

# Lecture 23

## Multiple State Machine Implementation & Clock Period

M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

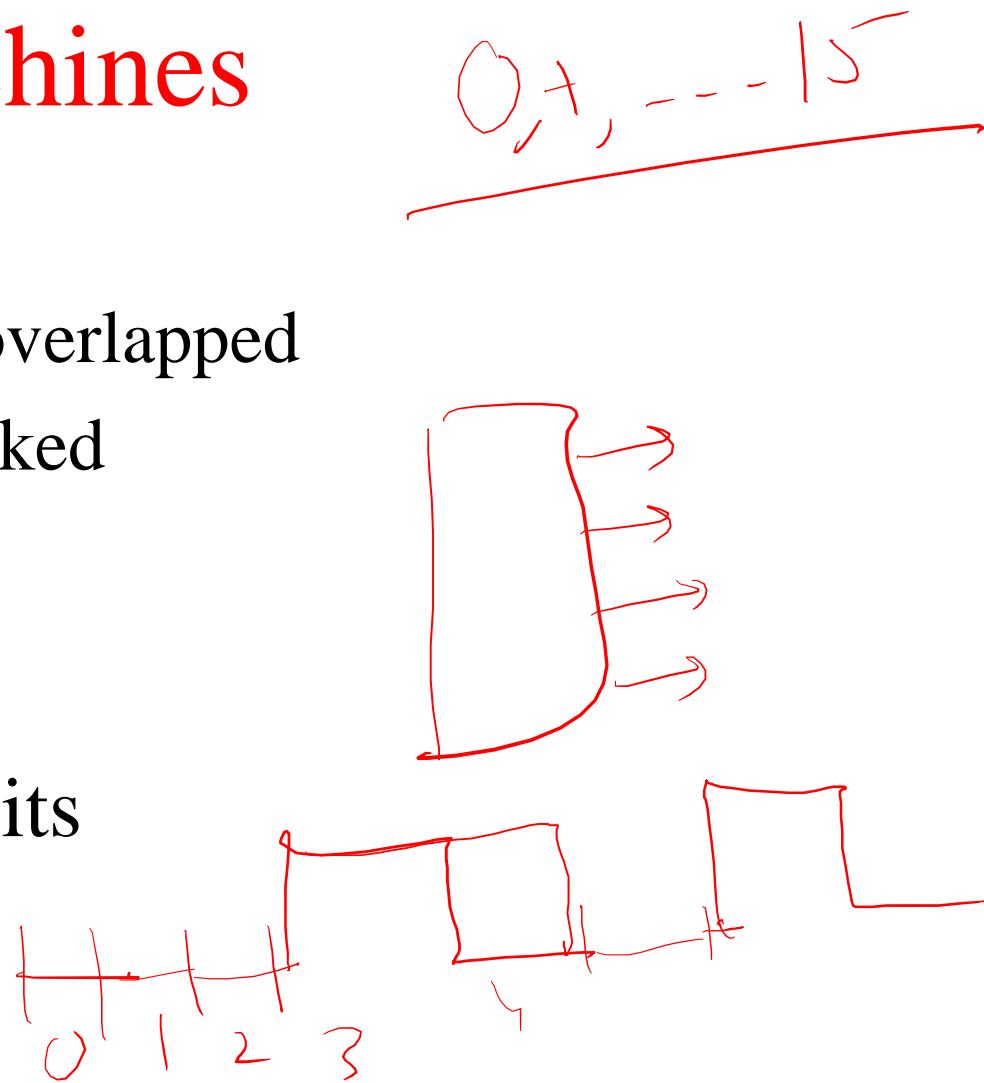
# Steps in State Machine Synthesis using Counters

- Encode the states
- Choose a counter with appropriate control inputs to implement the state register
- Use the counter functionality table to arrive at the spec. of the combinational logic
- Synthesize the combinational logic



# Applications of Sequential Machines

- Pattern matching
  - Overlapped or non-overlapped
  - Blocked or non-blocked
- Sequential decoding
- Controllers
- Memory based circuits

