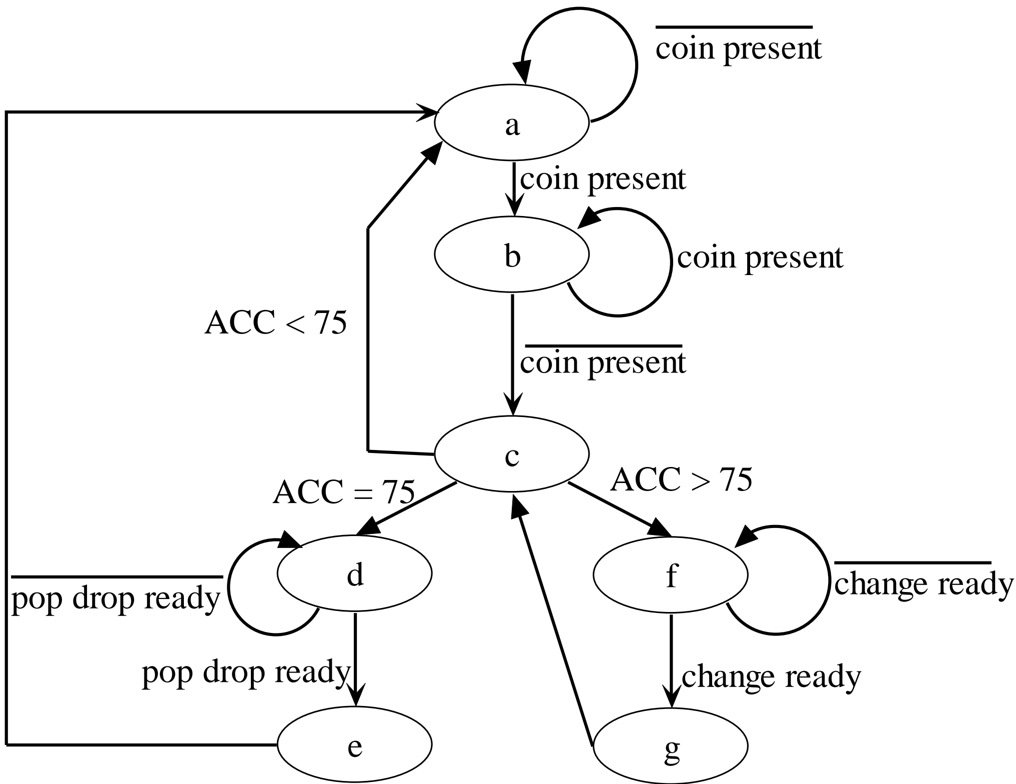


3.6.6 Finalize the Design: Controller Data Path Interface



3.6.7 VHDL Implementation

3.6.7 Digital Design



C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

3.6.7 Digital Design

C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

C \ AB	00	01	11	10
0				
1				

IV. Design of Arithmetic Circuits

4. Design of Arithmetic Circuits

4.1 Introduction to Arithmetic Circuits

4.2 Number Systems and Codes

4.3 Design of Fast Adders

4.4 Design of Fast Multipliers

4.5 Design of Fast divider

4.6 Floating Point Arithmetic

4.1 Introduction to Arithmetic Circuits

- A large proportion of data that is collected is numerical data which must be proceeded arithmetically
- Efficient high-speed hardware implementations of arithmetic operations are desirable
- Some arithmetic circuits are implemented in combinational logic : addition, subtraction, comparing
- Other arithmetic circuits are too bulky in Combinational logic : multiplier Div, Sqt, filtering
- Regular, iterative structure: a recurring theme in the design of arithmetic circuits.

4.2 Number Systems and Codes

4.2.1 Positional Number Systems

4.2.2 Octal and Hexadecimal Numbers

4.2.3 General Positional-Number-System Conversions

4.2.4 Addition and Subtraction of Non-decimal Numbers

4.2.5 Representation of Negative Numbers

4.2.6 Two’s Complement Addition and Subtraction

4.2.7 One’s Complement Addition and Subtraction

4.2.8 Binary Multiplication

4.2.9 Binary Division

4.2.1 Positional Number Systems

- Traditional number system
- A number is represented by a string of digits, each digit position has an associated weight.
  - 1734=1*1000+7*100+3*10+4
  - 585.55=5*100+8*10+5*1+5*0.1+5*0.01
  - 10 is called base or radix
- General : $D=d_{p-1}d_{p-2}\ldots d_1d_0d_{-1}d_{-2}\ldots d_{-n}$
- Binary : $B=b_{p-1}b_{p-2}\ldots b_1b_0b_{-1}b_{-2}\ldots b_{-n}$

$$D = \sum_{i=-n}^{p-1} d_i r^i \quad B = \sum_{i=-n}^{p-1} b_i 2^i$$

- $100.001_2=(?)_{10}$

4.2.2 Octal and Hexadecimal Numbers

- Radix 10 is important
- Radix 2 is important
- Radix 8 & 16 ? (When you are 40?)

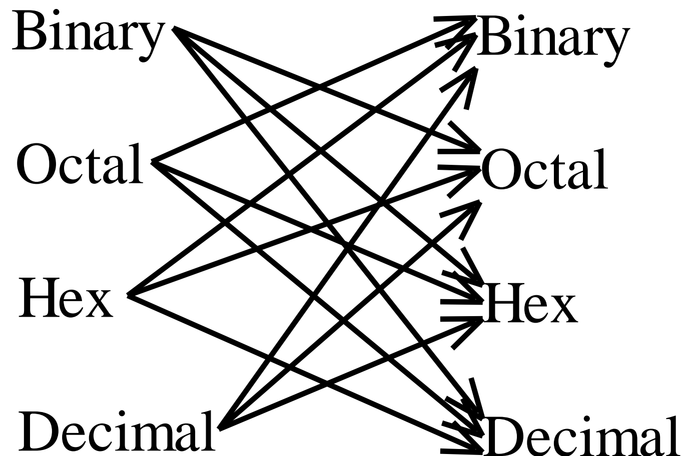
B	Decimal	Octal	Hex
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

4.2.3 General Positional-Number-System Conversions

- Radix r to decimal conversion

$$D = \sum_{i=-n}^{p-1} d_i r^i$$

  - The value of the number can be found by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix –10 arithmetic
  - 436.5_8=
- $D=d_{p-1}d_{p-2}\ldots d_1 d_0 d_{-1} d_{-2}\ldots d_{-n}$
- =(…((dp-1)*r+dp-2)*r+…)r+d1



4.2.4 Addition and Subtraction of Nondecimal Numbers

- To add two binary numbers X and Y, we add together the least significant bits with an initial carry (cin) of 0, producing carry (cout) and sum (s) bits. We continue processing bits from right to left, adding the carry out of each column into the next column's sum. (Examples)
- Binary subtraction is performed similarly, using borrow instead of carries between steps, and producing a difference bit d. (Examples)

4.2.5 Representation of Negative Numbers

1) Signed Magnitude Representation

- The signed magnitude system is applied to binary numbers by using an extra bit position to represent the sign.
- To build a adder and subtractor: disadvantage and advantage.

2) Complement Number Systems

- Radix complement system, the complement of an n-digit number is obtained by subtracting it from r^n
- Example (10’s complement, 9’s complement)
- Two’s complements
  - MSB serves as the sign bit
  - Examples

4.2.5 Representation of Negative Numbers

Decimal	Two's complement	One's complement	Signed magnitude
-8	1000		
-7	1001	1000	1111
-6	1010	1001	1110
-5	1011	1010	1101
-4	1100	1011	1100
-3	1101	1100	1011
-2	1110	1101	1010
-1	1111	1110	1001
0	0000	1111 or 0000	1000 or 0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111

4.2.6 Two's Complement Addition and Subtraction

- Addition
  - The results will always be the correct sum as long as the range of the number system is not exceeded.
  - Overflow detector :
  - $V = a_n b_n r_n' + a_n' b_n' r_n$ (example)
- Subtraction
  - Negate the subtrahend by taking its two's complement, and then add it to the minuend using the normal rules for addition
  - Example

4.2.7 One’s Complement Addition and Subtraction

- Addition
  - If we start at 1000 (-7) and count up, we obtain each successive one one’s complement number by adding 1 to the previous one, except at the transition from 1111 (0) to 0001 (1)
  - Suggestion: perform a standard binary addition, but add an extra 1 whenever we count past 1111
  - Examples
- Subtraction
  - Complement all bits of the subtrahend and proceed as in addition

4.2.8 Binary Multiplication

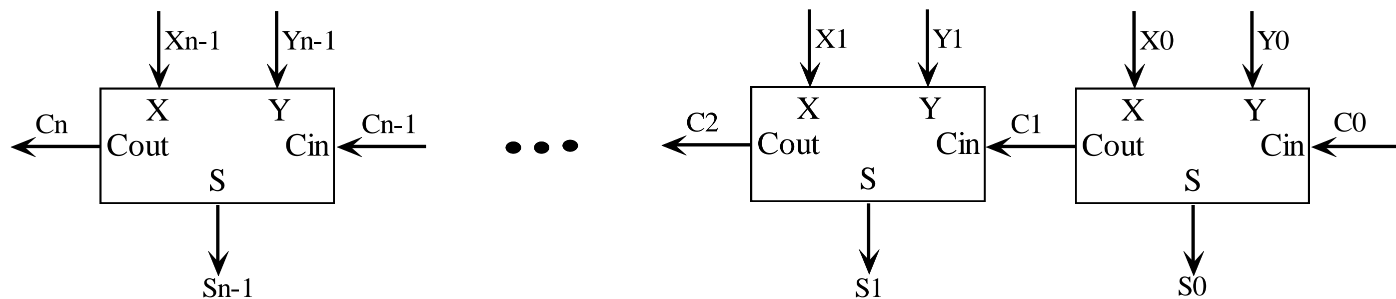
- Unsigned multiplication
  - Shift and Add Multiplication
  - Examples
- Signed multiplication
- Two's complement multiplication
  - Example

4.2.9 Binary Division

- Unsigned division
  - Shift and subtract
- Signed division
  - Using unsigned division
  - Make the quotient positive if the operands had the same sign, negative if they has different signs
  - The reminder should be given the same sign as the division

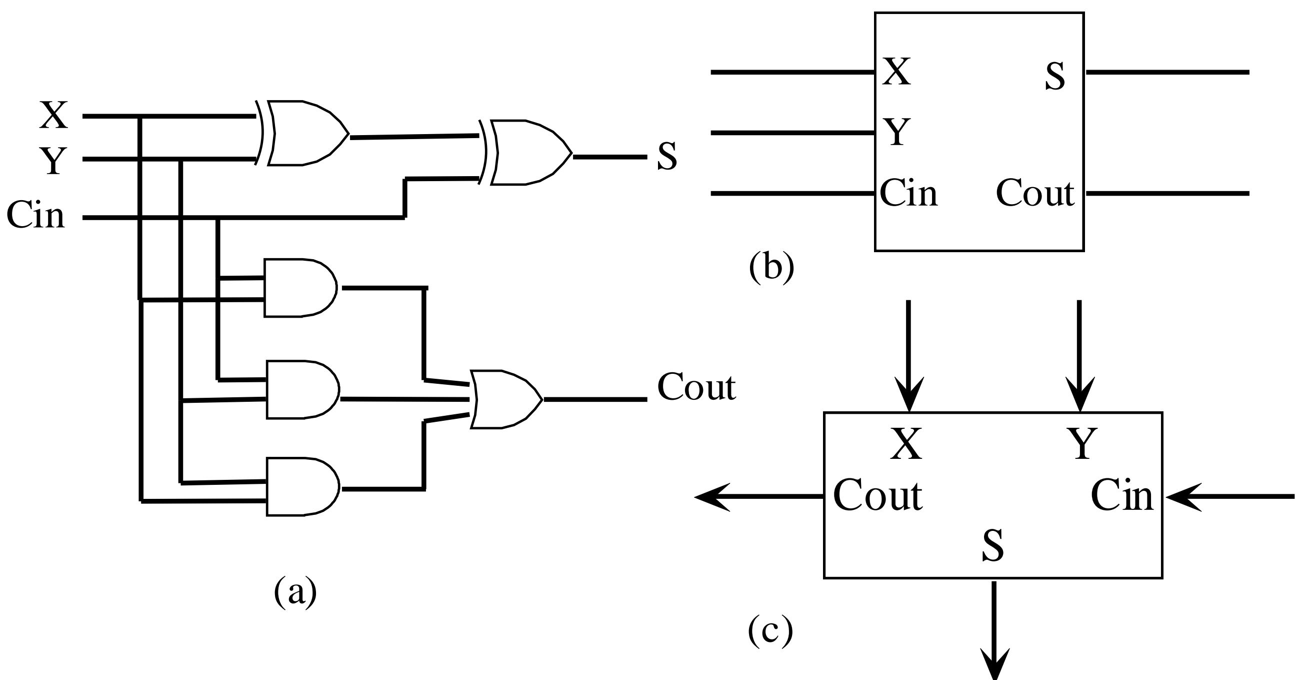
4.3 Design of Fast Adders

- Ripple carry adders
  - Decompose addition into bit-wise operations
  - Use the same circuit design for each bit
  - Get simple regular structure
- Carry signal ripples from LSB to MSB



Full Adder Circuit

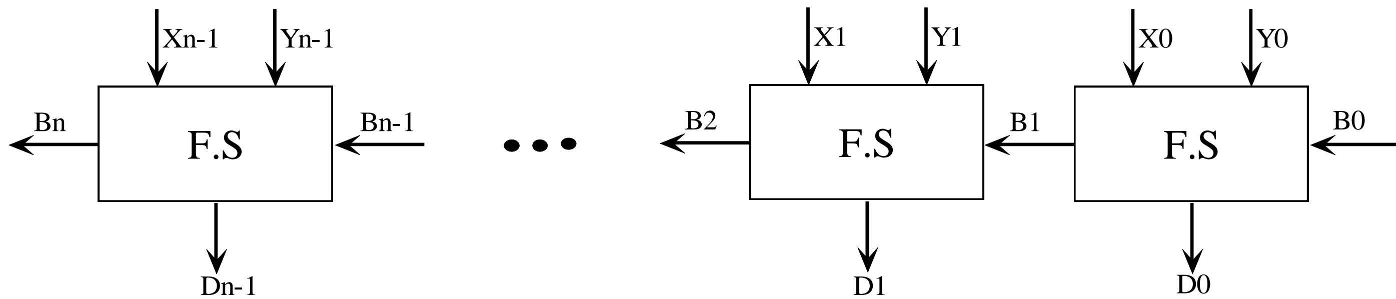
- C_{i+1}=Ci*ai+Ci*bi+ai*bi $S_i = a_i \oplus b_i \oplus c_i$
- Gate delay from C_i to C_{i+1}, Ci to Si?
- Critical path analysis:
  - Critical path: the signal path that limits the speed of the circuit
- Delay for n bit ripple carry adder ?
- Advantages: simple,iterative.
- Disadvantages: delays grows proportion to n

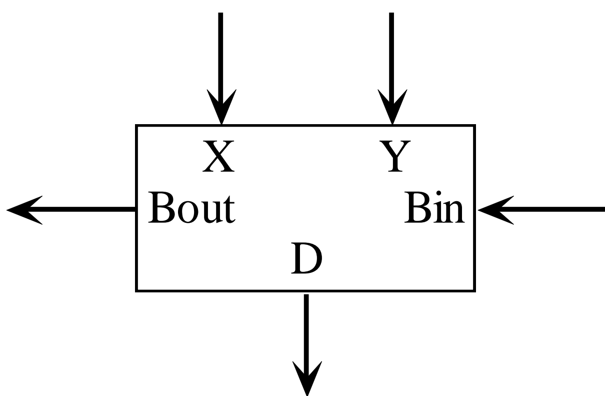


Full Subtractor Circuit

Gate delay from B_i to B_{i+1}, Bi to Di?

- Delay for n bit ripple carry adder ?
- Advantages & Disadvantages: same as ripple carry adder





Bi \ XiYi	00	01	11	10
0				
1				

Di

Bi \ XiYi	00	01	11	10
0				
1				

Bi

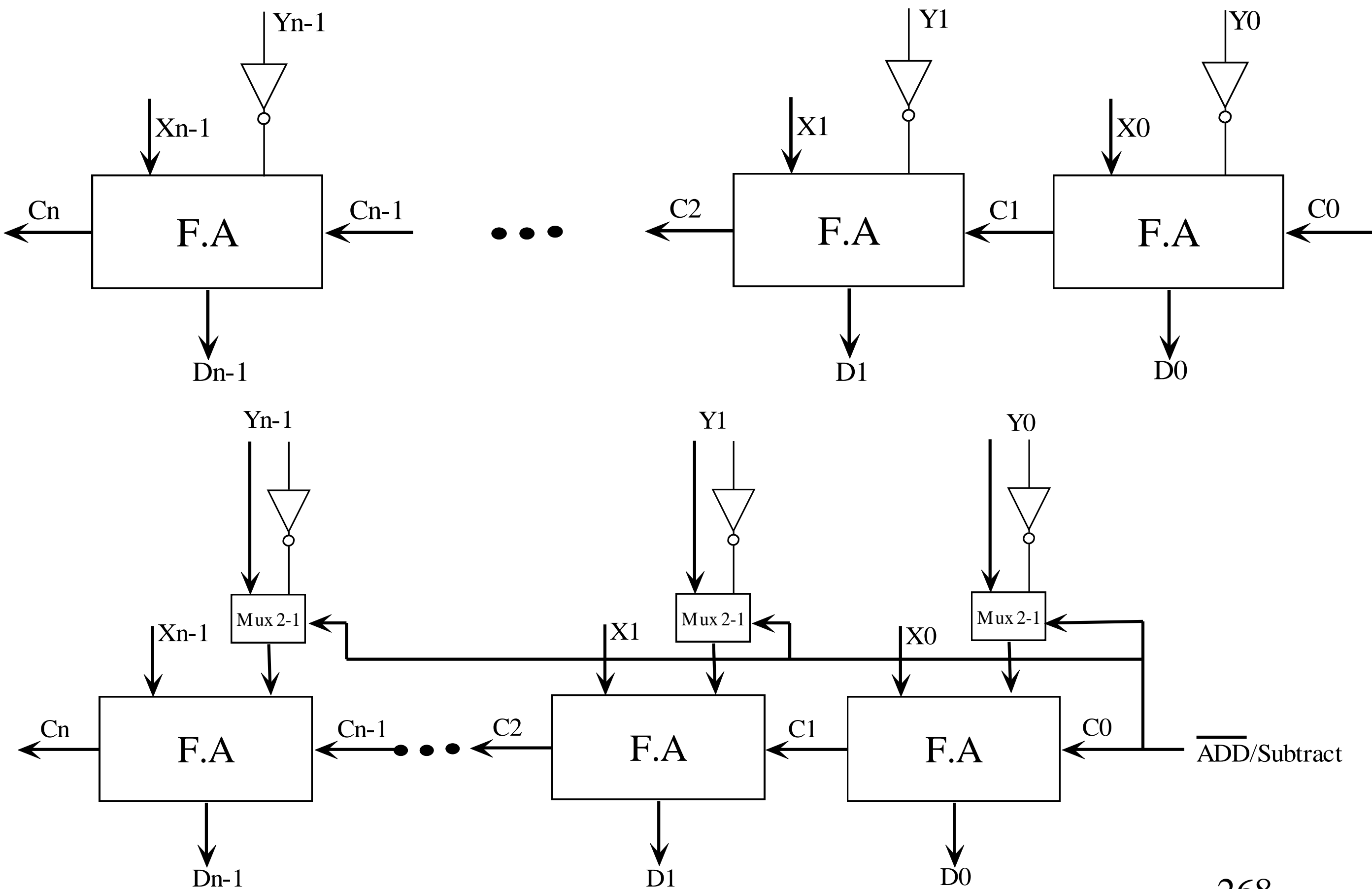
2’s Complement subtractors

Gate delay ?

Fast adder solution:

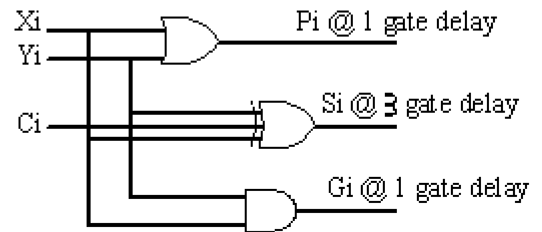
#1 PLA delay?