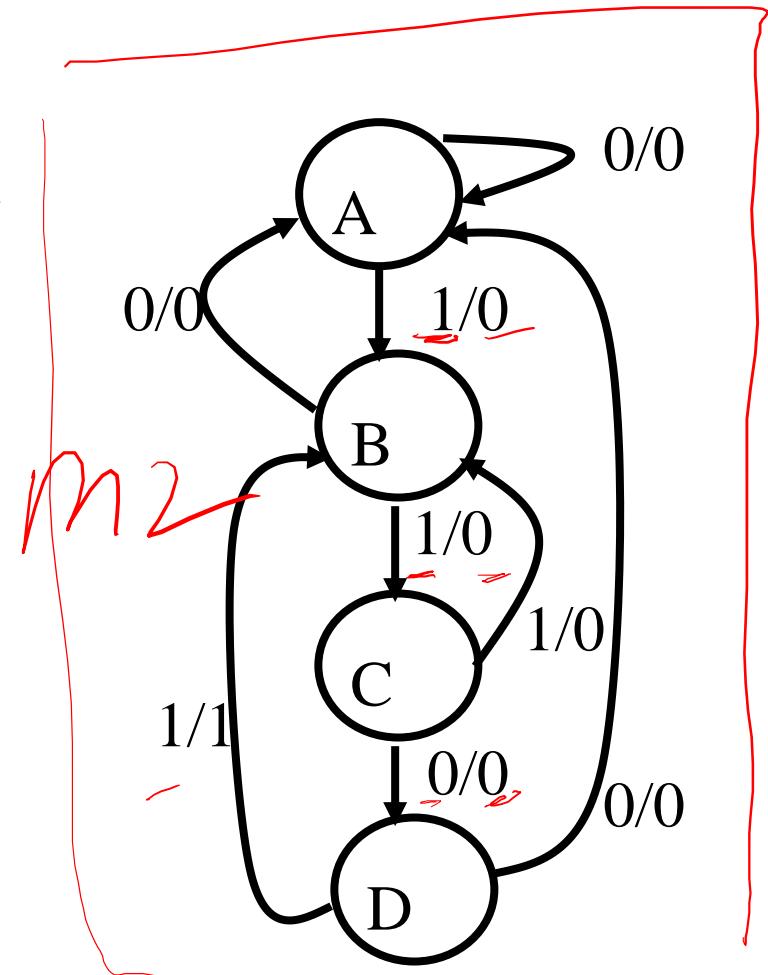
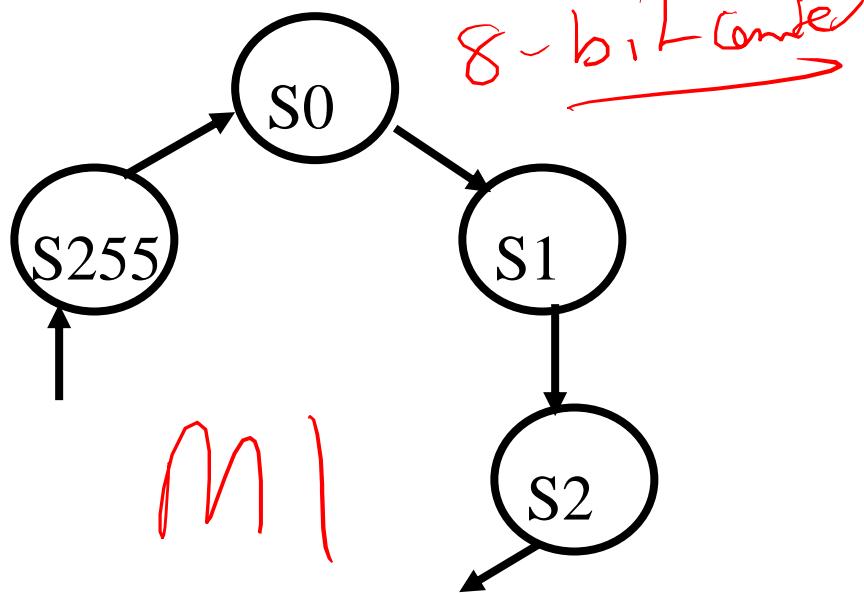