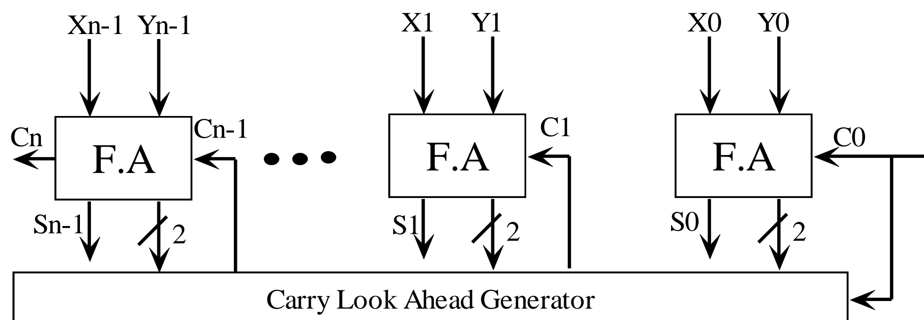
#2 Find a Compromise Circuit Design



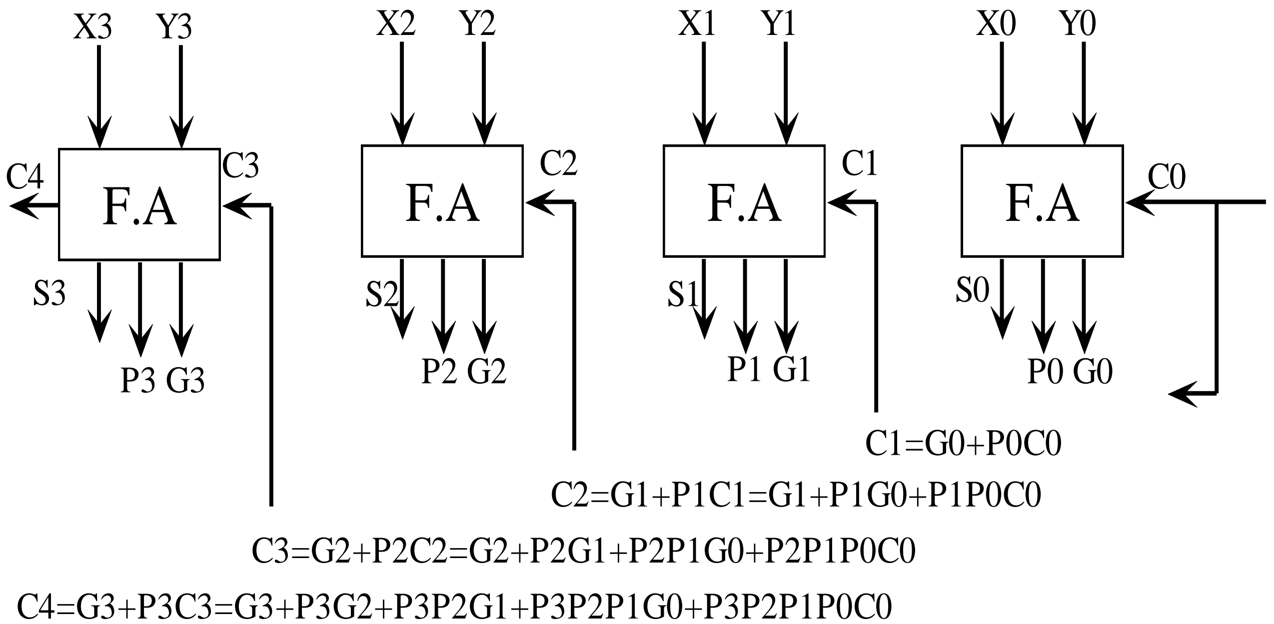
Carry Look Ahead Logic

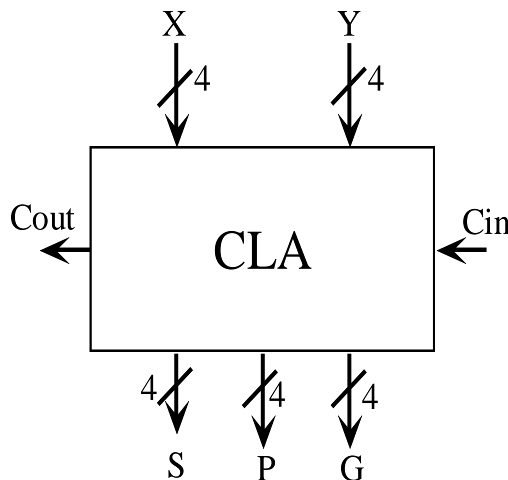
- In the 4-bit "ripple" adder, the carry out of each stage, Ci+1, is expressed as a function of Xi, Y, and Ci. The basic idea of carry look ahead logic is to express each Ci in terms of Xi, Xi-1,…X0, Yi, Yi-1, …Y0, and C0 directly. Delay from C0 to Cn-1?
- Recall C1=X0Y0+X0C0+Y0C0
- Define: generate Gi=XiYi
- propagate Pi=Xi+Yi
- C1= X0Y0+X0C0+Y0C0
- =X0Y0+(X0+Y0)C0=G0+P0C0
- C2=G1+P1C1=G1+P1G0+P0P1C0
- C3=G2+P2C2=G2+P2(G1+P1G0+P0P1C0)
- =G2+P2G1+P2P1G0+P2P1P0C0
- …



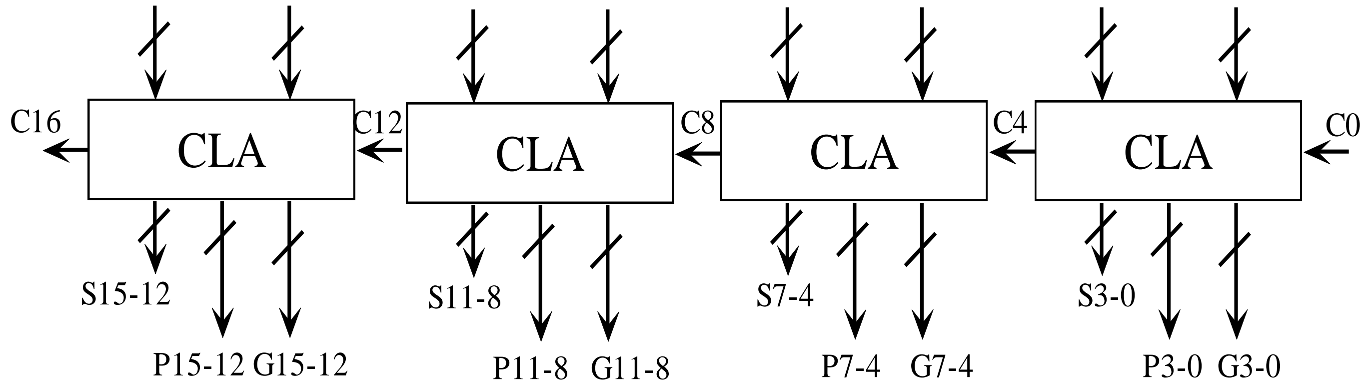


4-bit Carry Look- Ahead Adder (CLA)





Cascade 4-bit CLAs

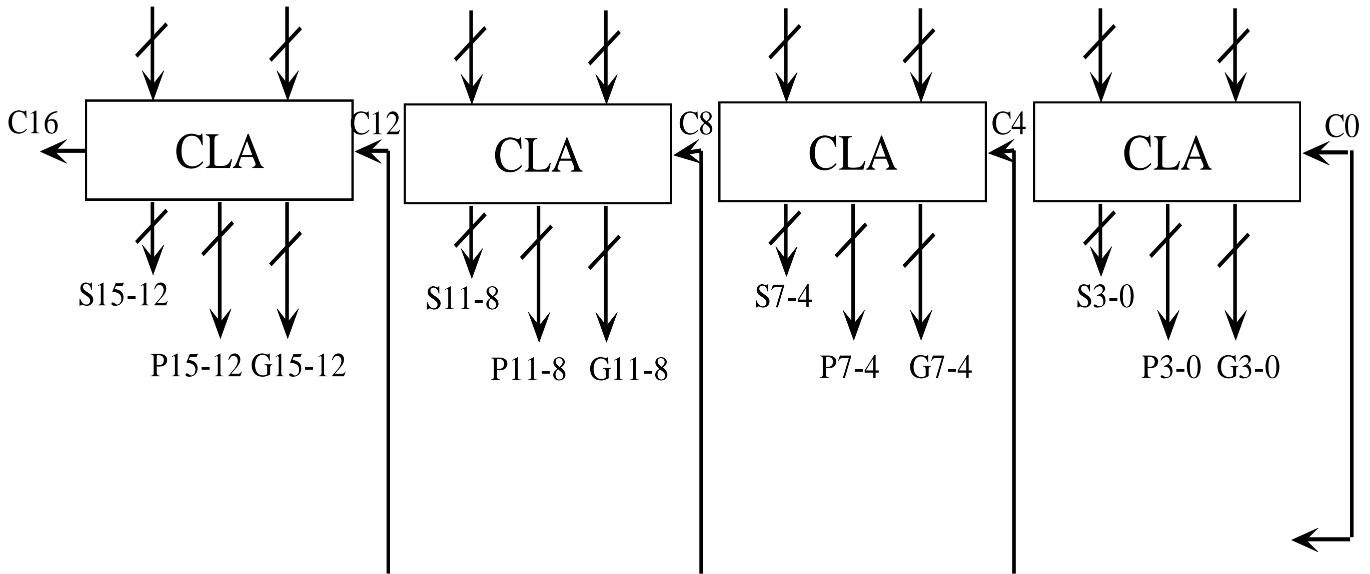


- Time Analysis:
- C0 → C4, C4→C8, C8→C12, C12 → C15, C15 →S15

Worst case delay:

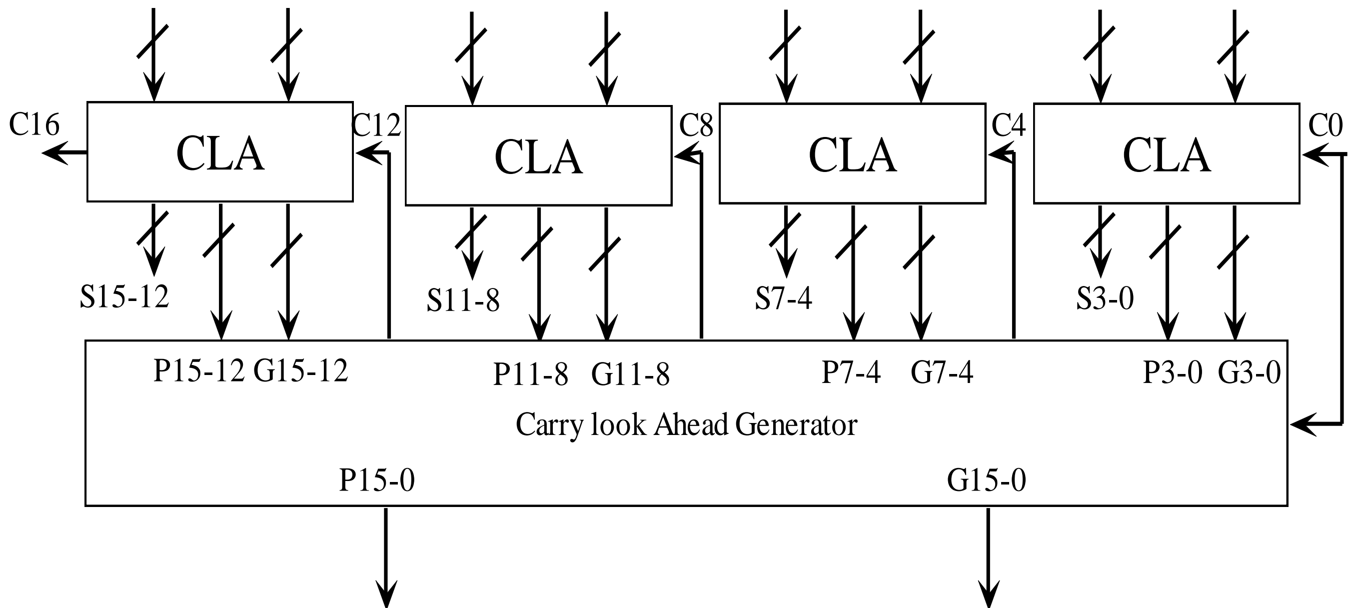
- 16-bit ripple carry adder worst case delay:
- General 4-bit CLAs delay:

Cascade look Ahead Generator



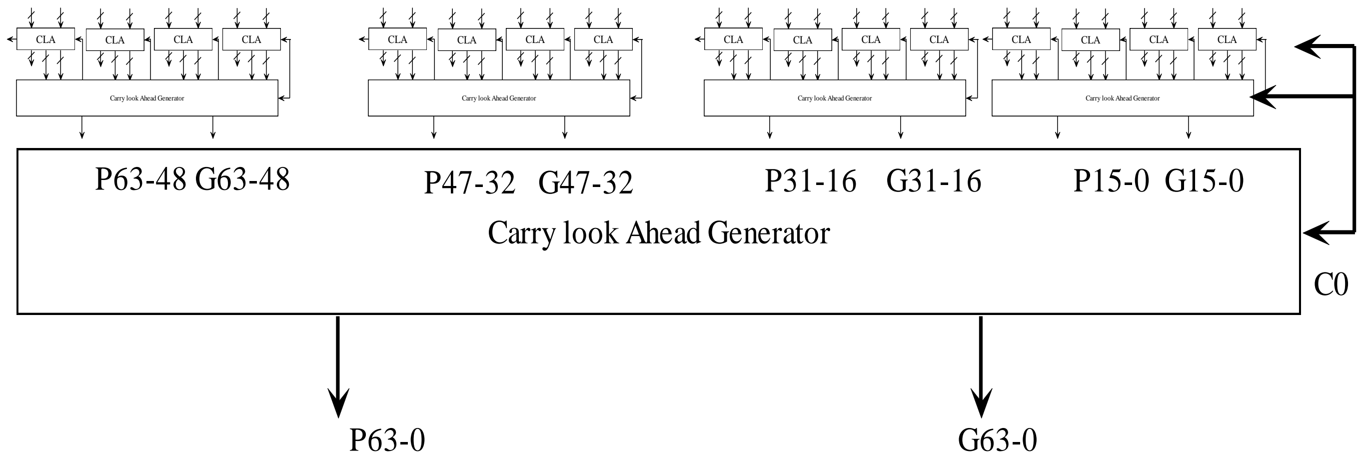
- G3-0=G3+P3G2+P3P2G1+P3P2P1G0
- P3-0=P3P2P1P0
- G7-4=G7+P7G6+P7P6G5+P7P6P5G4
- P7-4=P7P6P5P4
- G11-8=G11+P11G10+P11P10G9+P11P10P9G8
- P11-8=P11P10P9P8
- G15-12=G15+P15G14+P15P14G13+P15P14P13G12
- P3-0=P15P14P13P12

Fast 16 bit CLA and CLG



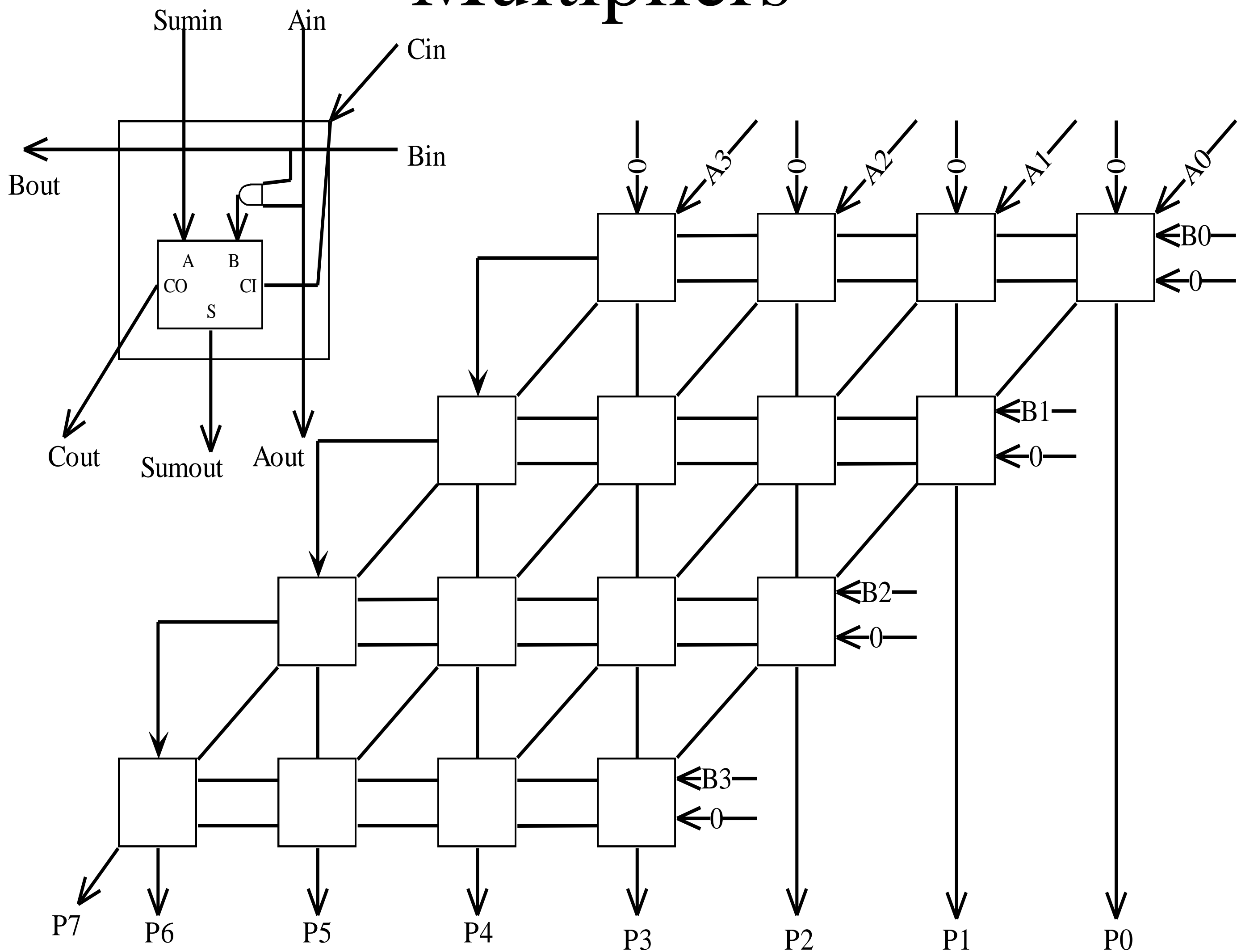
- G15-0=G15-12+P15-12G11-8+P15-12P1-8G7-4+P15-12P11-8P7-4G3-0
- P15-0=P15-12P11-8P7-4P3-0
- Time Analysis:
- C0 → G3-0, G3-0→C12, C12→C15
- C15→S15
- Worst case delay:
- Cascade CLA delay:
- Ripple carry adder delay:

Fast 64 Bit Adder Based on CLA and CLG



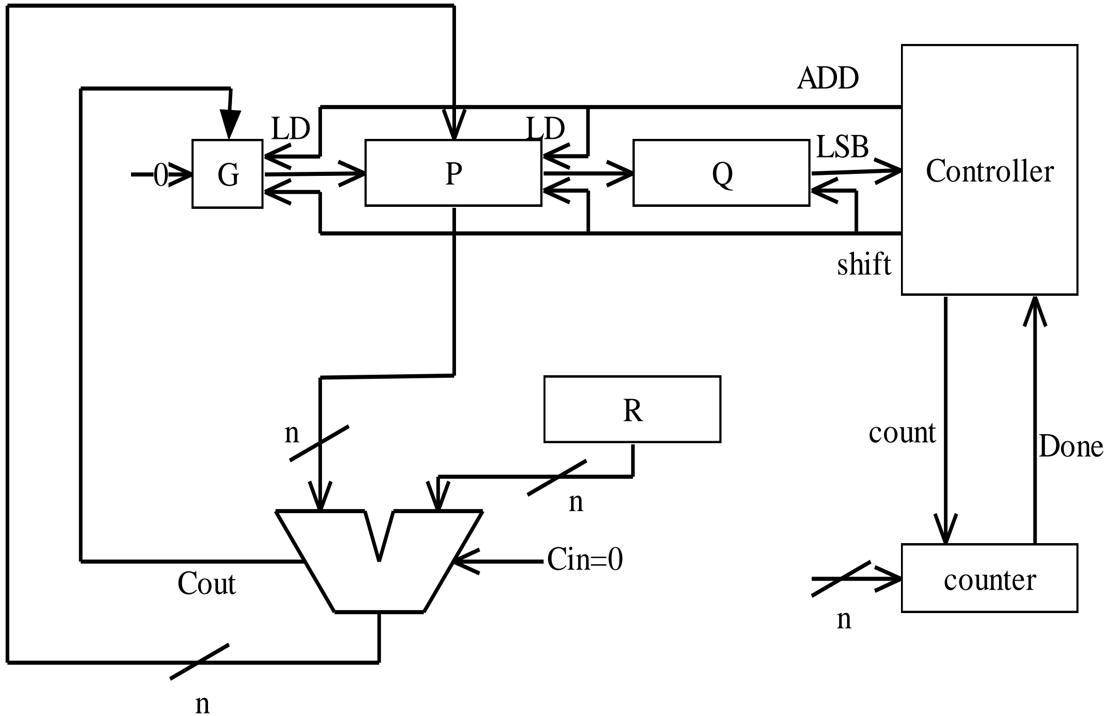
- Time Analysis:
- C0 → G3-0, G3-0→G15-0, G15→C48, C48→C60, C60→C63, C63→S63
- Worst case delay:
- Cascade CLA delay:
- Ripple carry adder delay:
- 128 bit CLA +CLG delay:
- General :

4.4.1 Combinational Multipliers



- Time Analysis:
- Cin → Cout, Cin→Sumout,
  - Worst case delay: 2(n-1)+3(n-1)+2(n-1)+3