# Interacting State Machines : Example

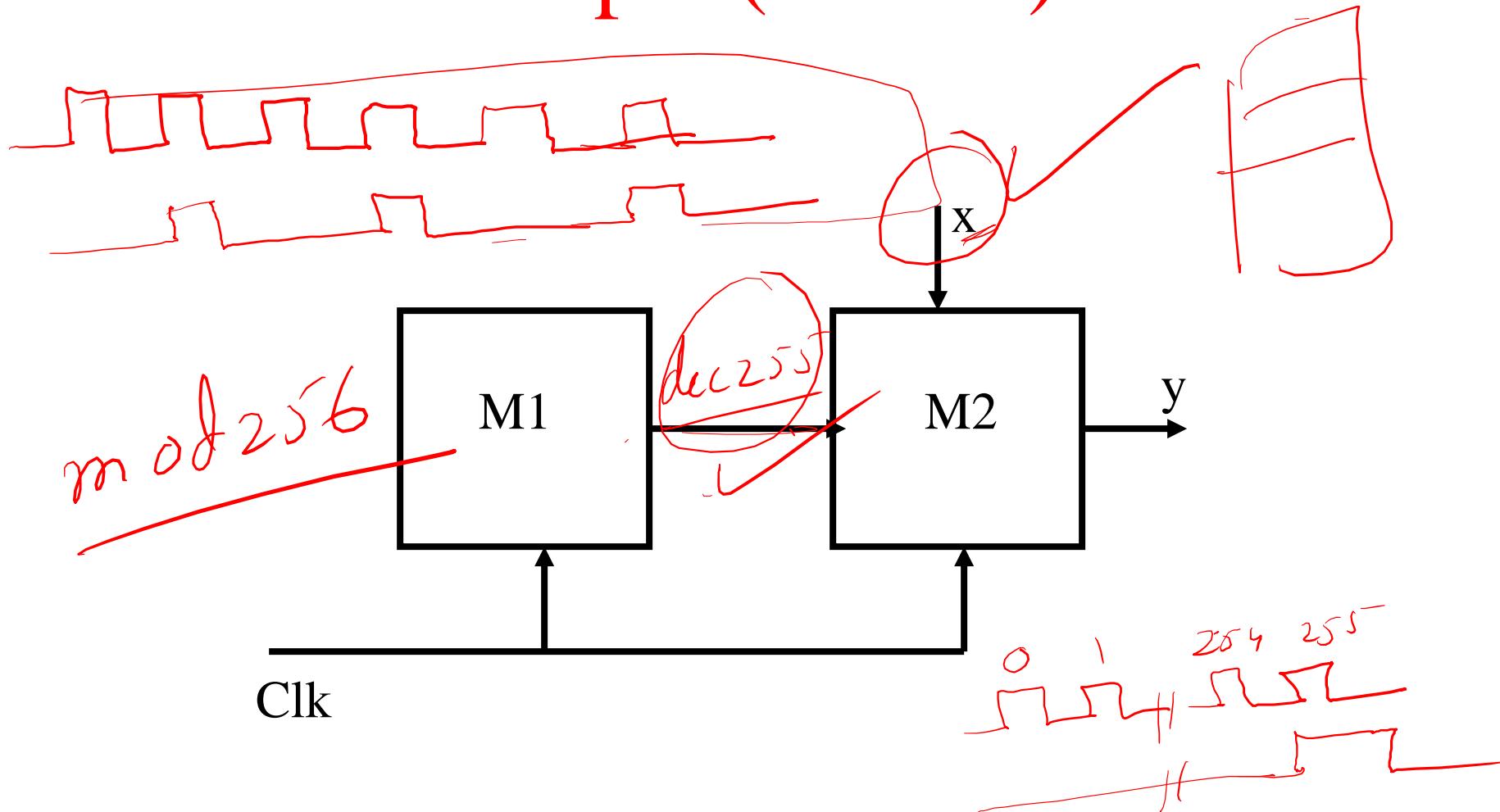
- Search for a pattern  $P = '1101'$  within blocks of 256 bits. The pattern should not cross block boundaries.
- Design two state machines  $M_1$  and  $M_2$ 
  - $M_1$  is a modulo 256 counter
  - $M_2$  is the pattern recognizer
- The 256th transition of  $M_1$  should initialize  $M_2$