4.4.2 Sequential Add/Shift Unsigned Multiplier



R				G	P				Q				operation
0	0	0	0	0	**0**	**0**	**0**	**0**	0	0	0	0	
1	0	1	1	0	**0**	**0**	**0**	**0**	1	1	0	1	initialize
1	0	1	1	0	**1**	**0**	**1**	**1**	1	1	0	1	add
1	0	1	1	0	**0**	**1**	**0**	**1**	1	1	1	0	shift
1	0	1	1	0	**0**	**0**	**1**	**0**	1	1	1	1	shift
1	0	1	1	0	**1**	**1**	**0**	**1**	1	1	1	1	add
1	0	1	1	0	**0**	**1**	**1**	**0**	1	1	1	1	shift
1	0	1	1	1	**0**	**0**	**0**	**1**	1	1	1	1	add
1	0	1	1	0	**1**	**0**	**0**	**0**	1	1	1	1	shift

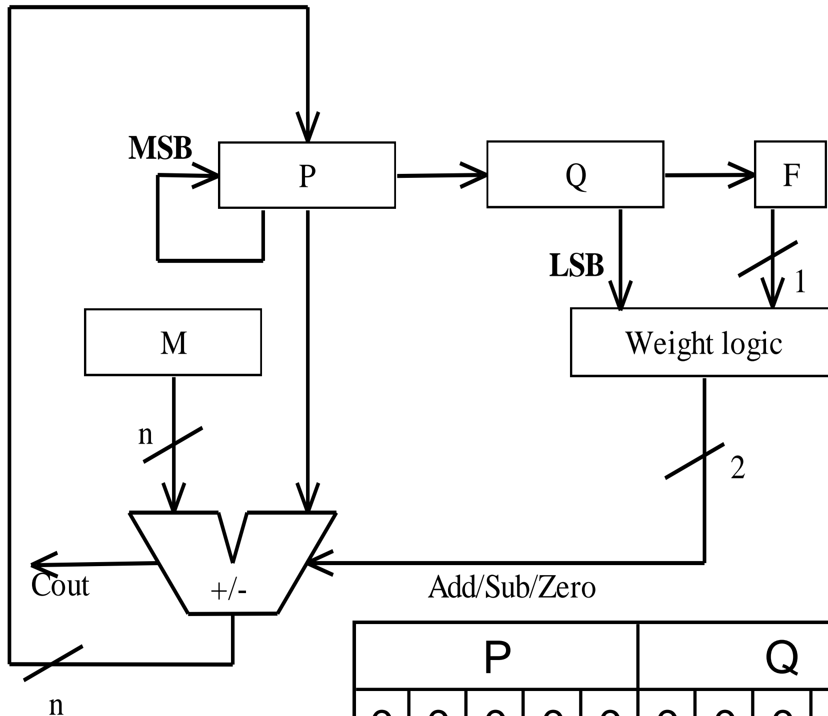
4.4.3Booth Re-coded Multipliers

- Used in the booth multiplication algorithm
- Consider adjacent bits as follows:
- 0 0 0 1 1 0 1 1
- 0 0 +1 1 -1 0 0 1
- Examples: (MSB is sign bit)
- 01010 100110
- 10011 0110101

4.4.3Booth Re-coded Multipliers (cont'd)

- The Booth Multiplication Algorithm
- Works for arbitrary combinations of positive and negative in 2's complement
  - Step1 recode the (positive or negative) multiplier
  - Step2: carry out the multiplication taking into account the sign of each nonzero multiplier bit
  - Examples: 45*30, and 13*(-16)
  - Example (-44)*(-19)

4.4.3Booth Multipliers (implementation)



00 → +0
01 → +M
10 → -M
11 → +0

P					Q					F	Operation
0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	1	1	0	0	1	0	initialize
0	1	1	0	0	1	1	0	0	1	0	-M
0	0	1	1	0	0	1	1	0	0	1	shift right
1	1	0	1	0	0	1	1	0	0	1	+M
1	1	1	0	1	0	0	1	1	0	0	shift right
1	1	1	0	1	0	0	1	1	0	0	+0
1	1	1	1	0	1	0	0	1	1	0	shift right
0	1	0	1	0	1	0	0	1	1	0	-M
0	0	1	0	1	0	1	0	0	1	1	shift right
0	0	1	0	1	0	1	0	0	1	1	+0
0	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	1	shift right

-12
-7

M= 10100

M*=01100

Q=11001

M
Q

4.4.3Fast Multipliers

- The booth algorithm can save effort only when …
- Worst case ?
- Fast multiplication: a modified booth algorithm that guarantees that no more than 0.5 n partial products are required
- Idea: consider two booth bits at a time.

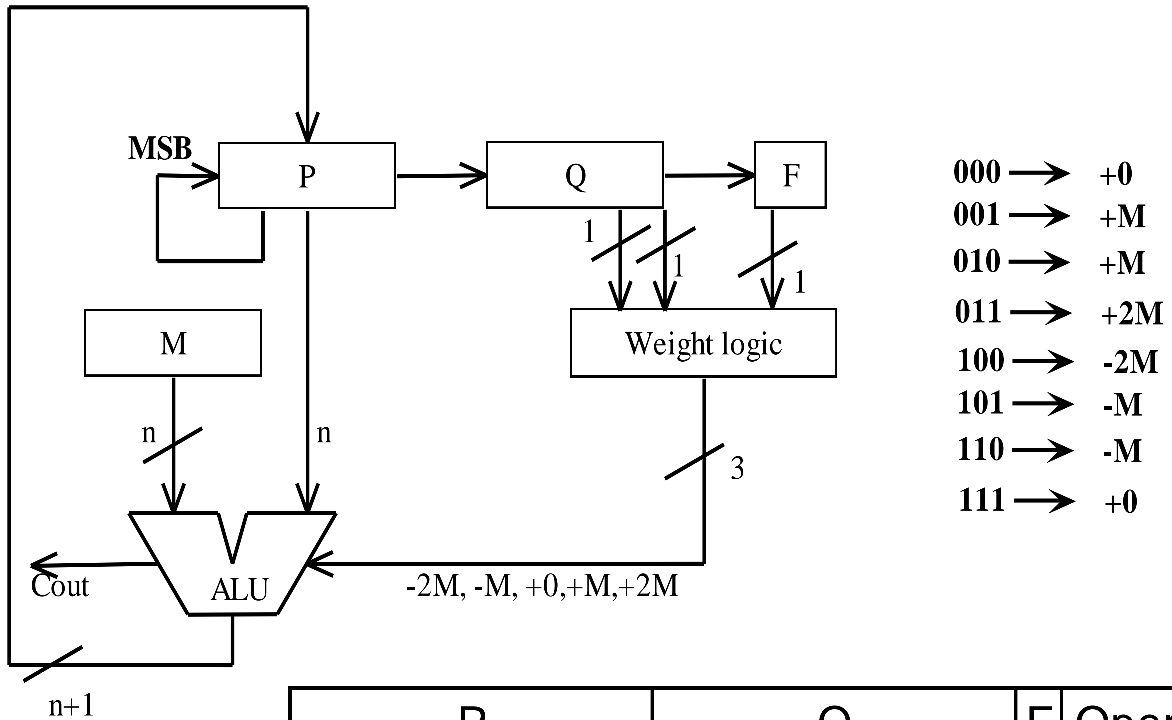
multiplier bit pair		next multiplier bit at right	booth bits	equivalent multiplicand
qi+1	qi	qi-1		
0	0	0	+0+0	0*M
0	0	1	+0+1	+1*M
0	1	0	+1-1	+1*M
0	1	1	+1+0	+2*M
1	0	0	-1+0	-2*M
1	0	1	-1+1	-1*M
1	1	0	+0-1	-1*M
1	1	1	+0+0	0*M

4.4.3Fast Multipliers(cont'd)

- Example (-24)*(-19)=101000*101101
  - Booth multiplication

  - Fast Multiplication

4.4.3Fast Multipliers Implementation



-24 (101000)
-19 (101101)

M=101000
2M=1010000
-M=011000
-2M=0110000

P								Q								F	Operation
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	1	0	initialize
1	1	1	0	1	0	0	0	1	1	1	0	1	1	0	1	0	+M
1	1	1	1	1	0	1	0	0	0	1	1	1	0	1	1	0	shift 2 bits
0	0	0	1	0	0	1	0	0	0	1	1	1	0	1	1	0	-M
0	0	0	0	0	1	0	0	1	0	0	0	1	1	1	0	1	shift 2 bits
0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	0	1	-M
0	0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	shift 2 bits
0	0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	1	shift 2 bits

4.5.1 Unsigned Binary Division

- Division is more complicated to implement in hardware than multiplication.
- Division by hand involves conditional subtraction and shifting
- Hardware implementations of division are usually sequential approximations of the hand division algorithm.
- Example : 100010010(dividend)
- 1101(divisor)

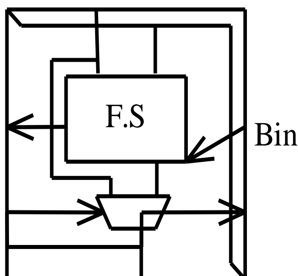
4.5.2 Purely Combinational Array Divider

Divisor

Dividend

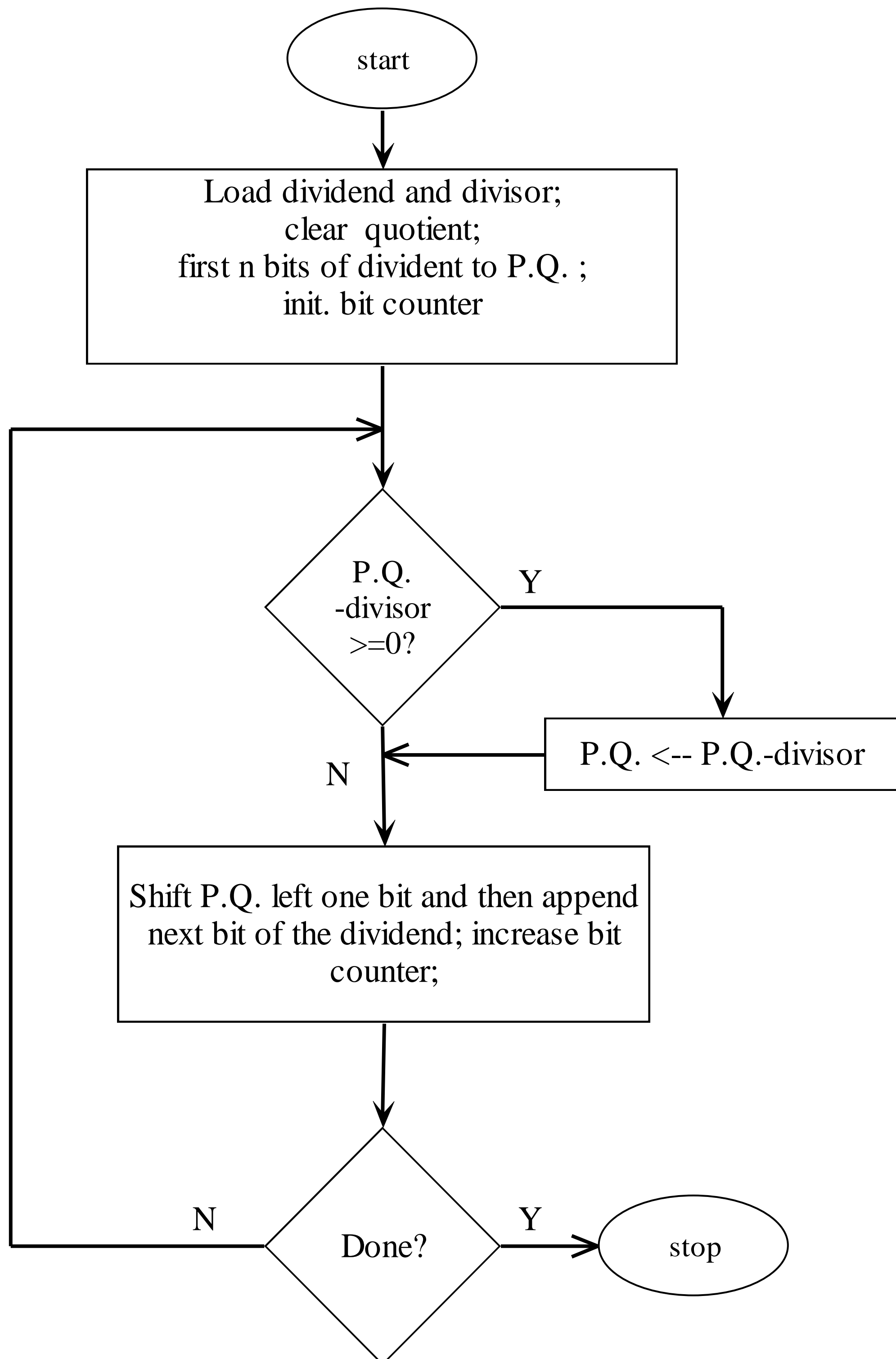
Quotient

Reminder

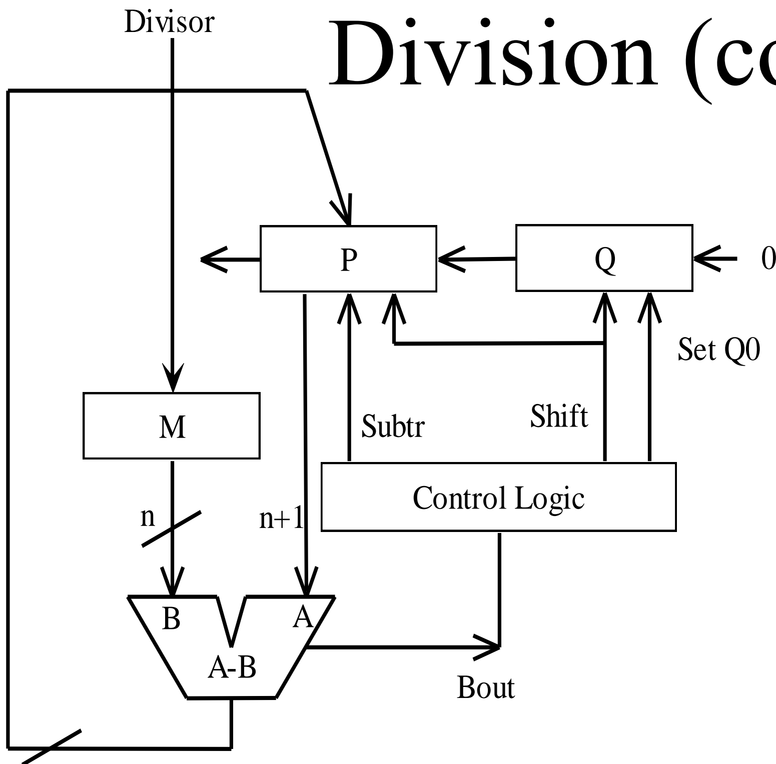


- Time delay 3 n^2 tg

4.5.3 Comparison Method Division



4.5.3 Comparison Method Division (cont'd)

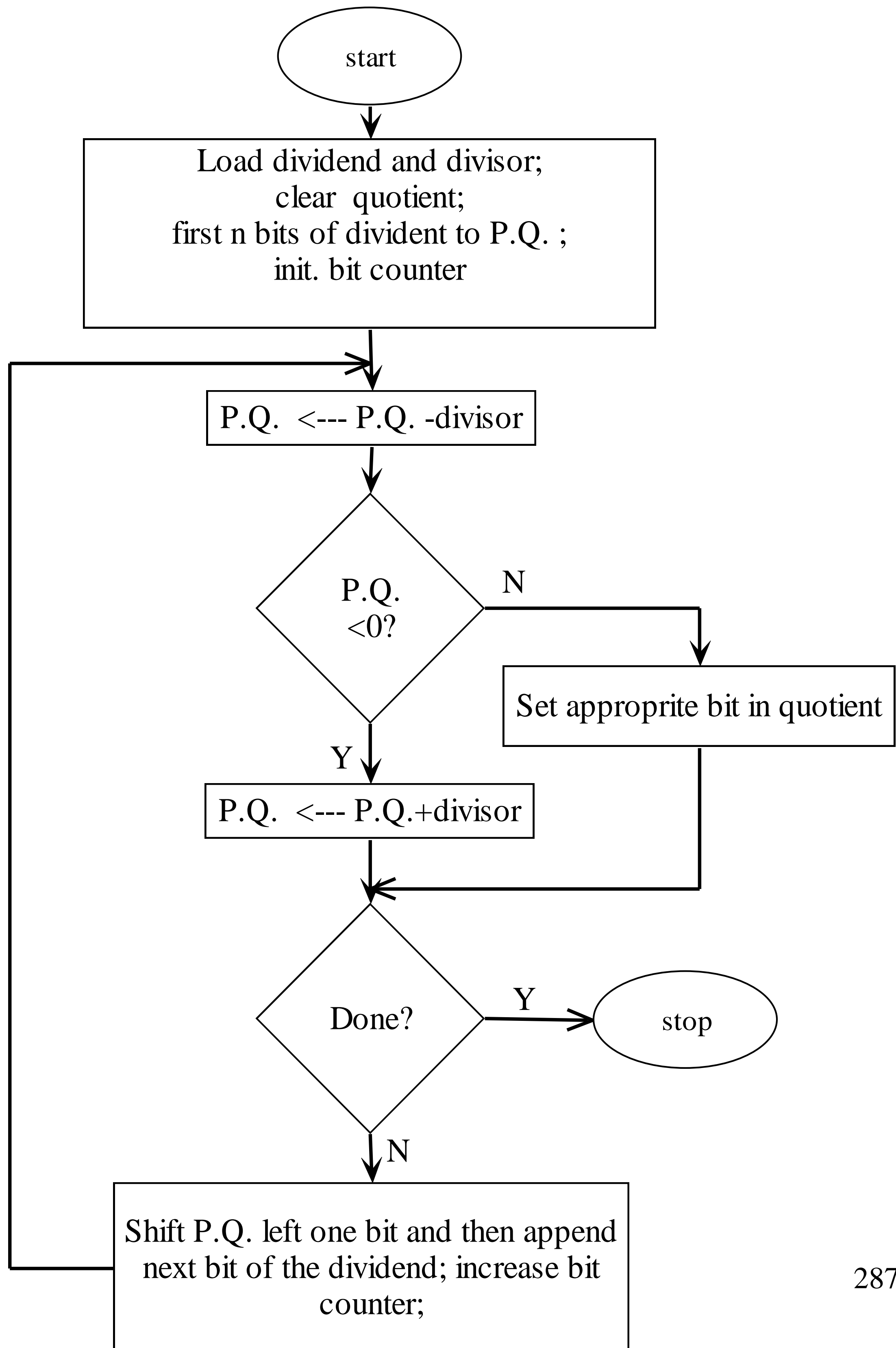


01011 | 100010010

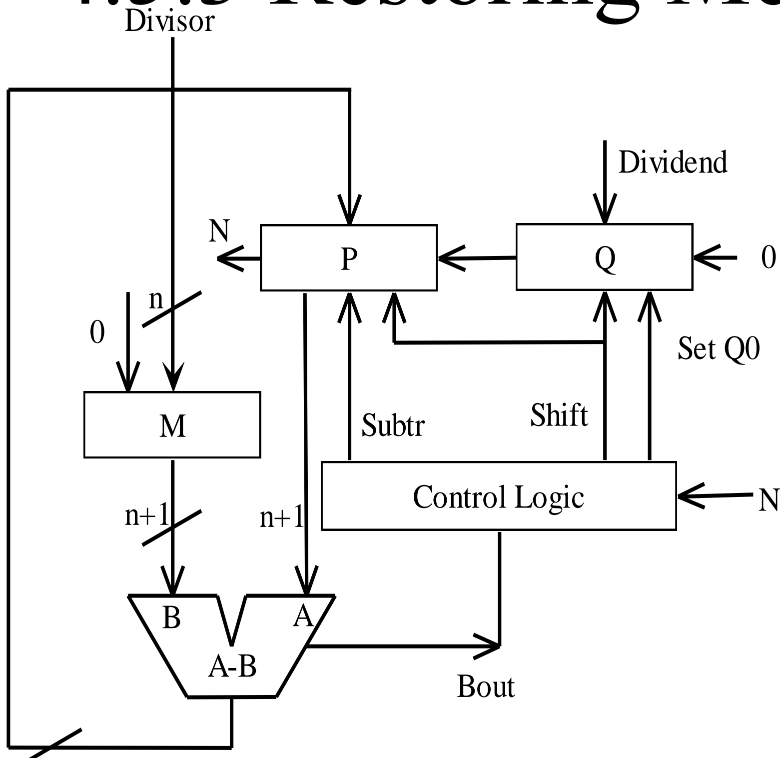
M=01011

B	P					Q									operation
1	0	0	0	0	0	1	0	0	0	1	0	0	1	0	load M,Q, clear P
1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	shift P.Q
1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	shift
1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	shift
1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	shift
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	subtr.+set Q0
1	0	0	1	1	0	0	0	1	0	0	0	0	0	1	shift
0	0	1	1	0	0	0	1	0	0	0	0	0	1	0	subtr. +set Q0
1	0	0	0	0	1	0	1	0	0	0	0	0	1	1	shift
1	0	0	0	1	0	1	0	0	0	0	0	1	1	0	shift
1	0	0	1	0	1	0	0	0	0	0	1	1	0	0	shift
1	0	1	0	1	0	0	0	0	0	1	1	0	0	0	

4.5.3 Restoring Method for Division



4.5.3 Restoring Method cont'd

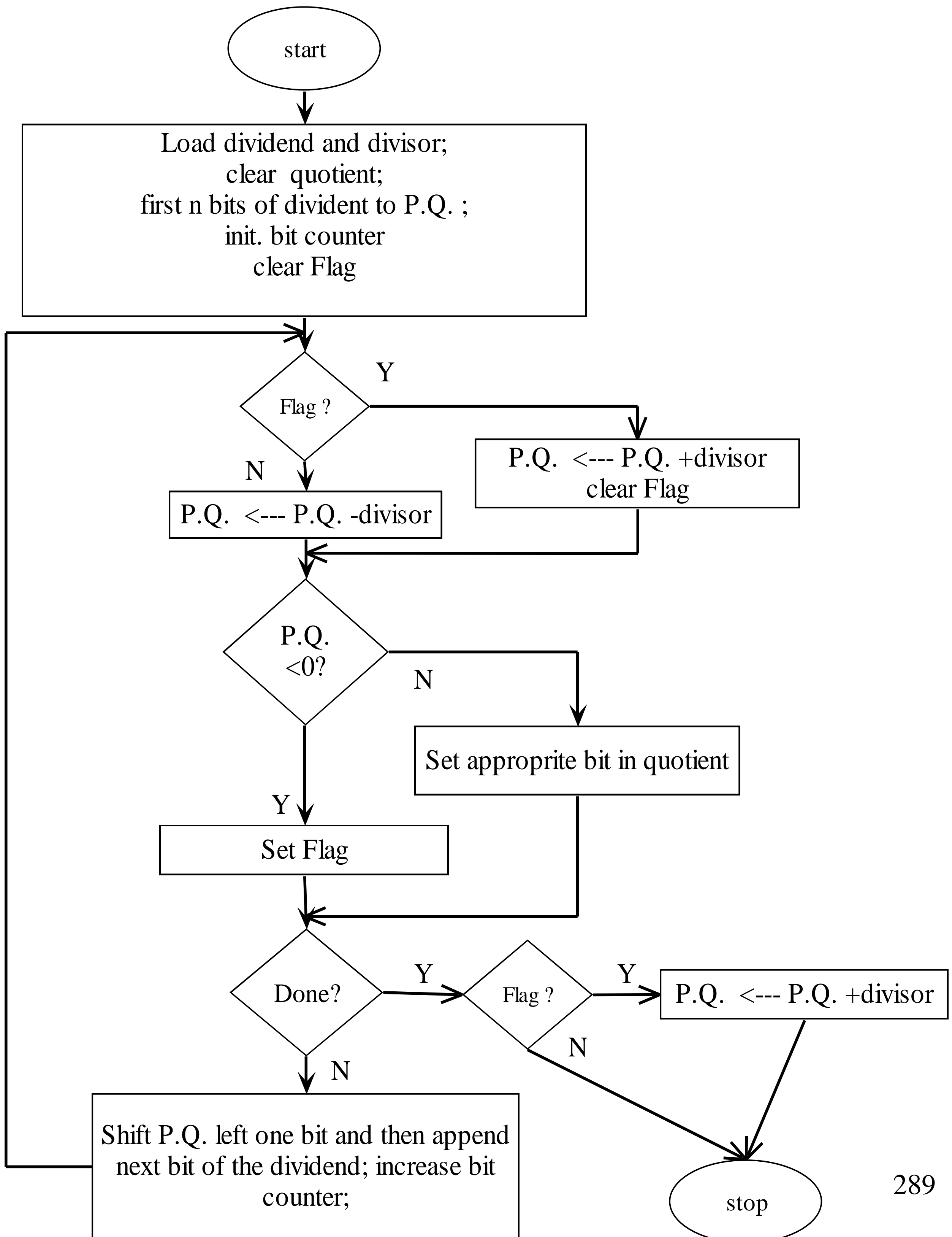


0100 | 1001

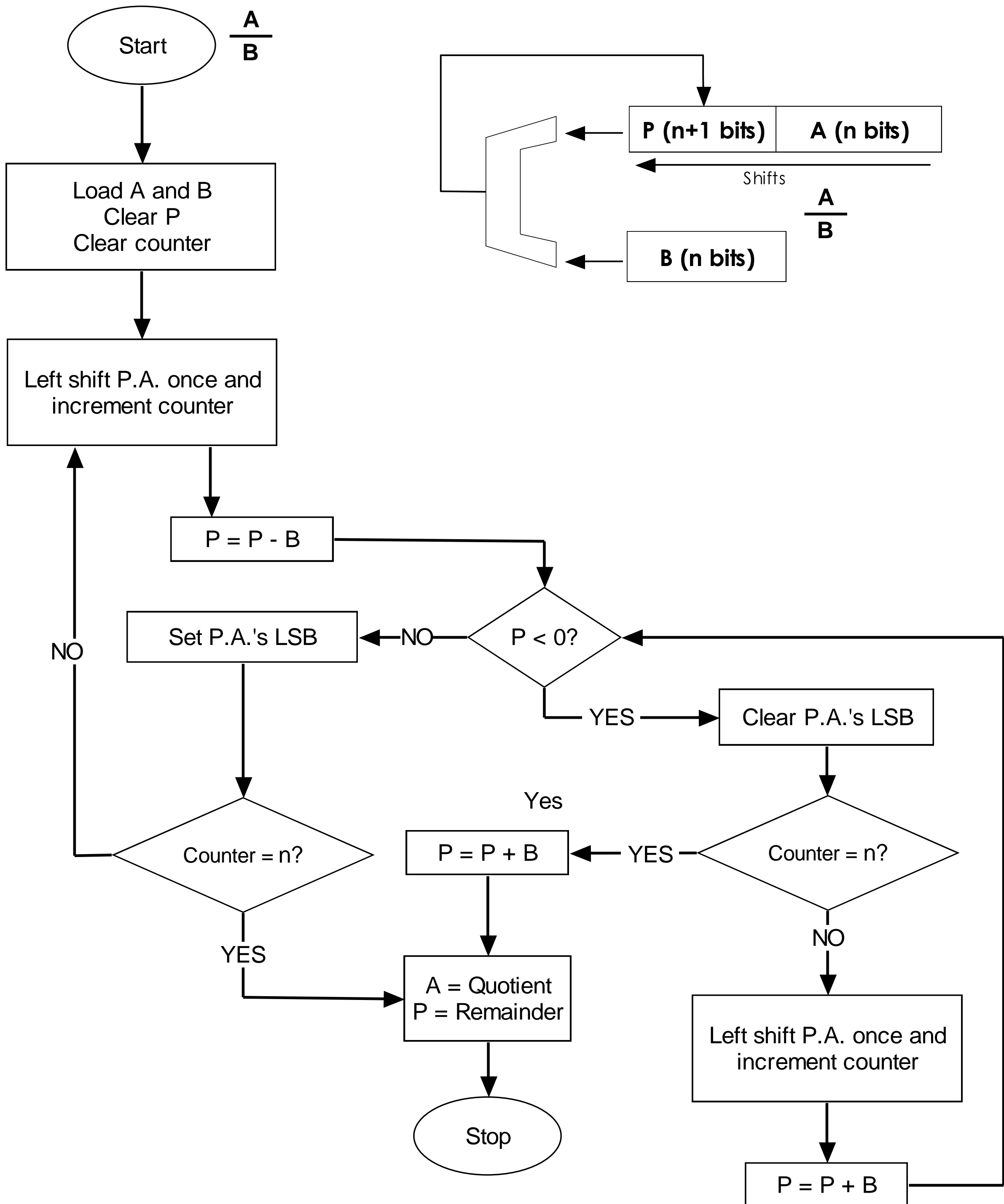
M=0100

M					P					Q					operation
0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	load M,Q, clear P
					0	0	0	0	0	1	0	0	1	0	shift left
					1	1	1	0	0	1	0	0	1	0	P <--P-M
					0	0	0	0	0	1	0	0	1	0	P<--P+M
					0	0	0	0	1	0	0	1	0	0	shift left
					1	1	1	0	1	0	0	1	0	0	P<--P-M
					0	0	0	0	1	0	0	1	0	0	P<--P+M
					0	0	0	1	0	0	1	0	0	0	shift left
					1	1	1	1	0	0	1	0	0	0	P<--P-M
					0	0	0	1	0	0	1	0	0	0	P<--P+M
					0	0	1	0	0	1	0	0	0	0	shift left
					0	0	0	0	0	1	0	0	0	1	P<--P-M
					0	0	0	0	1	0	0	0	1	0	shift left
					1	1	1	0	1	0	0	0	1	0	P<--P-M
					0	0	0	0	1	0	0	0	1	0	P<--P+M

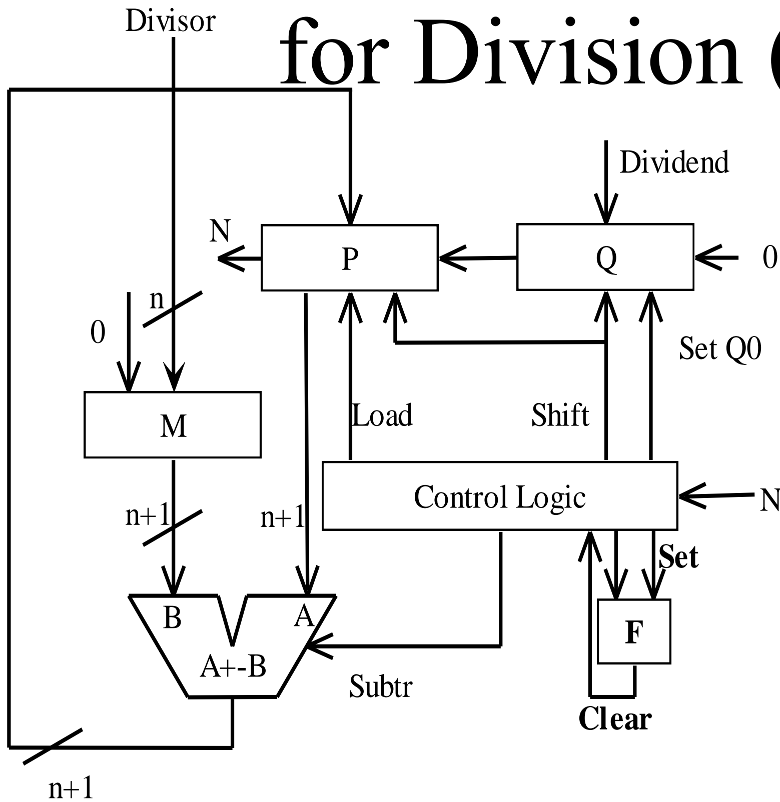
4.5.4 Non Restoring Method for Division



Non-restoring Division Dataflow Diagram



4.5.4 Non Restoring Method for Division (cont’d)



0100 |1001

M=0100

M					F	P					Q					operation
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	load M,Q, clear P
					0	0	0	0	0	0	1	0	0	1	0	shift left
					1	1	1	1	0	0	1	0	0	1	0	P=P-M
					1	1	1	0	0	1	0	0	1	0	0	shift left
					1	1	1	1	0	1	0	0	1	0	0	P=P+M
					1	1	1	0	1	0	0	1	0	0	0	shift left
					1	1	1	1	1	0	0	1	0	0	0	P=P+M
					1	1	1	1	0	0	1	0	0	0	0	shift
					0	0	0	0	0	0	1	0	0	0	1	P=P+M
					0	0	0	0	0	1	0	0	0	1	0	shift
					1	1	1	1	0	1	0	0	0	1	0	P=P+M*
					0	0	0	0	0	1	0	0	0	1	0	P=P+M

4.5.5 Possible Exception Conditions in Division

- Divide by zero
  - Undefined results
- Divide overflow
  - Divisor one word (n bit) , dividend two words (2n bit)
  - Possible results that the resulting quotient occupies n+1 bits when only n bits can be stored in a word
- Solution
  - Using software to avoid exceptions
  - Add special hardware to detect exceptions

4.6 Floating Point Arithmetic

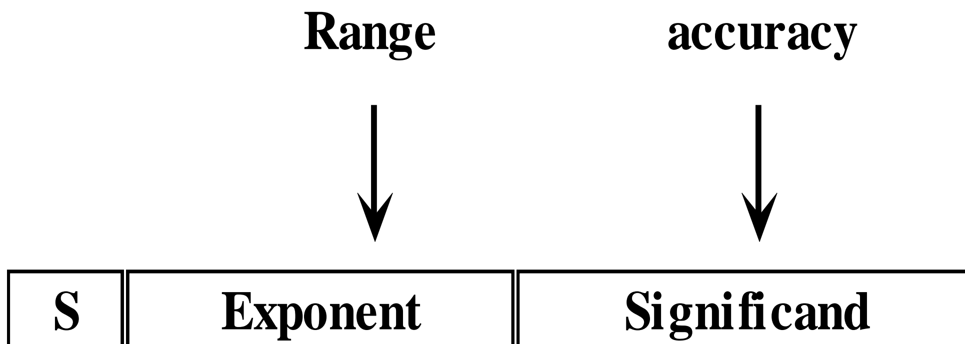
- Scientific Notation:
- Example : 123456789

- Normalized scientific notation
- 123456789

  - Same as scientific notation, except that there must be exactly one nonzero digit to the left of the radix point.
  - In normalized binary scientific notation, the bit the left of the radix point must be 1, or else the number is 0.

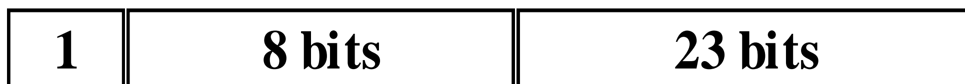
4.6.1 Hardware Representations of Scientific Notation

- Given: fixed word width
- Need to find out:
  - Number if bits in significant
  - Number of bits in exponent
  - Representation of negative numbers
- Problem:
  - Number of bits in exponent vs. number of bits in significand



4.6.1 IEEE 754 Floating Point Standard

- Single precision format (32 bits Wide)



- Double precision format: (64 Bit wide)

- Biased exponents? (logic and computer design fundamentals)
  - The exponent representation employed in most computers is known as a biased exponent. The bias is an excess number added to the exponent so that, internally, all exponents become positive.
- For single precision format offset=127
- For double precision format ?

4.6.1 IEEE 754 Floating Point Standard (cont'd)

- Example #1: Determine the IEEE 754 single precision representation of
- -4.125

1	10000001	00001

- 8.25

- Example #2 determine what decimal number is represented by following bits interpreted as a single precision IEEE 754 Value

1	10110110	1100

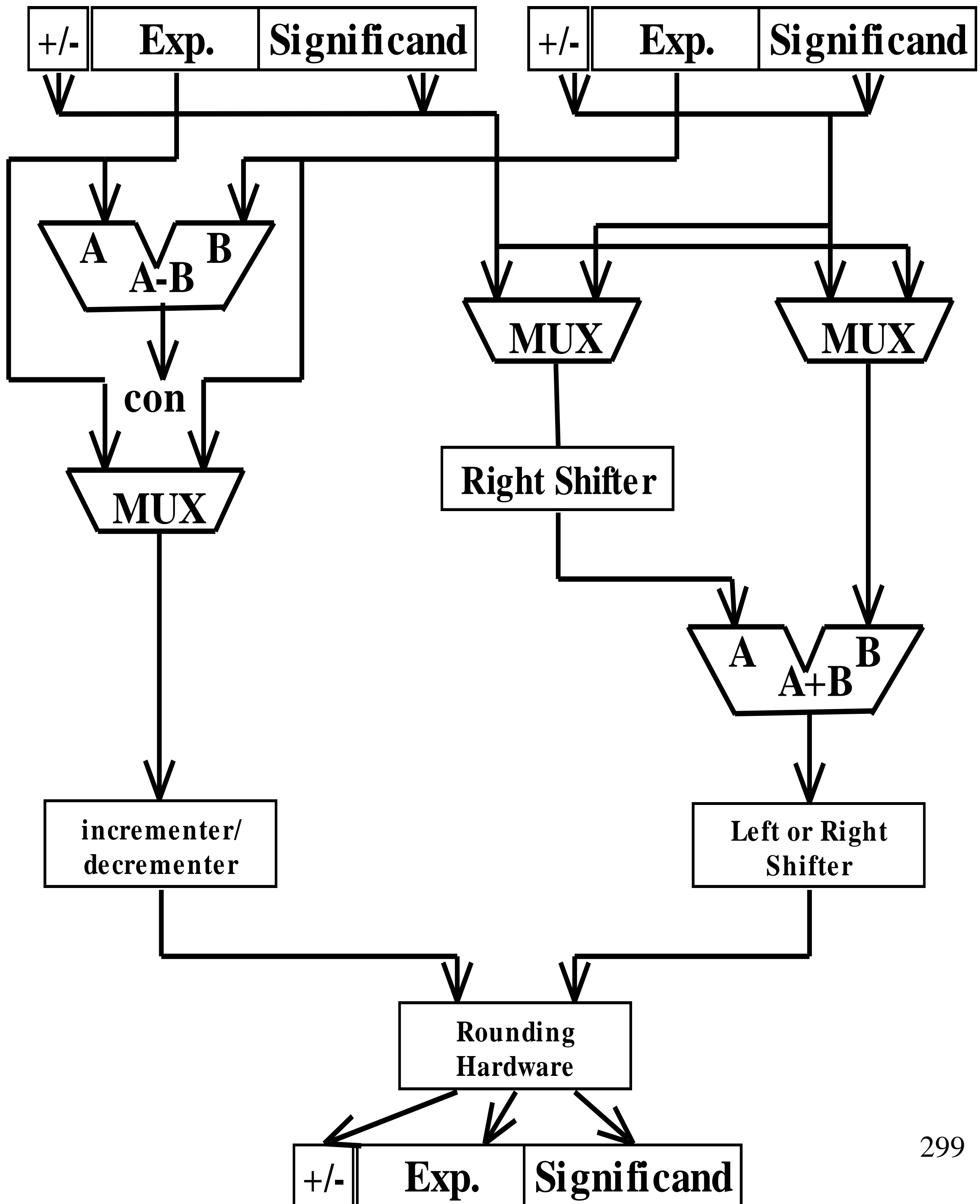
4.6.2 Floating Point Addition

I. Compare the exponents
   I. Shift the significant of the smaller number to the right to equalize the exponents
II. Add/subtract the significands
III. Normalize the results by either:
   I. Shifting right and incrementing exponent
   II. Shifting left and decrementing exponent
   III. Check for overflow or underflow
IV. Round the significand to the available number of bits
V. If results is not normalized, then repeat steps III and IV

4.6.2 Floating Point Addition(cont'd)

Example 0.625_{10}-0.4375_{10}

4.6.2 Floating Point Addition Data Path



4.6.3 Floating Point Multiplication

- Add the exponents
  - Subtract the bias (if any) from the sum
- Multiply the significands
- If necessary, normalize the product resulting from step I and step 2
  - Either shift significand right and add to exponent
  - Or shift significand left and subtract from exponent
  - Check the overflow and underflow
- Round the number to the available number of digits
  - Repeat step III and step IV if not normalized
- Obtain the correct sign of the product from the signs of the original operands

4.6.3 Floating Point Multiplication (cont'd)

- Example 7.25*-6.125

V. Digital System Testing

5. Digital System Testing

5.1 Introduction to Digital System Testing

5.2 Fault and Fault Models

5.3 Boundary Scan

5.1 Introduction to Digital System Testing

- Highlight of IC
  - LSI (1000), VLSI (100000) ULSI (1M)
  - Circuit densities: doubling every 18 month
  - Reliability up – better testing
  - Cost Down
  - Yield up
- Testing
  - Main purpose of fault testing is the detection of malfunctions
  - Location of the fault may also be desired

5.1 Introduction to Digital System Testing (cont'd)

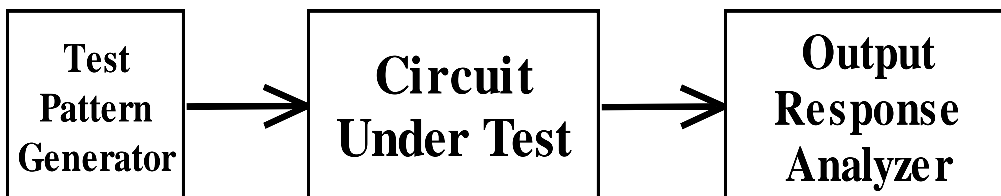
- Why test IC
  - To verify design
  - To detect faults arising from manufacture or ware-out
  - To ensure components meet design specification (parameter)
- Cost of Testing
  - IC $X
  - PCB board $ 10X
  - System $ 100X
  - Test cost > 50% production cost

5.1 Introduction to Digital System Testing (cont’d)

- Verification
  - A design is checked to meet the requirement and specifications
  - Functions test checks that the circuit implement the functions that is required, and does not implement the function that is not required
- Factors
  - Original design specifications
  - Delays, hazards
  - Routing, layout design rules
- Tools
  - Simulators : logic, timing, functions behavioral
  - HDL, VHDL