Blocked  
Overlaps

## Example (Contd.)

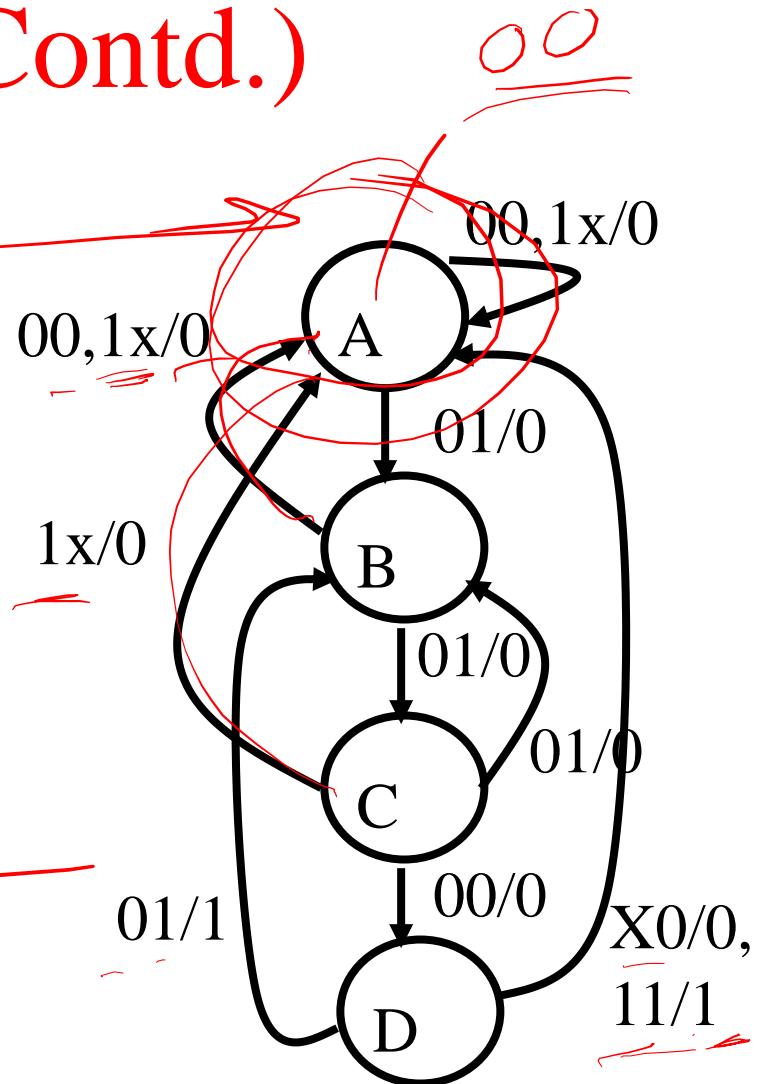
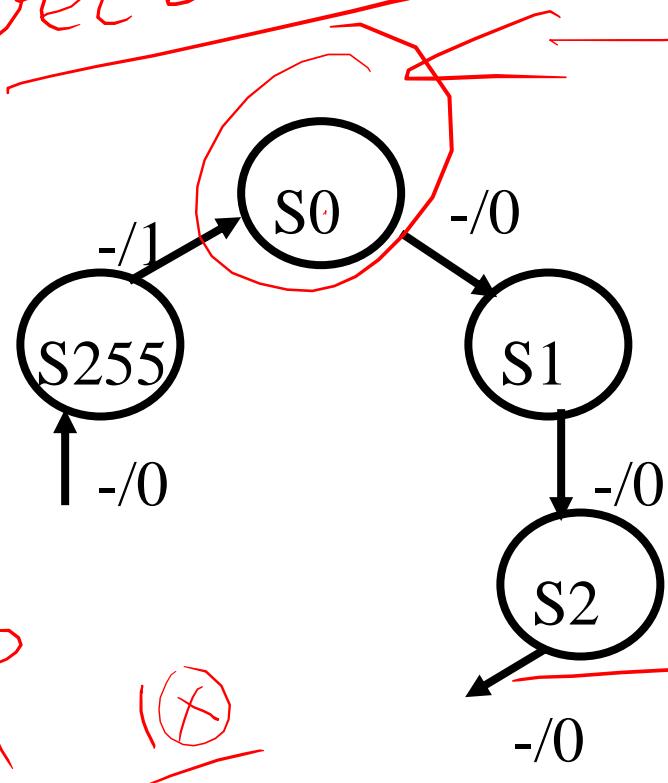