5.1 Introduction to Digital System Testing (cont'd)

- Testability
  - Observability: the internal signals can be determined at the output
  - Controllability: producing an internal signal value by applying signals to inputs
- Testing method
  - Off-line: external + BIT(built in test)
  - On-line: BIT (built-in test)
  - Example
- Structural testing

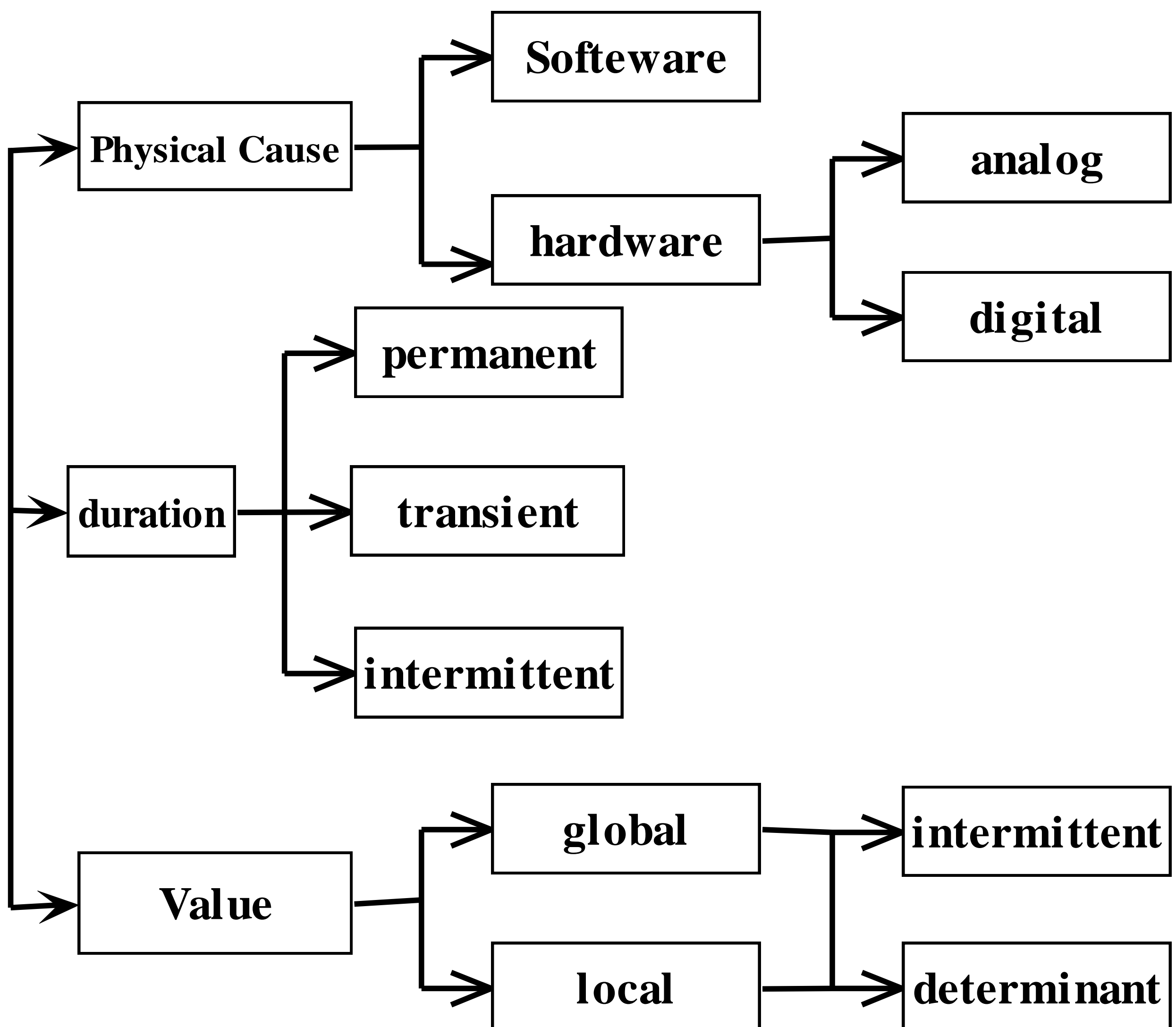


5.1 Introduction to Digital System Testing (cont'd)

- Ideal Testing Environment
  - Store the truth table of the circuit
  - Apply all possible input combinations
  - Examine the output responses
- Real testing environment
  - Fault modeling
    - Model physical detects by a small number of faults
  - Test pattern generation
    - Generate a test pattern
  - Output response analysis
    - Analyze the circuit output without storing al the circuit responses
- Fault coverage
  - Number of detected faults/ number of faults modeled

5.2 Fault and Fault Models

- Before testing takes place, it is necessary to examine failure mechanisms, their effects and methods of modeling them

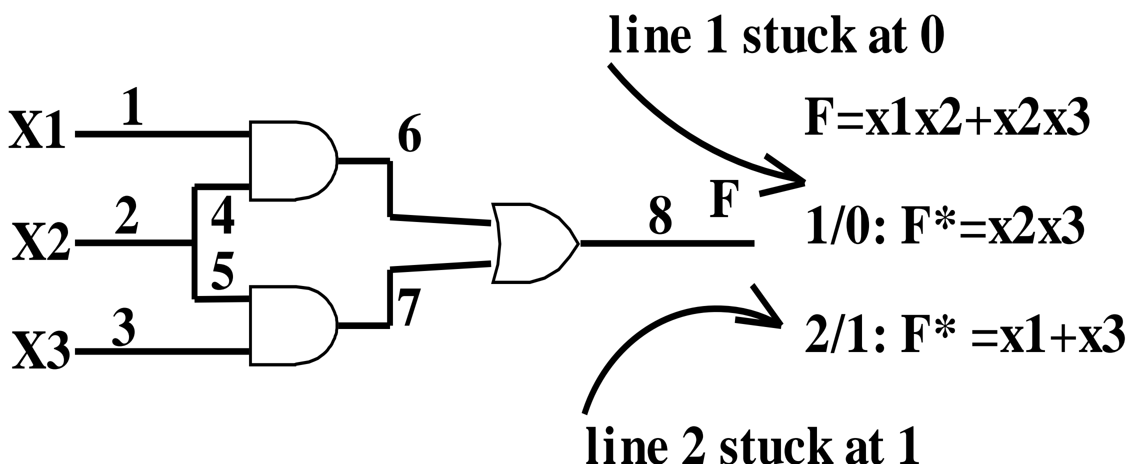


5.2 Fault and Fault Models (cont’d)

- Fault characteristics:
  - Permanent fault– fault in existence long enough to be observed at test time
  - Transient fault: fault appears and disappears in short intervals of time
  - Intermittent fault: fault which appears and disappears at regular time intervals
- Faults models
  - A logical abstraction describing the functional effects of a physical failure
  - Different levels of modeling are used based on different primitives
  - Block: functional model, block diagram (data-path)
  - Gate: switching level, gates and latches
  - Circuit: electrical model, use transistors, resistors capacitors
  - Geometrical-model: layout description of chip ..

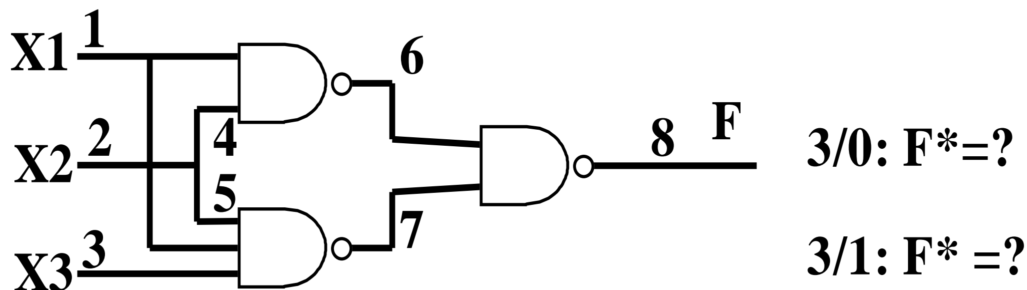
5.2.1 Stuck at Fault Model

- Most commonly used fault model
- May consider single or multiple stuck-at-faults
- The effect of the fault is modeled by having a line segment stuck at 0 or 1
  - All gates are perfect, problem may occur on line segments.
  - Single stuck at fault (SSF)
  - Example:
  - {x1,x2,x3} : inputs {1 2 3,4,5,6,7,8} : line segment number, F : correct output function, F* incorrect output function



5.2.1 Undetectable Faults

- Undetectable fault: its effect cannot be seen at the output, no matter what test pattern is applied



- Problem: redundancy may lead to undetectability
  - Why redundancy?
  - How to detect?
  - Undetectable faults may lead to problems in detecting other detectable faults

5.2.1 Equivalent Faults

- What is equivalent faults?
- Example1:

a 1 — b 2 — 3 — F (AND gate)

1/0: F* =?

2/0: F* =?

3/0: F*=?

- Example 2:

a 1 — b 2 — 3 — F (OR gate)

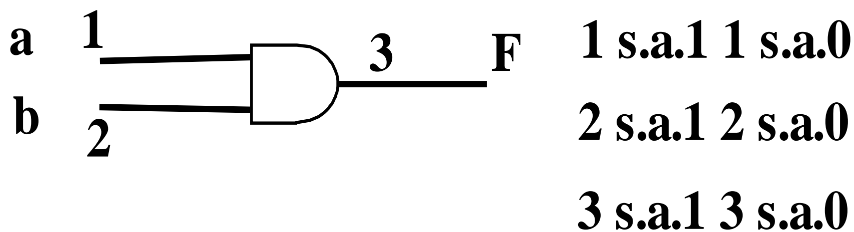
1/1: F* =?

2/1: F* =?

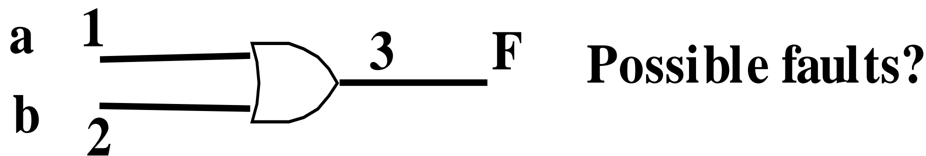
3/1: F*=?

5.2.1 Equivalent Faults (cont'd)

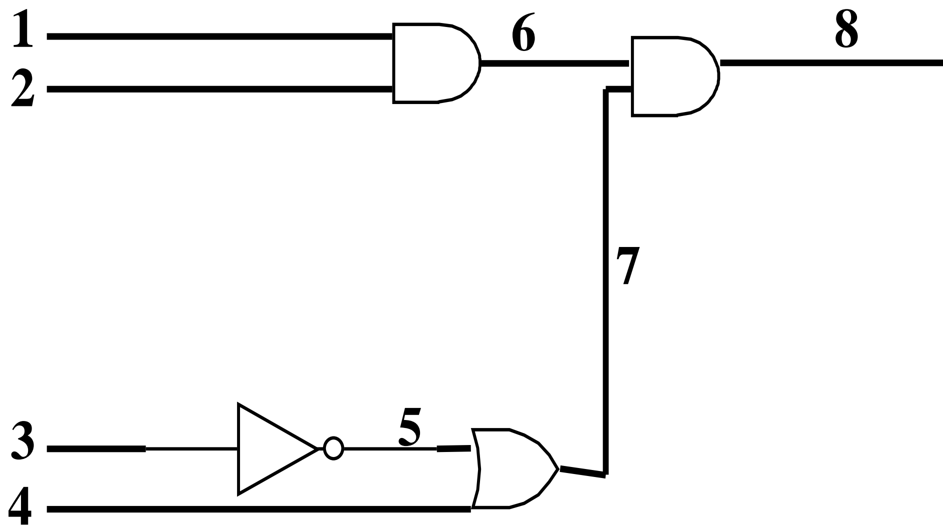
- What is equivalent faults?
- Example1:



- Example 2:

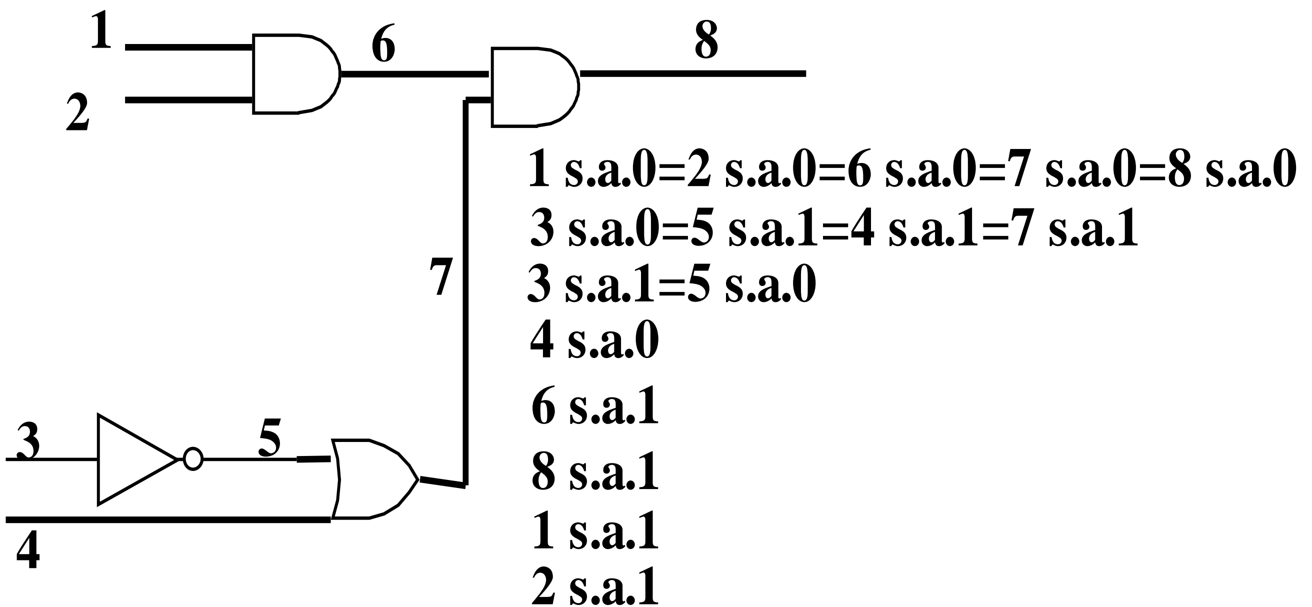


5.2.1 Equivalent Faults (cont'd)



5.2.2 Determine Test Pattern Generation

- Find a test vector for a given fault
  - Fault propagation
  - Path sensitization and backtracking

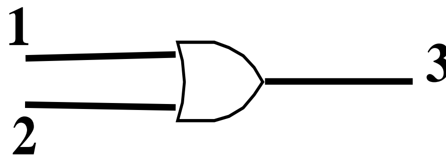


  - 1/0 : 1234= 1101,1111,1100
  - 3/0 : 1234= 1110
  - 3/1 : 1234= 1100
  - 4/0 : 1234 = 1111
  - 6/1 : 1234 =
  - For each of the test faults, find the test vectors
  - Remove the redundant one, find the minimum set

5.2.2 Determine Test Pattern Generation (cont'd)

Fault equivalence depends on individual gates

- AND any input/0 ←→output/0
- Or any input/1 ←→output/1
- NAND any input/0 ←→output/1
- NOR any input/1 ←→output/0
- NOT any input/0 ←→output/1
- any input/1 ←→output/0



5.2.2 Determine Test Pattern Generation (ATPG)

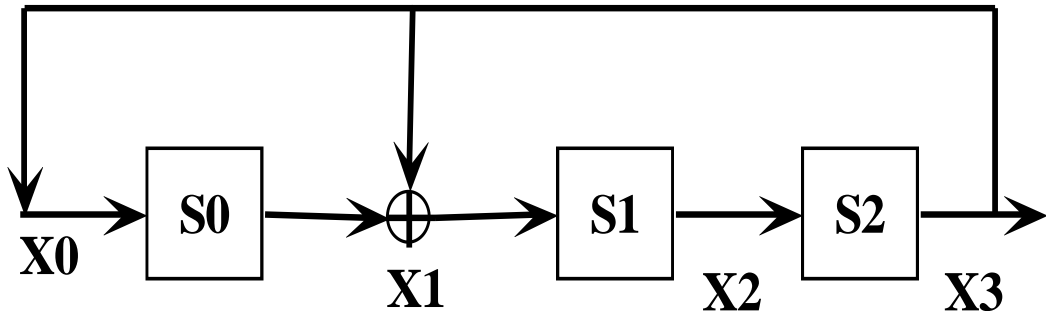
- Given a circuit of n line segments
  - There are 2n single stuck at faults
  - Fault collapsing reduces 2n faults to k faults
  - For each of the k faults, find a test vectors
  - Find a minimal test set from the test vectors that detect the k faults

- Advantages:
  - Short test length
  - 100% fault coverage

- Disadvantages:
  - very computational expensive
  - Need to store test vectors

5.2.3 Pseudorandom Test Pattern Generation

- Using an autonomous linear feedback shift register (ALFSR)



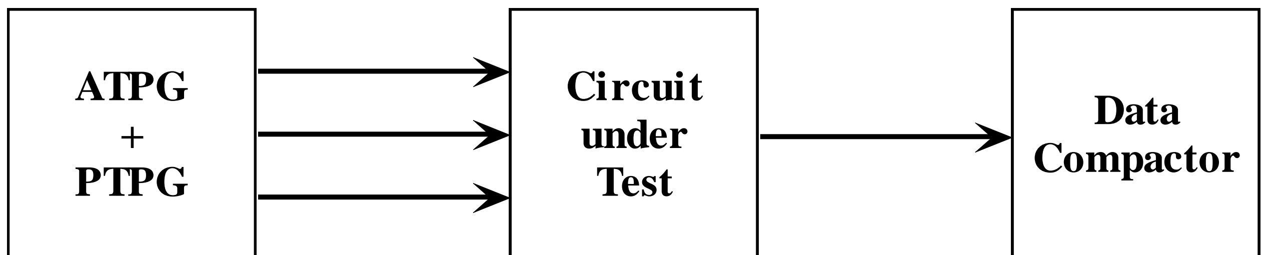
$P(X)=X^3+X+1$

Time	So	S1	S2
T1	0	0	1
T2	1	1	0
T3	0	1	1
T4	1	1	1
T5	1	0	1
T6	1	0	0
T7	0	1	0

5.2.3 Pseudorandom Test Pattern Generation (cont'd)

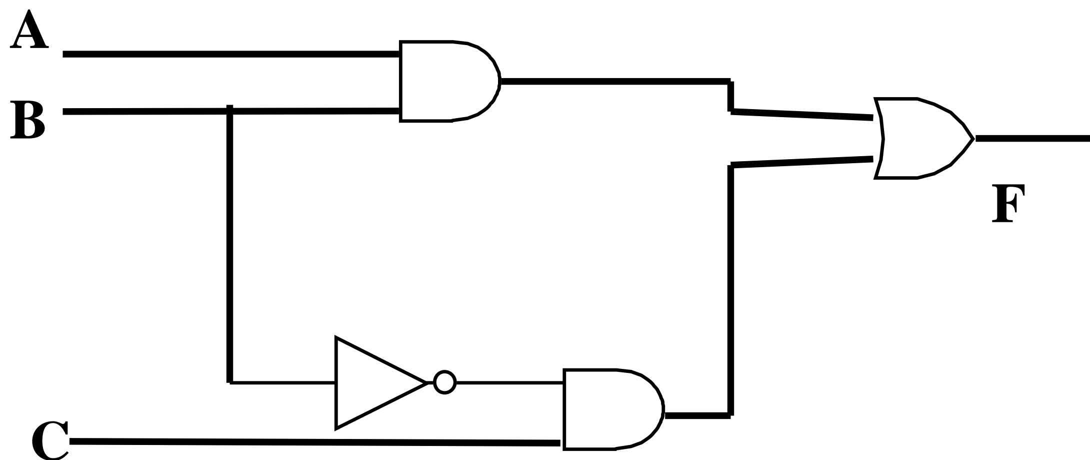
- Given a degree n P(X) in binary field
  - Degree n→ n stage ALFSR
  - If an ALFSR cycles through all 2^n-1 non zero states, it has the maximum length cycle, the corresponding P(x) is primitive. Otherwise, it is non primitive
  - Example :
  - $P(x)=X^7+X^3+X^2+1$
  - Pseudorandom Test Pattern Generation (PTPG)
  - Advantages: easy to generate
  - No need for storage
  - Suitable for built-in self test
  - Disadvantages:
  - Relative long test length
  - Some faults may not be tested?
- Solution
  - DO PTPG cover 90-95%
  - DO ATPG cover rest