## Example (Contd.)



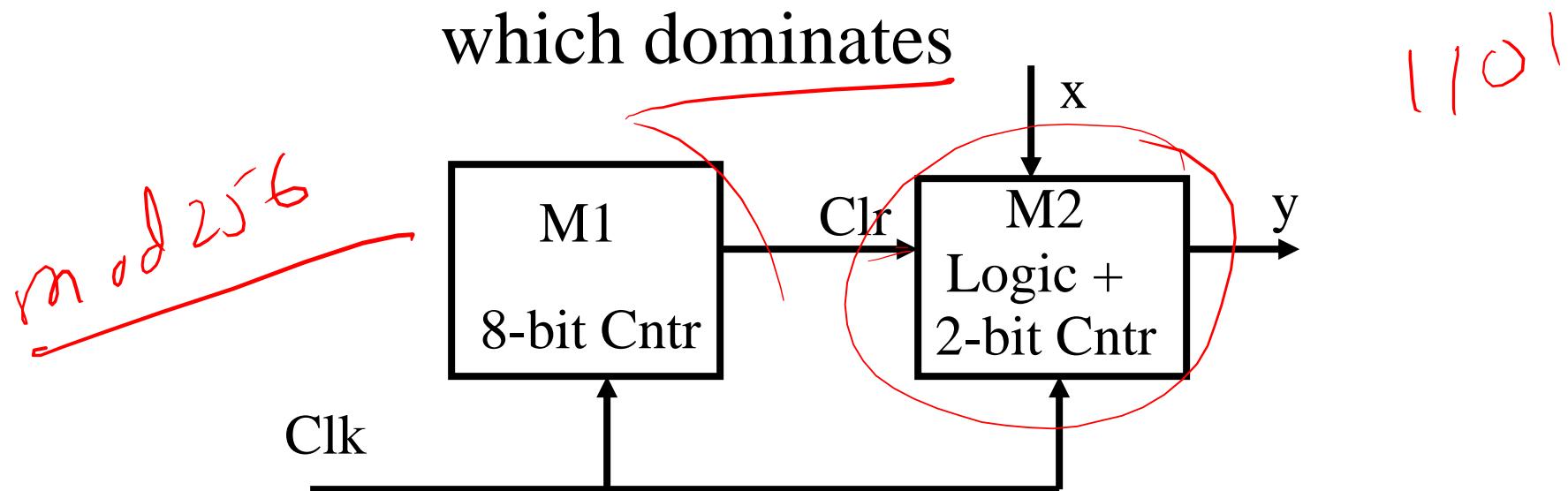
00  
00  
00  
00  
Dec 255

## Example (Contd.)



# Design Summary: Example

- M1 : 8-bit free running counter
- M2 : Counter with synchronous clear which dominates



$$Q = f(P_I, P_S)$$

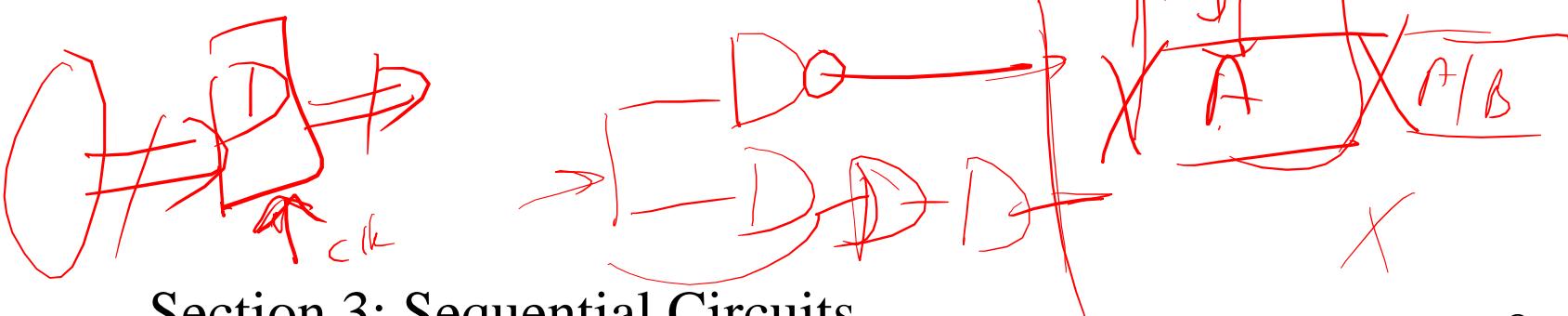
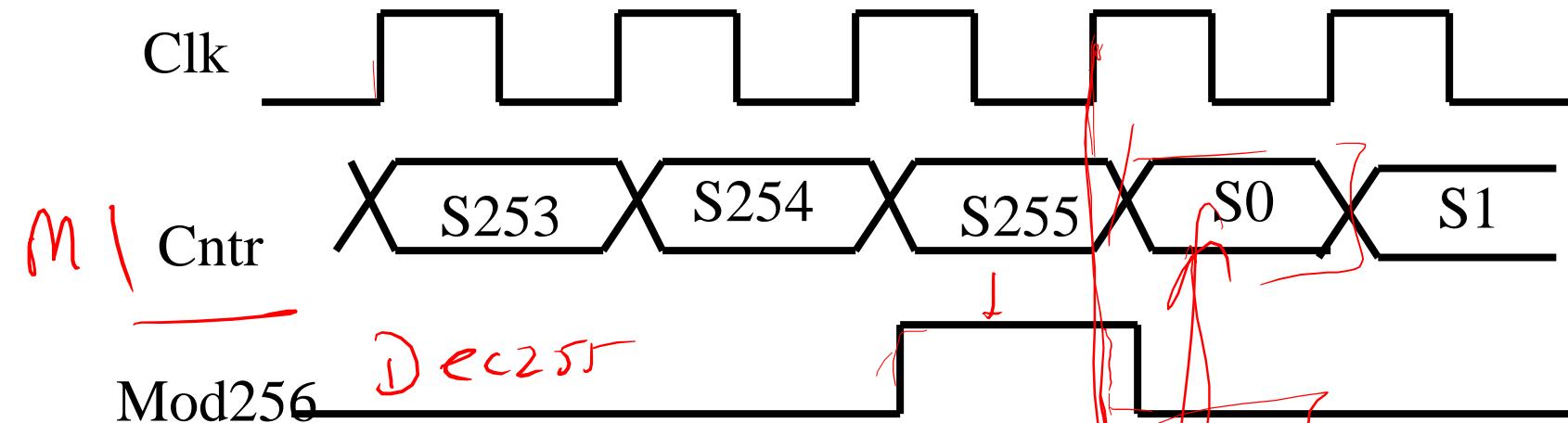
$$Q = f(P_S)$$