5.2.4 An Off-line Test (signature analysis)



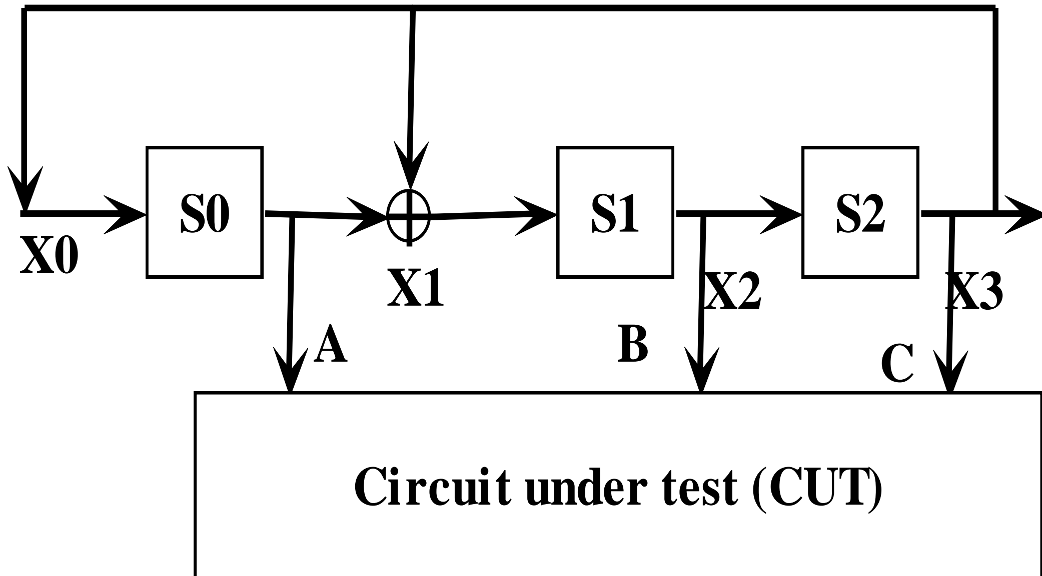
5.2.4 An Off-line Test (Cont’d)

- Example



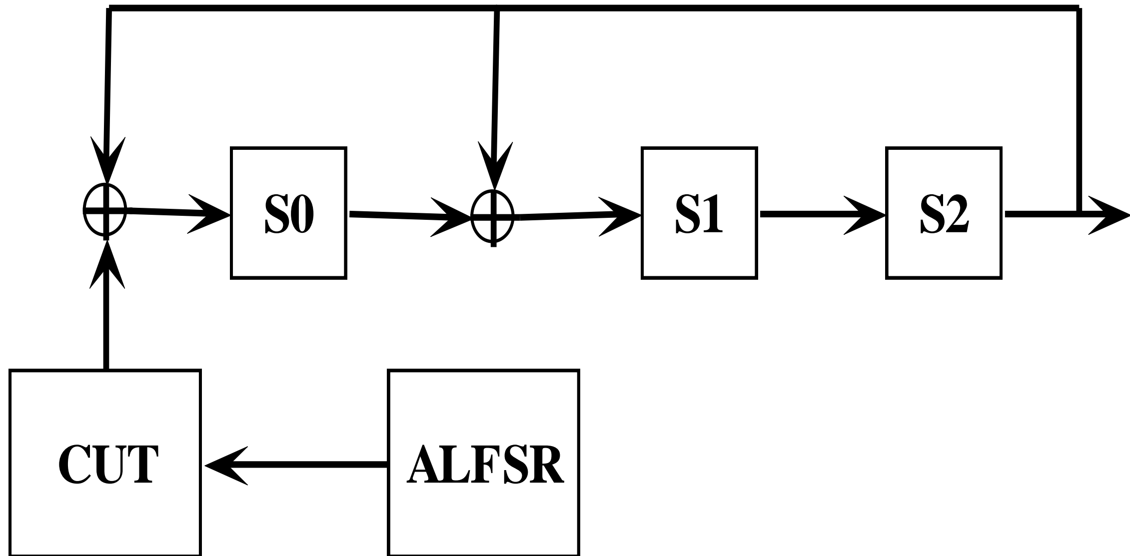
$F = AB + \overline{B}C$

5.2.4 An Off-line Test (Cont'd) example



Time	So	S1	S2	f
T1	0	0	1	1
T2	1	1	0	1
T3	0	1	1	0
T4	1	1	1	1
T5	1	0	1	1
T6	1	0	0	0

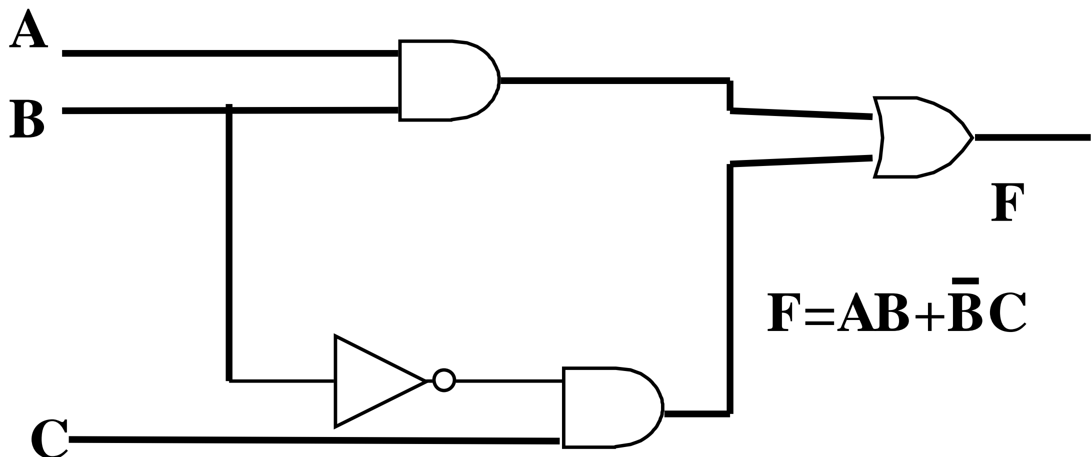
5.2.4 An Off-line Test (Cont'd) example



Time	f	So	S1	S2
T0	--	0	0	0
T1	1	1	0	0
T2	1	1	1	0
T3	0	0	1	1
T4	1	0	1	1
T5	1	0	0	1
T6	0	1	1	1

- At T6, s0s1s2=111 is a fault free signature

5.2.4 An Off-line Test (Cont'd) example



5 s.a.1 **$F^* = A + \bar{B}C$**

Time	So	S1	S2	f
T1	0	0	1	1
T2	1	1	0	1
T3	0	1	1	0
T4	1	1	1	1
T5	1	0	1	1
T6	1	0	0	1

5.2.4 An Off-line Test (Cont'd) example

- At T6, s0s1s2=011 is a fault signature
- Fault is detected.

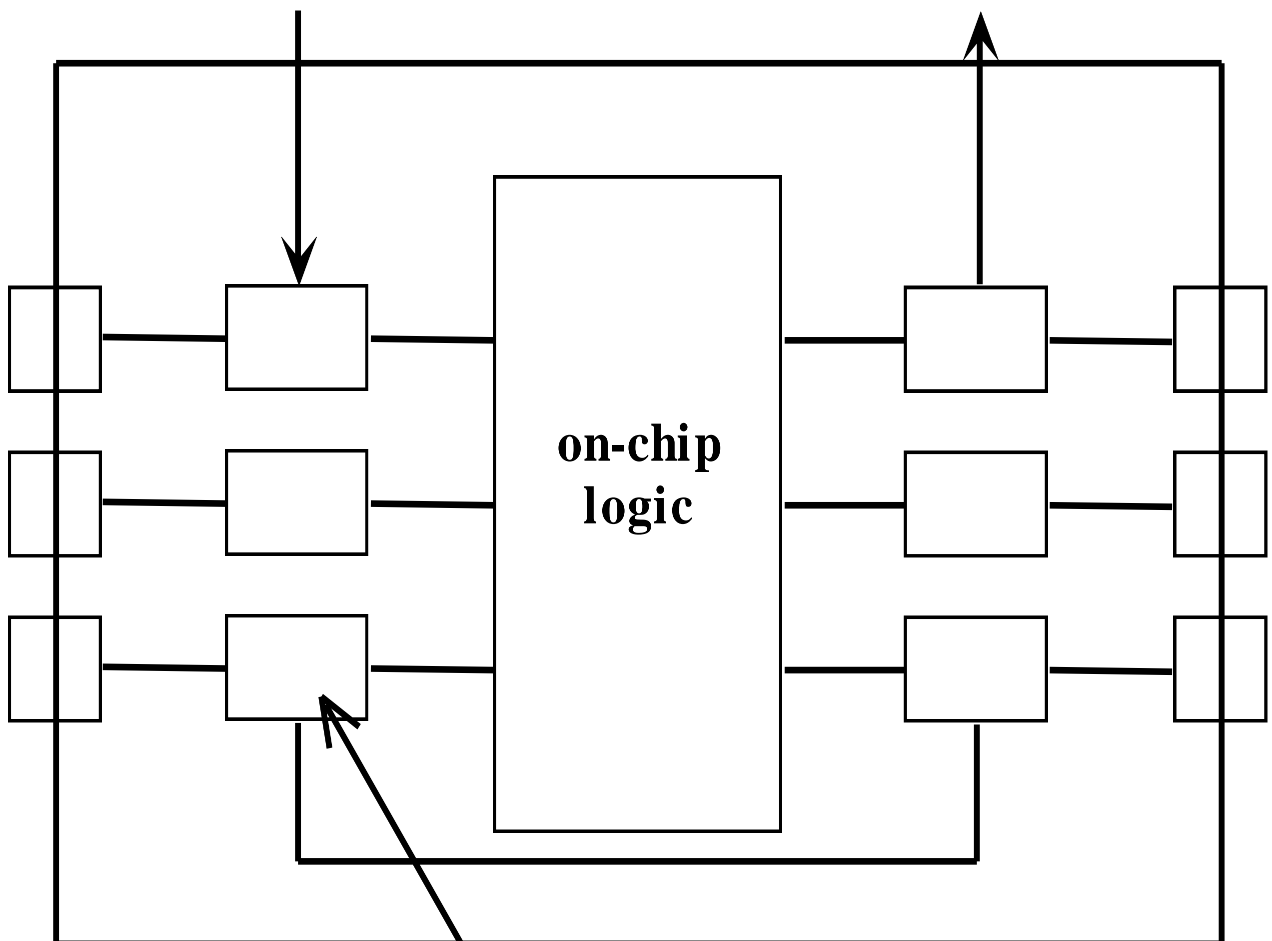
Time	f	So	S1	S2
T0	--	0	0	0
T1	1	1	0	0
T2	1	1	1	0
T3	0	0	1	1
T4	1	0	1	1
T5	1	0	0	1
T6	1	0	1	1

5.3 Boundary Scan

- A design and test technique that allows circuits to be tested via the board edge connectors
  - I/O hardware is modified in design to incorporate a programmable shift register
  - Overhead minimal as it uses some empty space around I/O pads
- Problems of board test
  - Increasing IC complexity
  - Increasing use of surface-mount technology

5.3 Boundary Scan (cont'd)

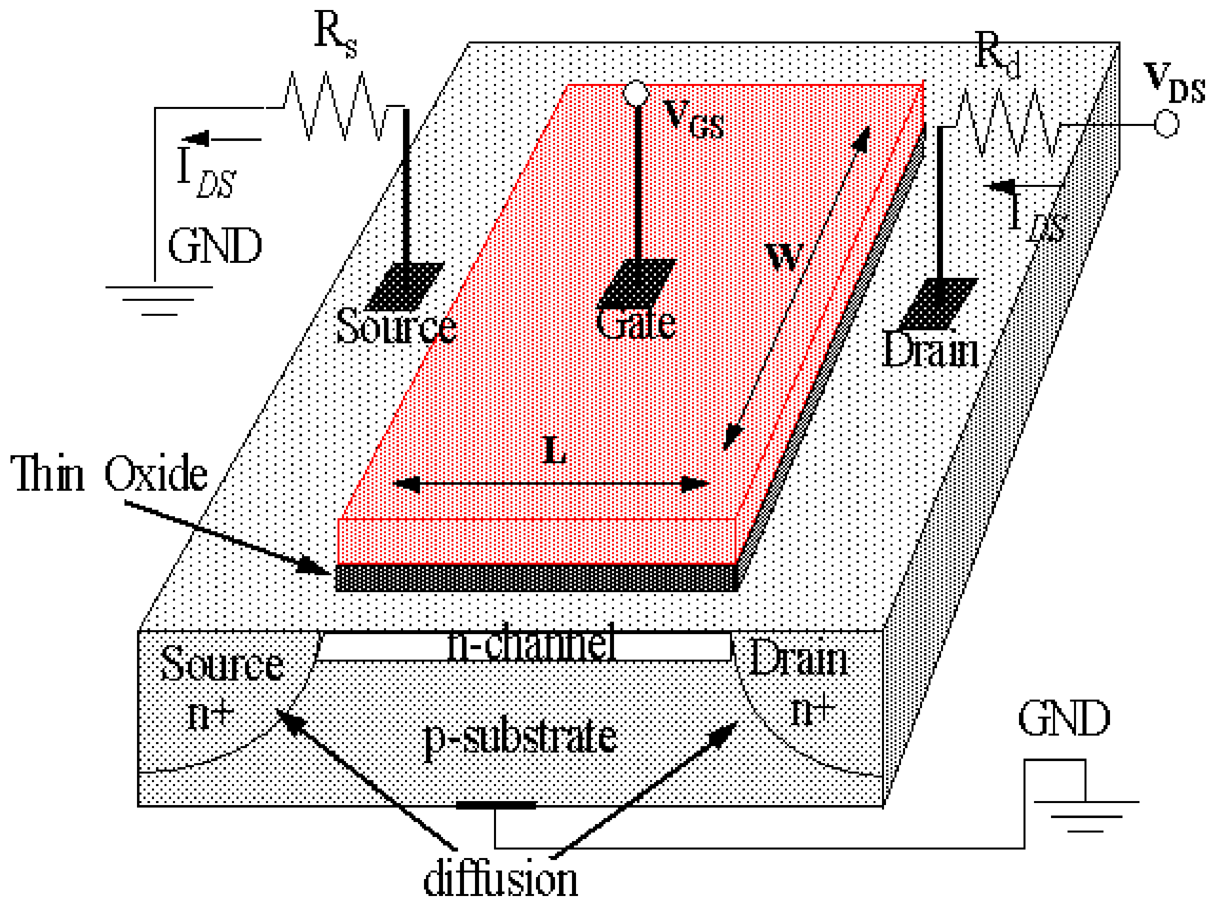
- Inclusion of programmable shift register stage adjacent to each I/O pin such that signals at component boundaries can be controlled and observed using scan testing principles



Section II. Fabrication

1. BASIC TRANSISTOR CHARACTERISTICS

1.1 MOSFET Behavior



M	**metal (gate)**
O	**oxide**
S	**semiconductor**

n^-	**$<10^{15}$ atoms /cm^3**
n^+	**$>10^{18}$ atoms /cm^3**
Si	**$5x10^{22}$ atoms /cm^3**

p dopant	boron	(3 valence electrons)
n dopant	phosphorous or arsenic	(5 valence electrons)

For $0 < V_{DS} < (V_{GS} - V_T)$, the transistor operate in the triode region, also called the linear region.

$$I_D = \mu_n c_{ox} \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}{}^2\right]$$

$$c_{ox} = \frac{\varepsilon_{ox}}{t_{ox}}; \varepsilon_{ox} permittivity; t_{ox} thickness$$

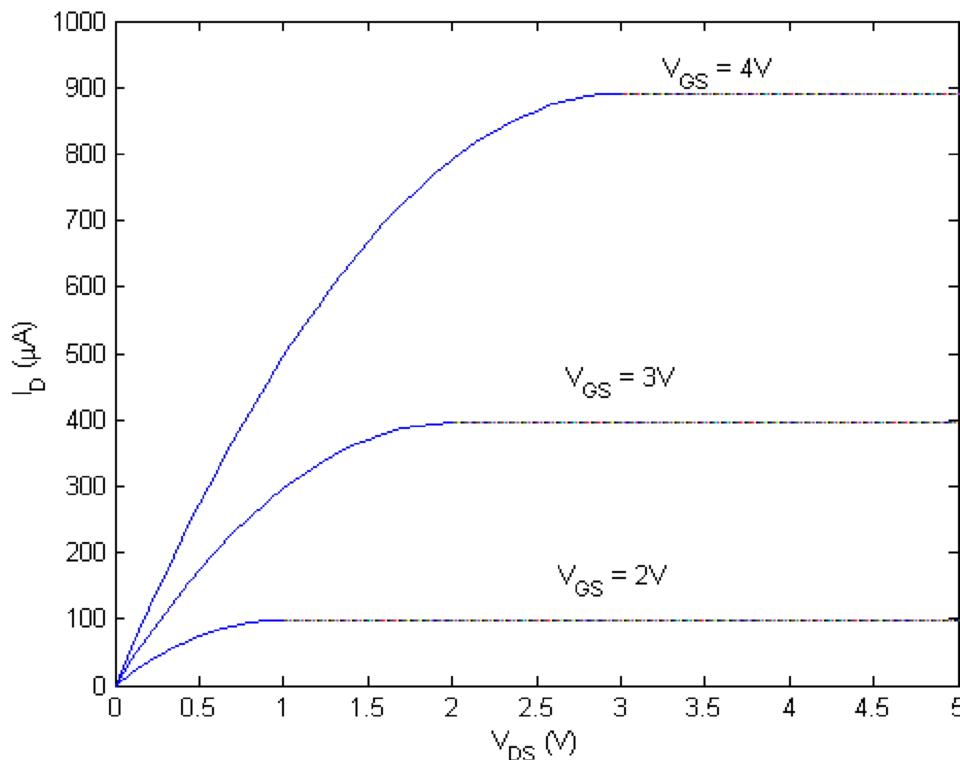
C_{ox}: gate oxide capacitance per unit area

$k_n' = \mu_n C_{ox}$ is called *process transconductance parameter*

μ_n is the average mobility of electrons in the channel.

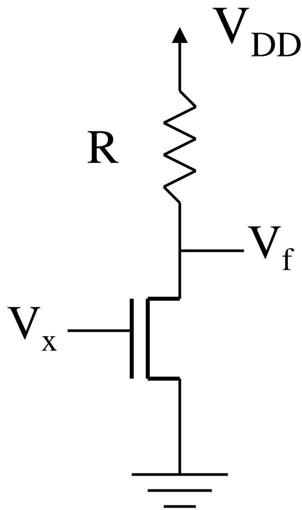
When $V_{DS} = V_{GS} - V_T$, the transistor is in the saturation region.

$$I_D = \mu_n c_{ox} \frac{W}{2L}(V_{GS} - V_T)^2$$



The current-voltage relationship in the NMOS transistor

1.2 Voltage Levels in Logic Gates



$$V_f = V_{DD} \frac{R_{DS}}{R_{DS} + R};$$

$$R_{DS} = V_{DS} / I_D = 1 / \left[k_n' \frac{W}{L} (V_{GS} - V_T) \right]$$

For

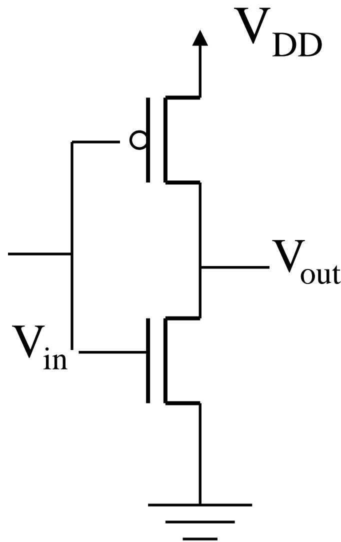
$k_n' = 60 \mu A / V^2, W / L = 2.0 \mu m / 0.5 \mu m, V_{GS} = 5V, and$

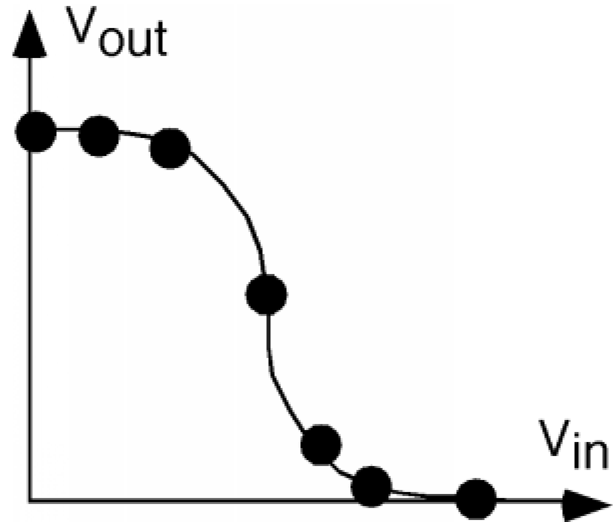
$V_T = 1V, R_{DS} \approx 1k\Omega$

For

$R = 25k\Omega, V_f \approx 0.2V$

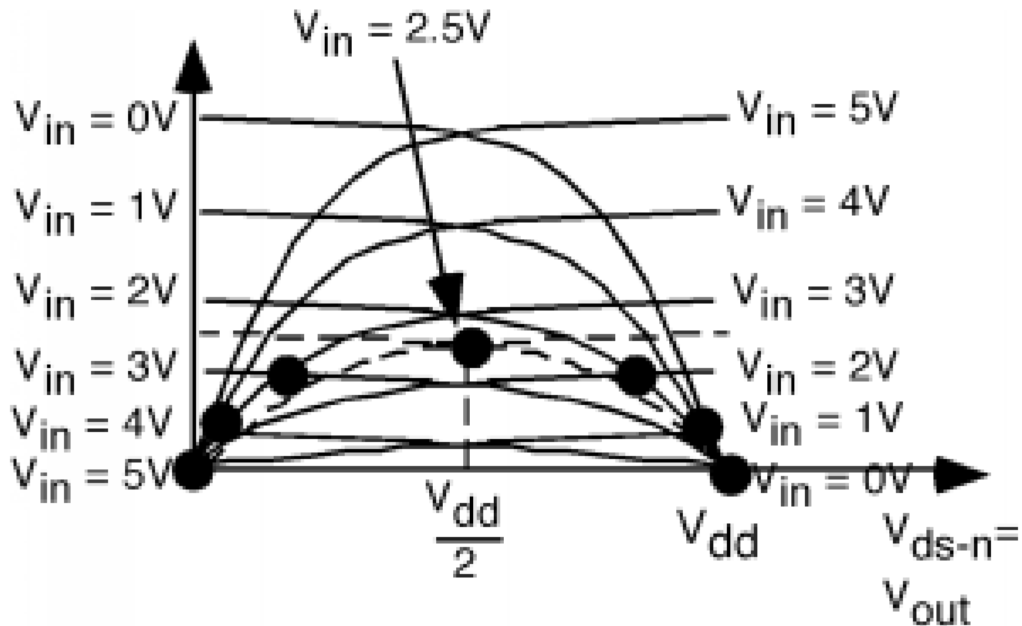
1.3 CMOS Inverter





p-substrate contact
m1-m2 contact
p-diffusion contact (source)
m2
(Out)
m1
n-substrate contact
n-diffusion contact (source)
glass
GND
V_{DD}
layer #3
layer #2
layer #1
p+
n+
n+
p+
p+
n+
(drains)
n-well
p substrate
n-transistor
polysilicon gate (In)
p-transistor

Cross-section of a CMOS inverter



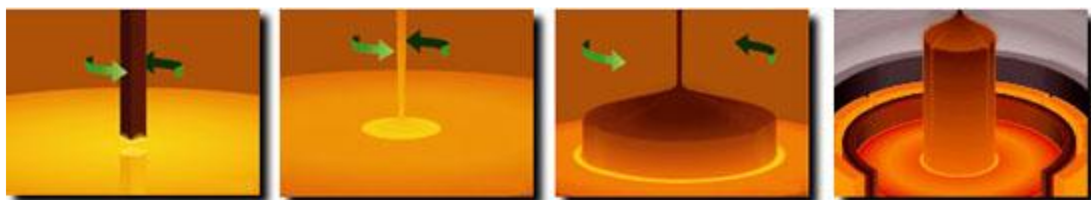
2. TECHNOLOGY

There are 4 'basic' processes required in IC manufacturing.

- Thin film deposition or growth
- Photolithography
- Etching
- Ion implantation and diffusion

The Semiconductor Manufacturing Process - Wafer Manufacturing

I. Crystal Pulling – Czochralski (CZ) method

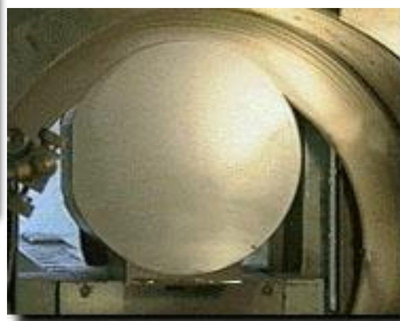


- Doped polycrystalline silicon melted at 1400 °
- Inert gas atmosphere of high-purity argon
- Single crystal silicon "seed" is placed into the melt and slowly rotated as it is "pulled out".
- Single crystalline ingot diameter is determined by a combination of temperature and extraction speed.
- The ingots are characterized by the orientation of their silicon crystals. One or two "flats" are ground into the diameter of the ingot.

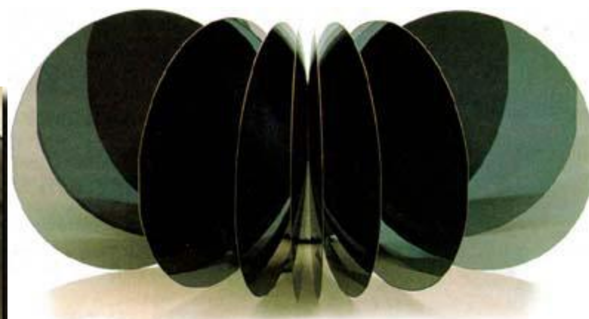
II. Wafer slicing

- After characterization, the ingot is sliced into individual wafers with precision "ID Saw".

Single Crystal Silicon Ingot



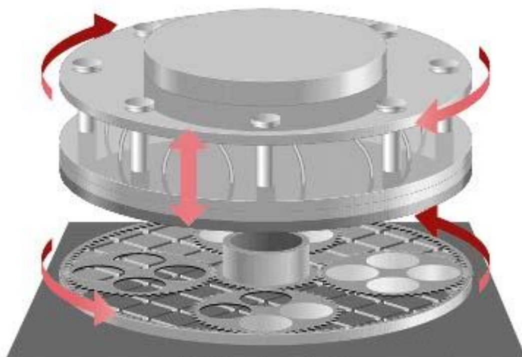
ID Wafer Slicing Saw



III. Wafer lapping, etching

• The sliced wafers are mechanically lapped using a counter-rotating lapping machine and an aluminum oxide slurry to flatten the wafer surface, makes them parallel and reduces mechanical defects.

• Wafers are then etched in a solution of nitride acid / acetic acid to remove microscopic cracks or surface damage followed by a series of high-purity RO/DI water baths.


Wafer Lapping Machine
(Mitsubishi Materials Silicon)

Wafer Polishing
(Strasbaugh Corporation)

IV. Wafer polishing and Cleaning

• Next, the wafers are polished in a series of combination chemical and mechanical polishing processes called CMP.

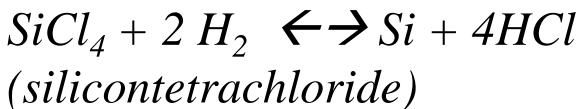
• The polishing process usually involves two or three polishing steps with progressively finer slurry and intermediate cleanings using RO/DI water.

• An SC1 solution (ammonia, hydrogen peroxide and RO/DI water) is used for final cleaning to remove organic impurities and particles. Next, natural oxide and metal impurities are removed with HF and finally SC2 solution causes super clean new natural oxides to grow up on the surface.

V. Wafer epitaxial processing

• A process called epitaxy (EPI) is used to grow a layer of single crystal silicon from vapor onto a single crystal silicon substrate at high temperatures.

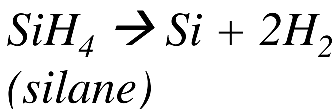
• The growth of a single crystalline silicon layer from the vapor phase is called vapor-phase epitaxy (VPE).



$SiCl_4 + 2\,H_2 \leftrightarrow Si + 4HCl$

(silicontetrachloride)

The reaction is reversible i.e. if HCl is added Si is etched from the surface of the wafer. Another non-reversible reaction that produces Si is,



$SiH_4 \rightarrow Si + 2H_2$

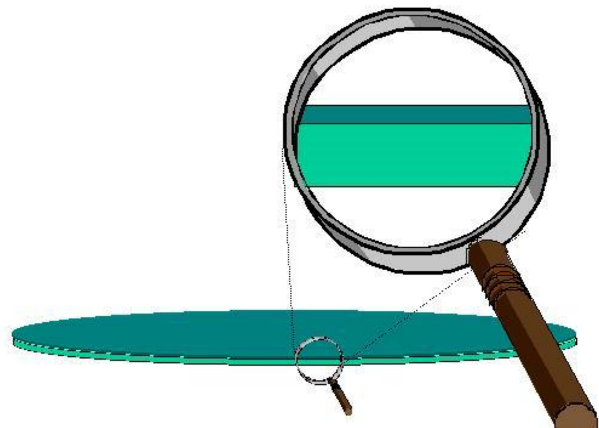
(silane)

Epitaxial Reactor
(Mitsubishi Materials Silicon)

• The purpose of EPI growth is to create a layer with different, usually lower, concentration of electrically active dopant on the substrate. For example, an n-type layer on a p-type wafer.

• Approx. 3% of wafer thickness.

• Contamination free for the subsequent construction of tranisistors.



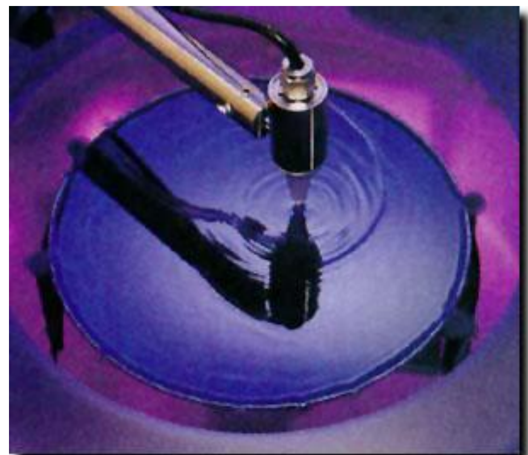
The Semiconductor Manufacturing Process
- Photolithography

I. Photoresist coating

• Photoresist is a photo-sensitive material applied to the wafer in a liquid state in small quantities. The wafer is spun at 1000 to 5000 rpm which spreads the “puddle” into a uniform layer between 2 and 200 μm thick.
• There are two types of photoresist: negative and positive.
- positive – exposure to light breaks down complex molecular structure, making it easy to be dissolved.
- negative – exposure to light causes molecular structure to become more complex and more difficult to be dissolved.

The steps involved in each photolithography step are as follows;

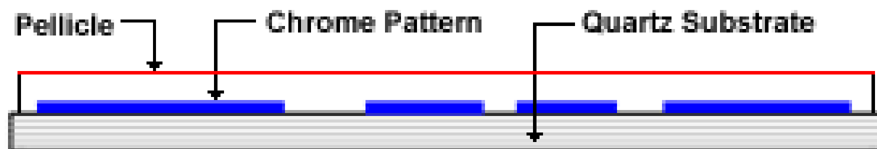
- clean wafers
- deposit barrier layer SiO_2, Si_3N_4, Metal
- coat with photoresist
- soft bake
- align masks
- expose pattern
- develop photoresist
- hard bake
- etch windows in photoresist
- remove photoresist



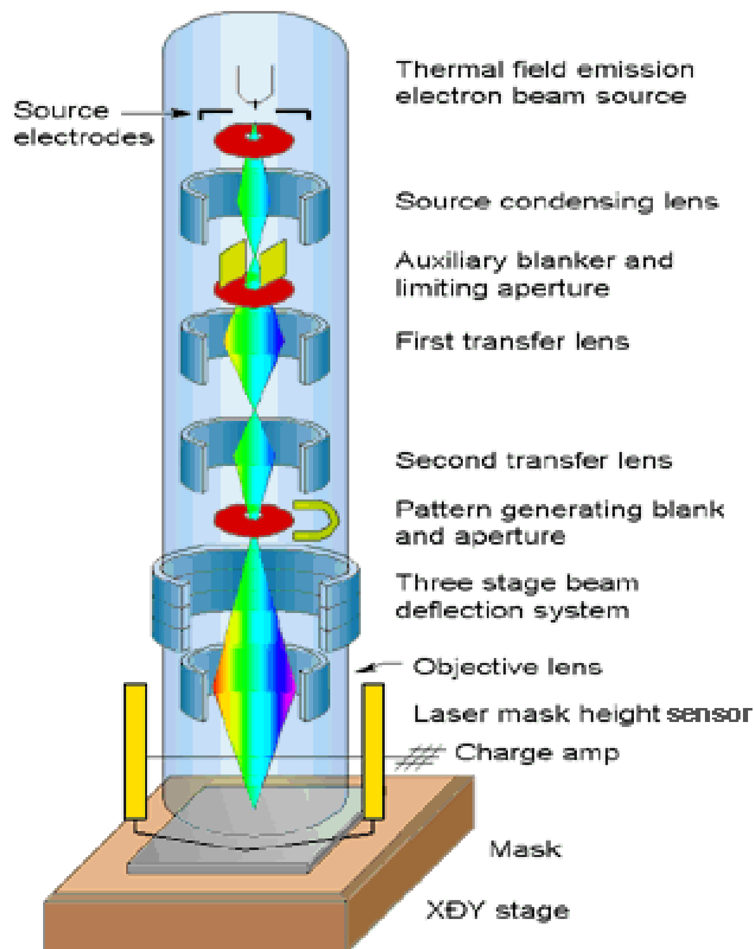
Photoresist Application
(Ontrak)

II. Pattern Preparation

• IC designers design the pattern for each layer using CAD software. The pattern is then transferred to an optically clear quartz substrate (reticle) with a chrome pattern using a laser pattern generator or an e-beam.



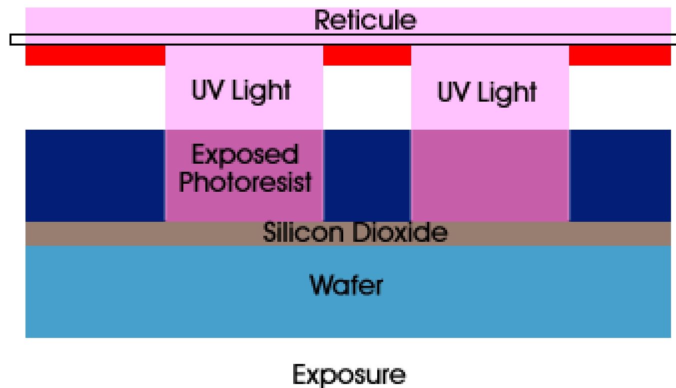
Reticle



E-Beam Pattern Generator
(Etec Corporation)

III. Device layer pattern transfer

• Applying and exposing photoresist to create a device layer on an actual wafer is similar to the process used to create reticles. For actual production, a tool called stepper is used.

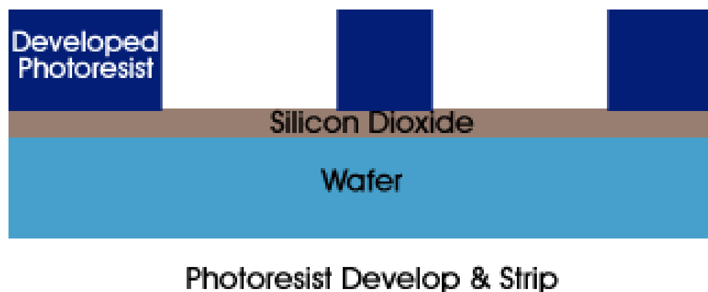


Exposure

Stepper
(ASM Lithography)

III. Develop and Bake

• After exposure, wafers are developed in either an acid or base solution to remove the exposed areas of photoresist.
• Once the exposed photoresist is removed, the wafer is "soft-baked" at a low temperature to harden the remaining photoresist.



Photoresist Develop & Strip

* Dust particles are the main concern in the photolithography process. Room air quality is measured by Class i.e. a Class 10 room has less than 10 dust particles of size greater than 0.5μm per cubic foot of air. In order to avoid contamination of the wafer surface with dust particles wafer processing is carried out in clean rooms.

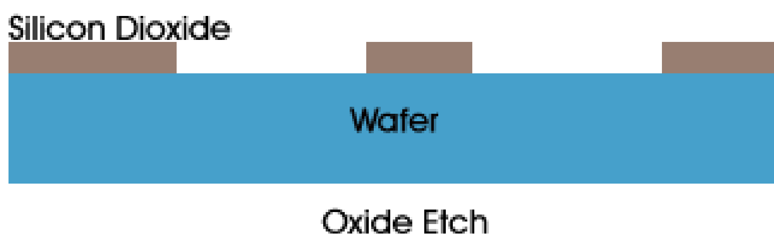
The Semiconductor Manufacturing Process
- Etching and Ion Implantation

I. Wet and Dry Etch

- Etching with chemicals takes place at large wet benches.
- Different types of acid, base and caustic solutions are used for removing selected areas of different material.
- BOE, or buffered oxide etch, prepared from hydrofluoric acid buffered with ammonium fluoride is used to remove silicon dioxide without etching away underlying silicon or polysilicon layer.
- Phosphoric acid is used to etch silicon nitride layers.
- Nitric acid is used to etch metals.
- Sulfuric acid is used to remove photoresist.
- For dry etch, the wafer is placed into an etching chamber and etching is done by plasma.
- Personnel safety is a primary concern.
- Many fabs use automated equipment perform the etching process.

II. Photoresist strip

- The photoresist is then completely stripped off the wafer, leaving the oxide pattern on the wafer.



Oxide Etch

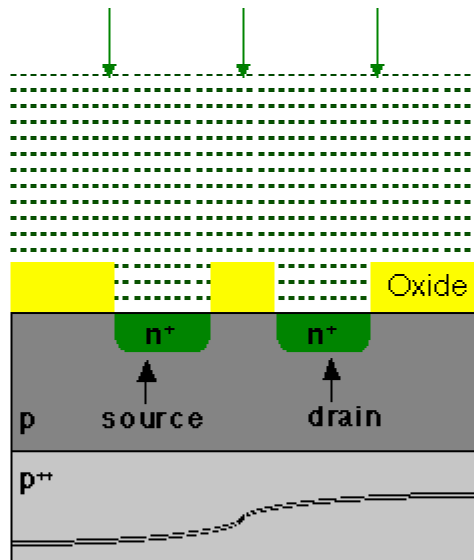
III. Ion Implantation

• Ion implantation changes the electrical characteristics of precise areas within an existing layer on the wafer.

• An ion implanter uses a high-current accelerator tube and steering and focusing magnets to bombard the surface of the wafer with ions of a particular dopant.

• For the MOSFET example, the oxide acts as a barrier when dopant chemicals are deposited on the surface and diffused into the surface.

• Annealing is done by heating the silicon surface to 900°C. The implanted dopant ions diffuse further into the silicon wafer.



The Semiconductor Manufacturing Process

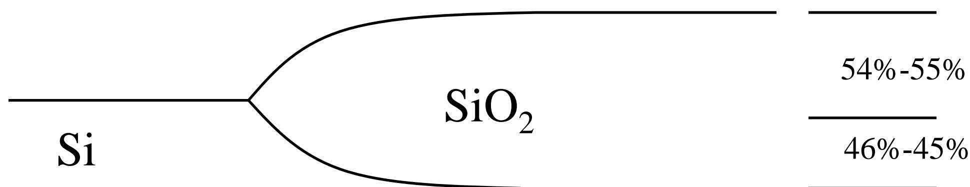
- Thin Film Deposition

I. Silicon oxidation

SiO_2 grows thermally when silicon is in the presence of oxygen. Oxygen comes from oxygen gas or water vapor. A temperature of 900 to 1200 ^{o}C is required.

The chemical reactions that occur are
- $Si + O_2 \rightarrow SiO_2$
- $Si + 2H_2O \rightarrow SiO_2 + 2H_2$

- The surface of the silicon wafer after selective oxidation will appear as follow,



- Both oxygen and water will diffuse through the existing SiO_2 and combine with Si to form additional SiO_2. Water (steam) diffuses easier than oxygen, hence there is a much faster growth rate with steam.
- Oxide is used to provide insulating and passivation layers and form transistor gates. Dry O2 is used to form gates and thin oxide layers. Steam is used to form thick oxide layers. Insulating oxide layers are usually about 1500 Å and gate layers are usually between 200 Å to 500 Å.

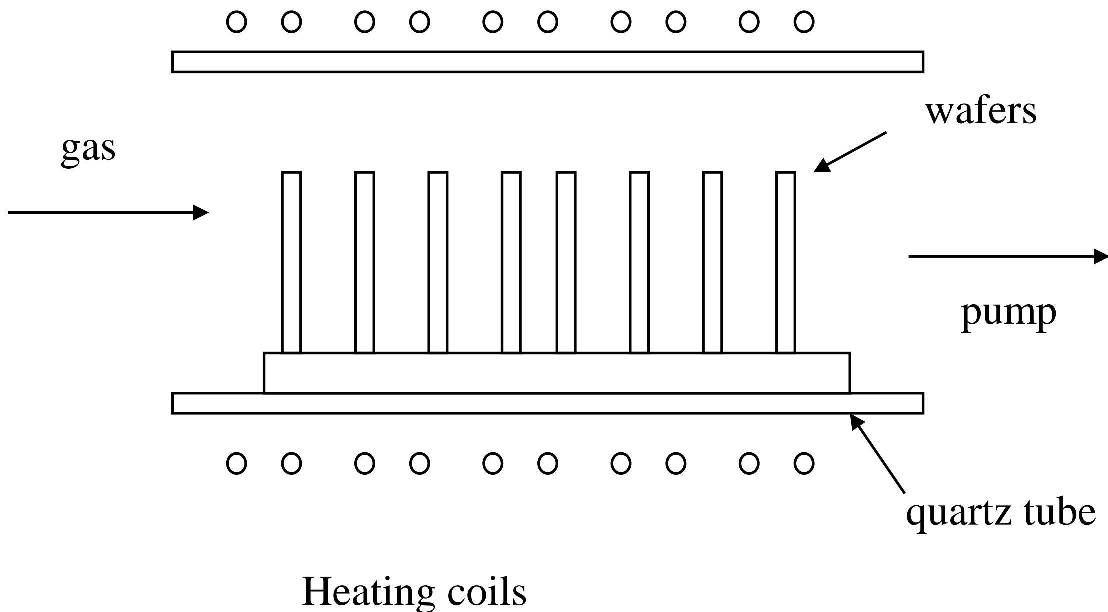
II. Chemical Vapor Deposition

• Chemical Vapor Deposition (CVD) forms thin films on the surface of the substrate by either thermal decomposition and/or reaction of gaseous compounds.