more

## Register & Latch Waveforms

$$NS = g(P_S, Q_S)$$

$$NS = \underline{f(P_S, P_I)}$$



Section 3: Sequential Circuits

# Multiple State Machines:

## Another Example

e-mail  
@

8-bit

- In a bit stream, count the number of “@” (ASCII Code) characters in blocks of 256 8-bit characters
- Three state machines: M1, M2 and M3
  - ✓ – M1: Pattern recognizer for “@” character
  - ✓ – M2: 8-bit counter for counting 256 characters
  - ✓ – M3: 8-bit Counter for counting no. of “@”

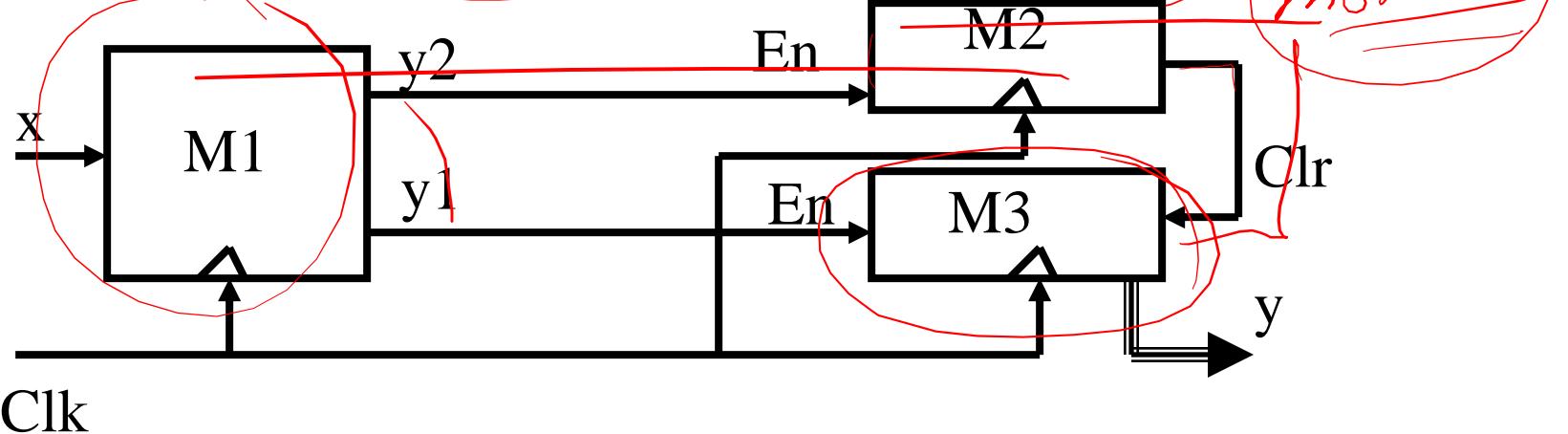
~~for ( ) { } if ( ) { } do { } while ( ) { }~~

## Second Example (Contd.)

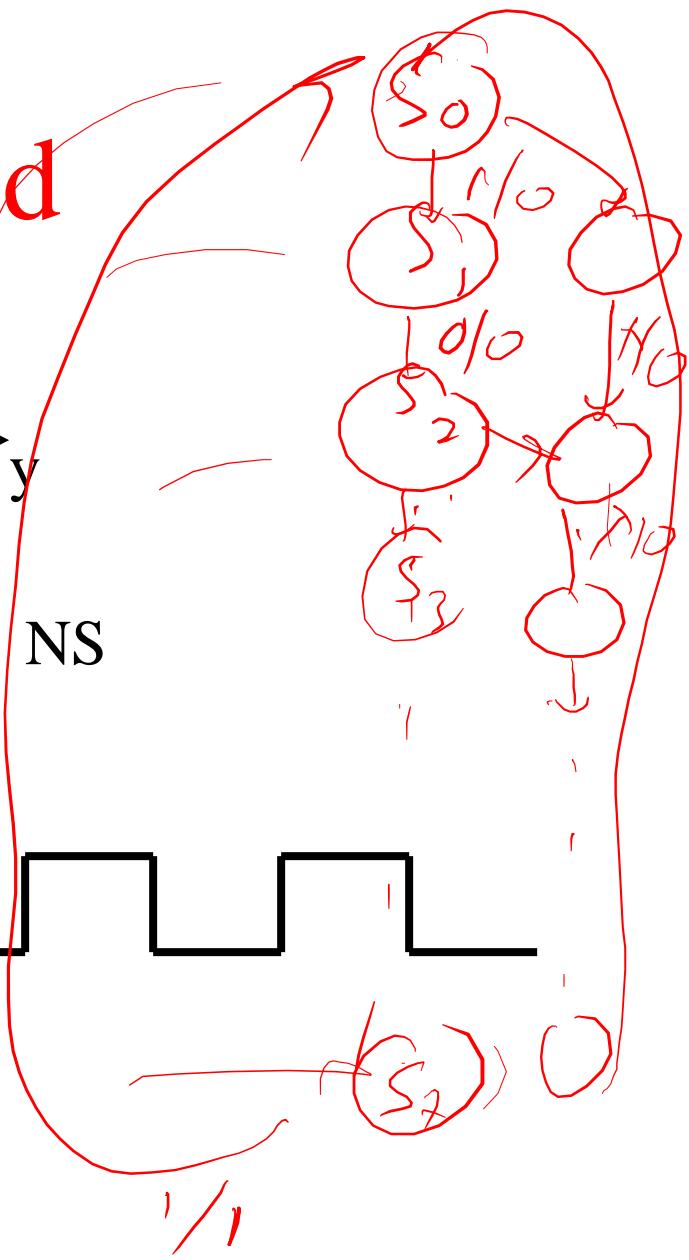
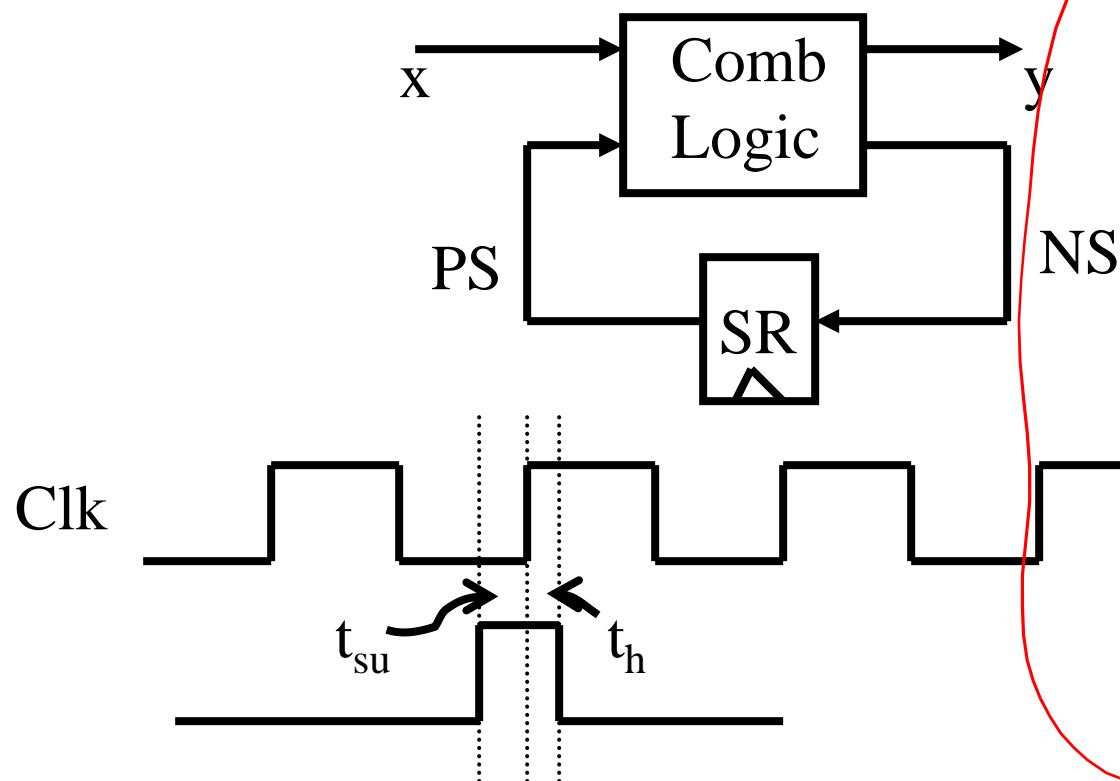
Specification of M1