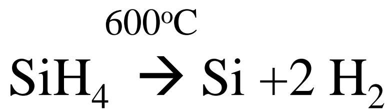
• There are three basic types of reactors for CVD,

- atmospheric chemical vapor deposition
- low pressure CVD (LPCVD)
- plasma enhanced CVD (PECVD)

• A sketch of a Low Pressure CVD process is show below,



i) Polysilicon

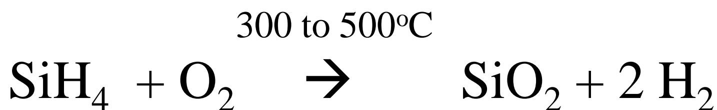


$$SiH_4 \xrightarrow{600^oC} Si + 2\,H_2$$

Deposits 100 to 200 Å /min

- Phosphorous (phosphine), Boron (Diborane) or Arsenic gases can be addded. Polysilicon can also be doped with diffusion gases after it has been deposited.

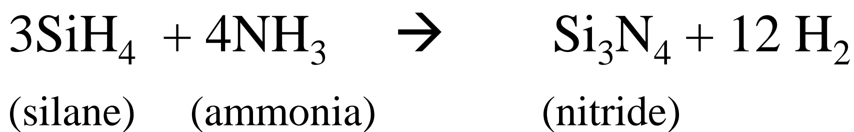
ii) Silicon Dioxide



$$SiH_4 + O_2 \xrightarrow{300\text{ to }500^oC} SiO_2 + 2\,H_2$$

- SiO2 is used as an insulator or passivation layer. Usually phosphorous is added to give better 'flow' properties.

iii) Silicon Nitride



$$3SiH_4 + 4NH_3 \rightarrow Si_3N_4 + 12\,H_2$$

(silane) (ammonia) (nitride)

III. Sputtering

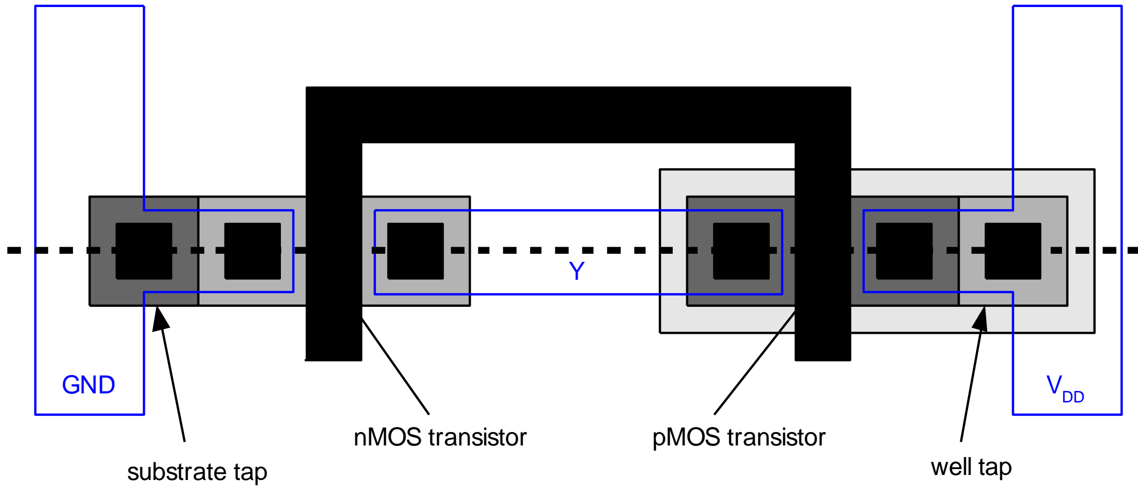
If a target is bombarded with high energy ions such as Ar+ then atoms in the target will be dislodged and transported to the substrate.

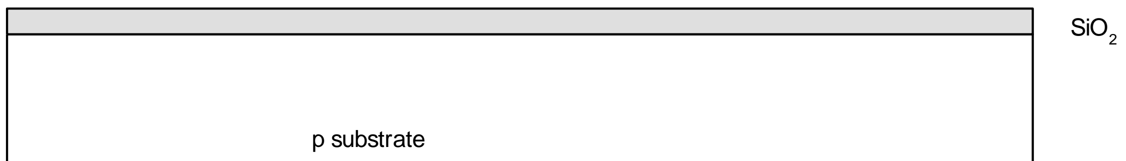
Metals such as Al, Ti can be used as a target.

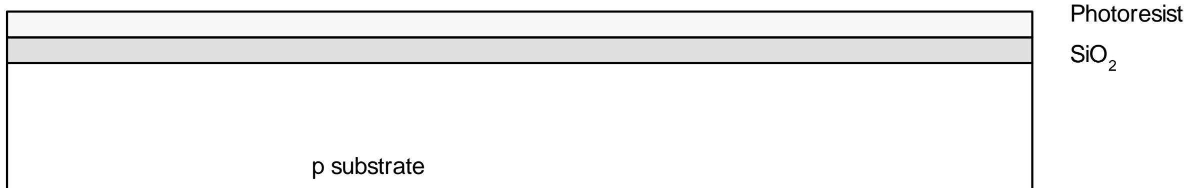
IV. Evaporation

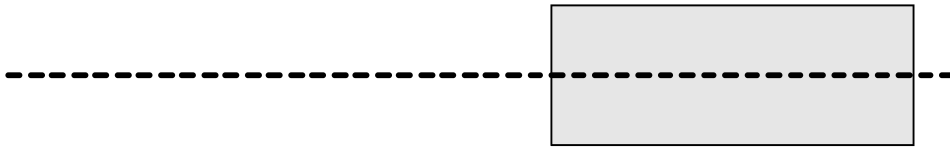
If Al or Au (gold) is heated to the point of evaporation then the vapor will condense and form a thin film that covers the surface of the wafer.

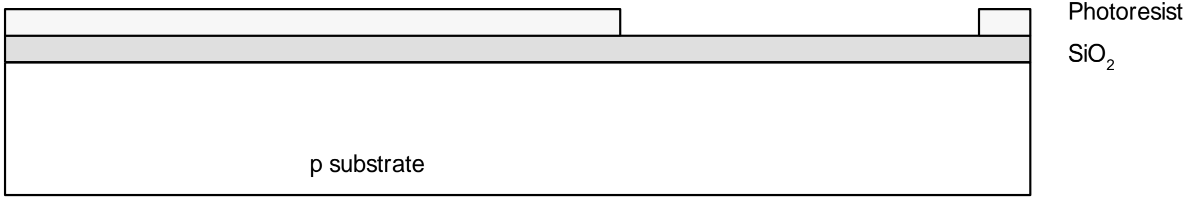
Example: Inverter Mask Set

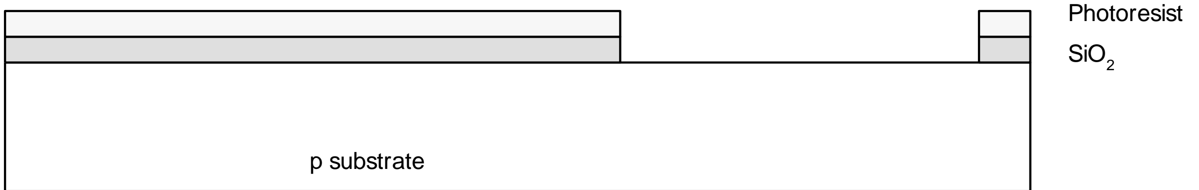


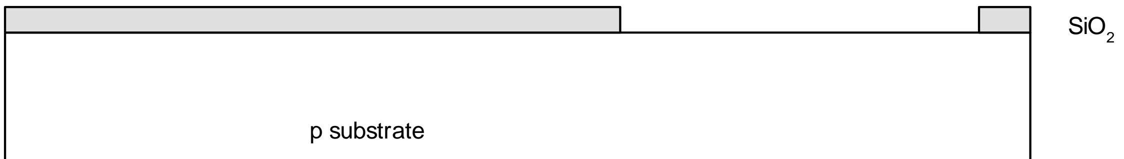


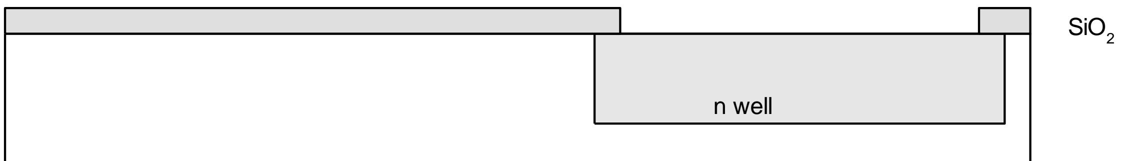


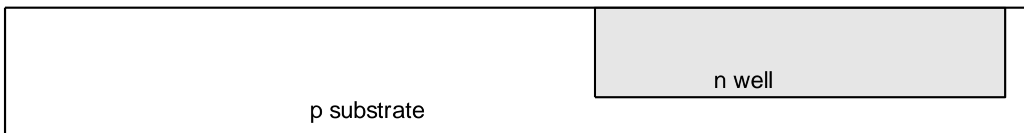



Photoresist
SiO2
p substrate


Photoresist
SiO2
p substrate

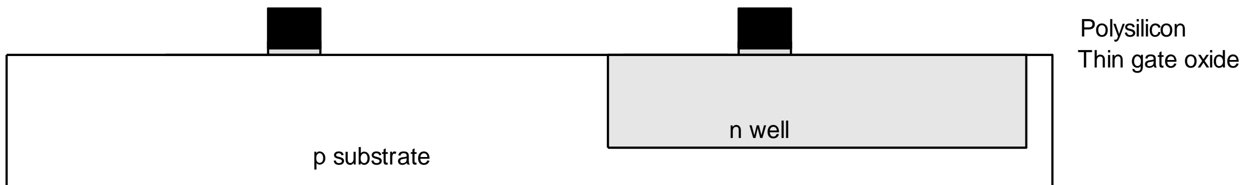

SiO2
p substrate

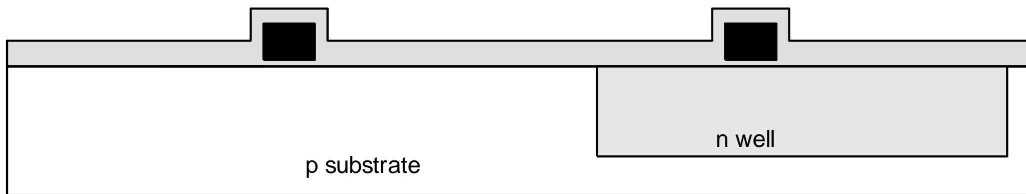

SiO2
n well

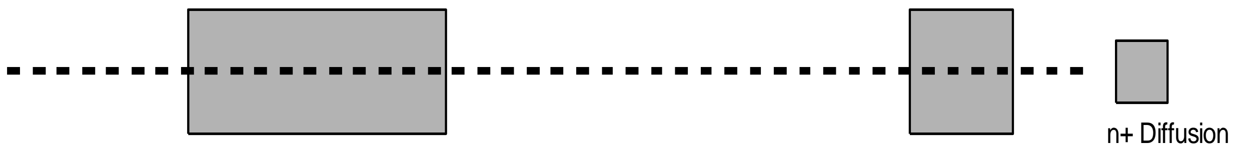

n well
p substrate

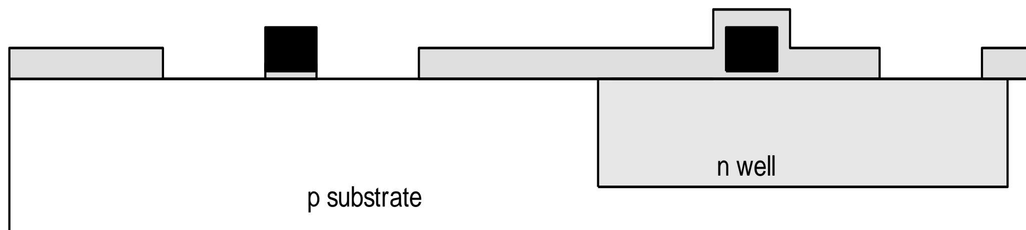
Polysilicon
Thin gate oxide
n well
p substrate

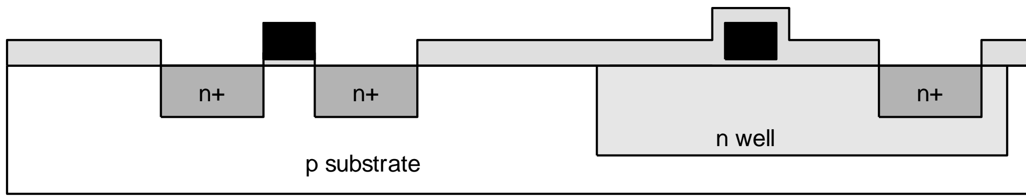
Polysilicon

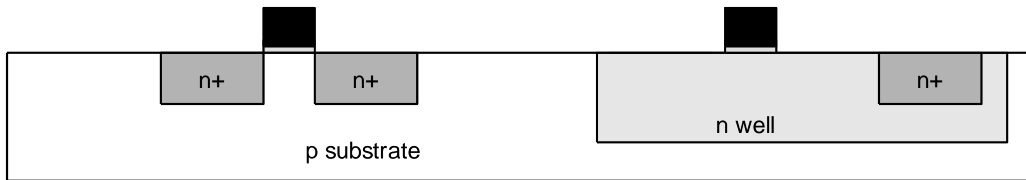

Polysilicon
Thin gate oxide
n well
p substrate

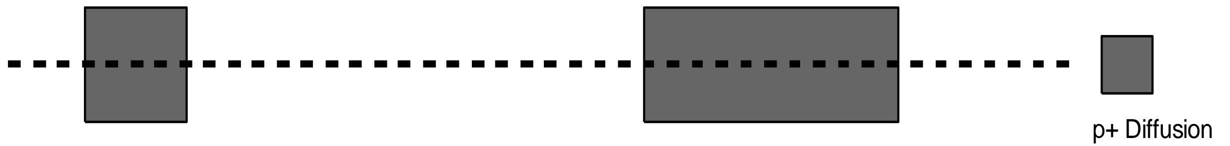

n well
p substrate

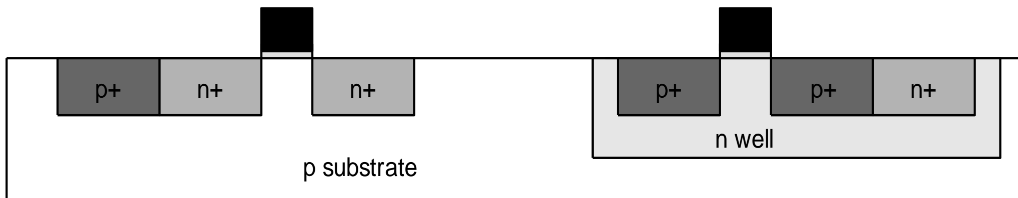

n+ Diffusion

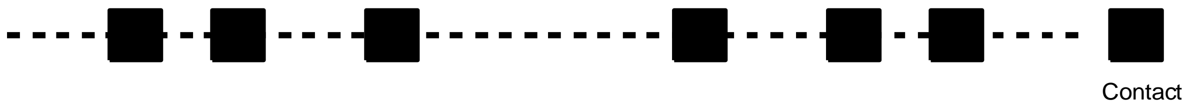

n well
p substrate

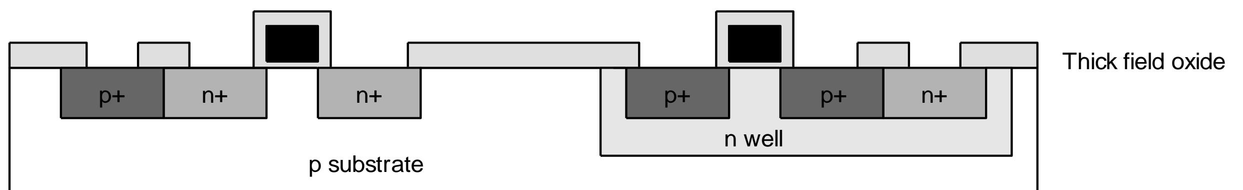

n+
n+
n+
n well
p substrate

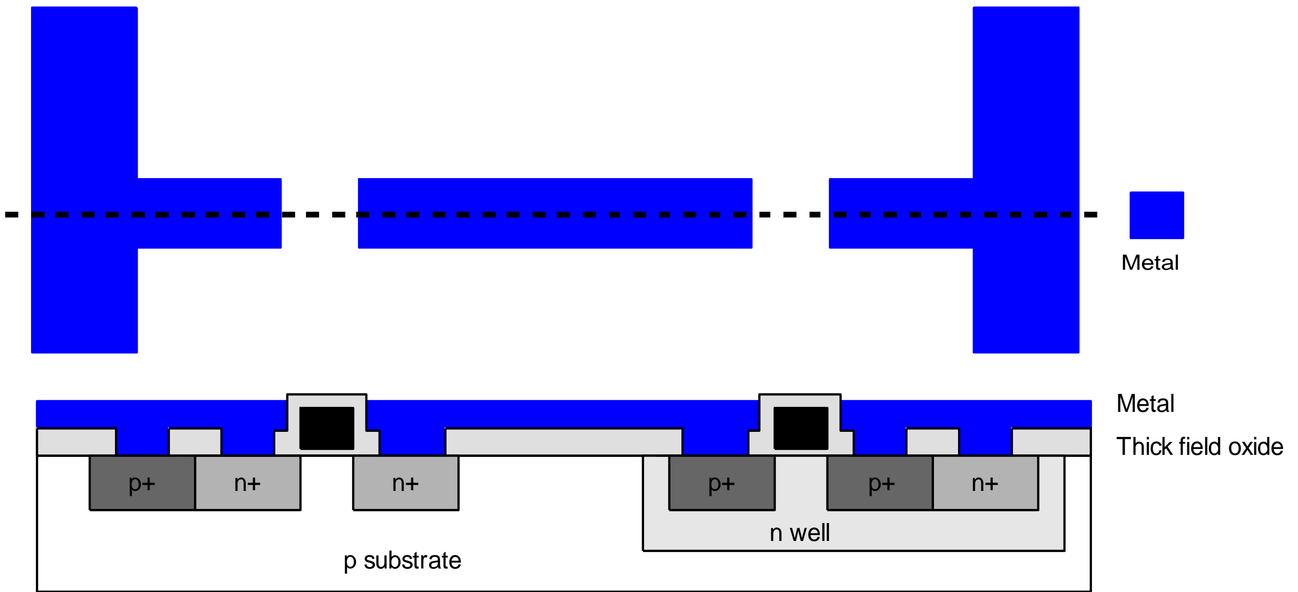

n+
n+
n+
n well
p substrate


p+ Diffusion


p+
n+
n+
p+
p+
n+
n well
p substrate


Contact


Thick field oxide
p+
n+
n+
p+
p+
n+
n well
p substrate



- Six masks
  - n-well
  - Polysilicon
  - n+ diffusion
  - p+ diffusion
  - Contact
  - Metal

n well

Polysilicon

n+ Diffusion

p+ Diffusion

Contact

Metal

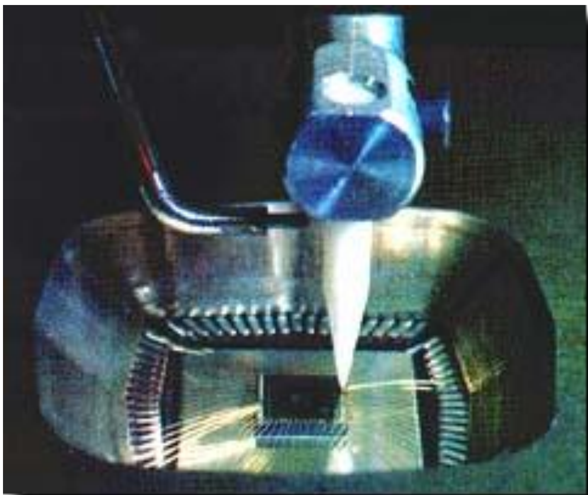
The Semiconductor Manufacturing Process
- Post-processing

I. Probe Test and Wafer Dicing

• After the final passivation layer and backside prep, automated methods are used to test the device on the wafer.
• A probe tester is used to check the operation of the device. Devices that fail the test are marked with colored dye.
• After probe test, the wafer is diced into individual die.

II. Wire Bonding and Packing

• Individual devices are attached to a lead frame and aluminum or gold leads are attached via thermal compression or ultrasound welding.
• The packaging is completed by sealing the device into a ceramic or plastic package.



Wire Bonding
(Kulicke & Soffa Industries, Inc.)

Wire Bonding
(Kaijo Corporation)

Packaging