~~y1 = 1 if  $\langle x(t-7) \dots x(t) \rangle = "@"$  and  $t \bmod 8 = 7$~~

~~y2 = 1 if  $t \bmod 8 = 7$~~



# Clock Period



# Clock Period Computation

$t_o$ : Critical path delay (x,PS) to y

$t_{ns}$ : Critical path delay (x,PS) to NS

$t_d$ : SR delay

$t_{su}$ : Setup time of the SR

$t_h$ : Hold time of the SR

$$t_{clk} \geq \max\{ t_d + t_o, t_d + t_{ns} + t_{su} \}$$

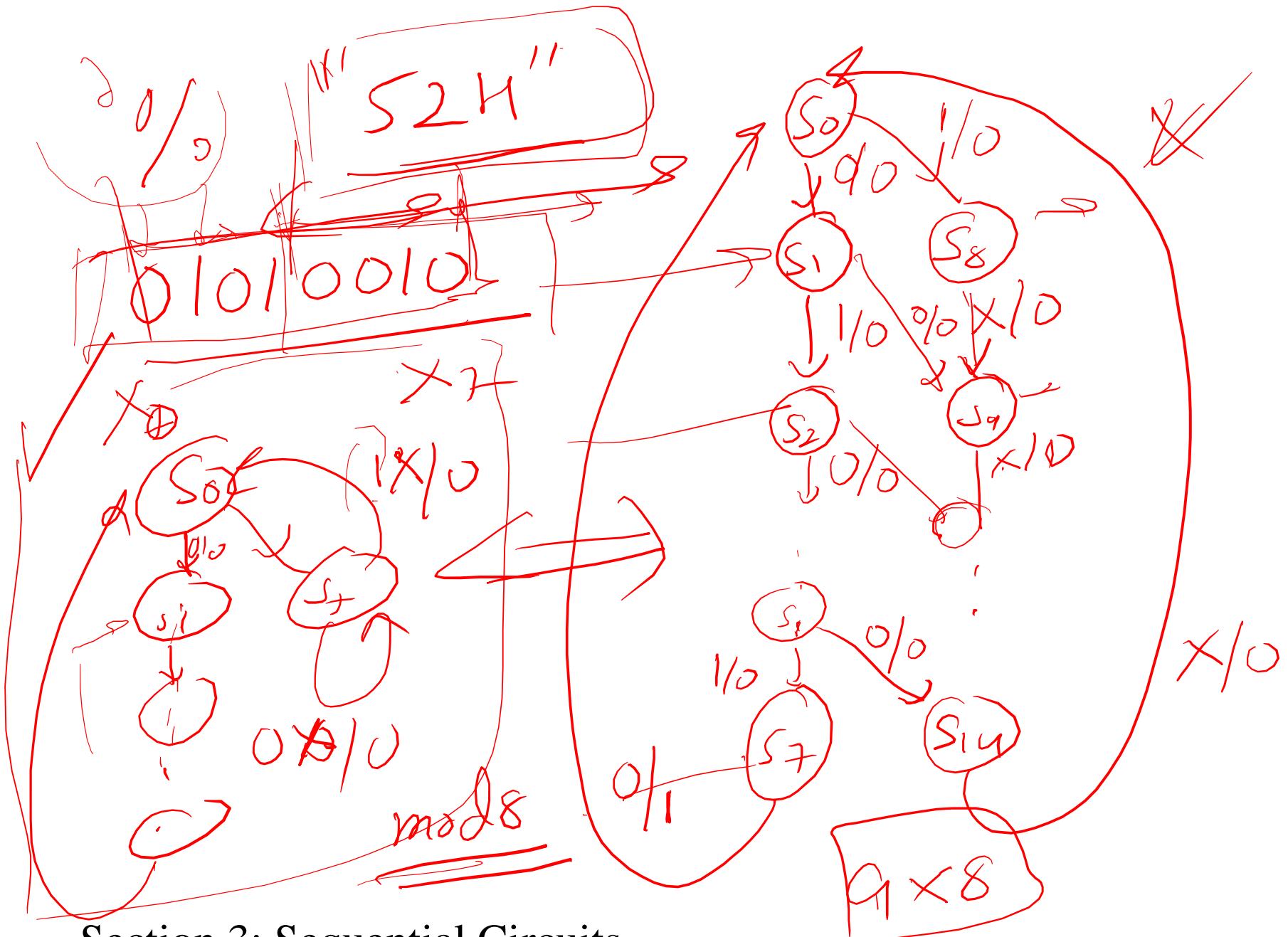
# Lecture 24

# System Design Case Studies

M. Balakrishnan

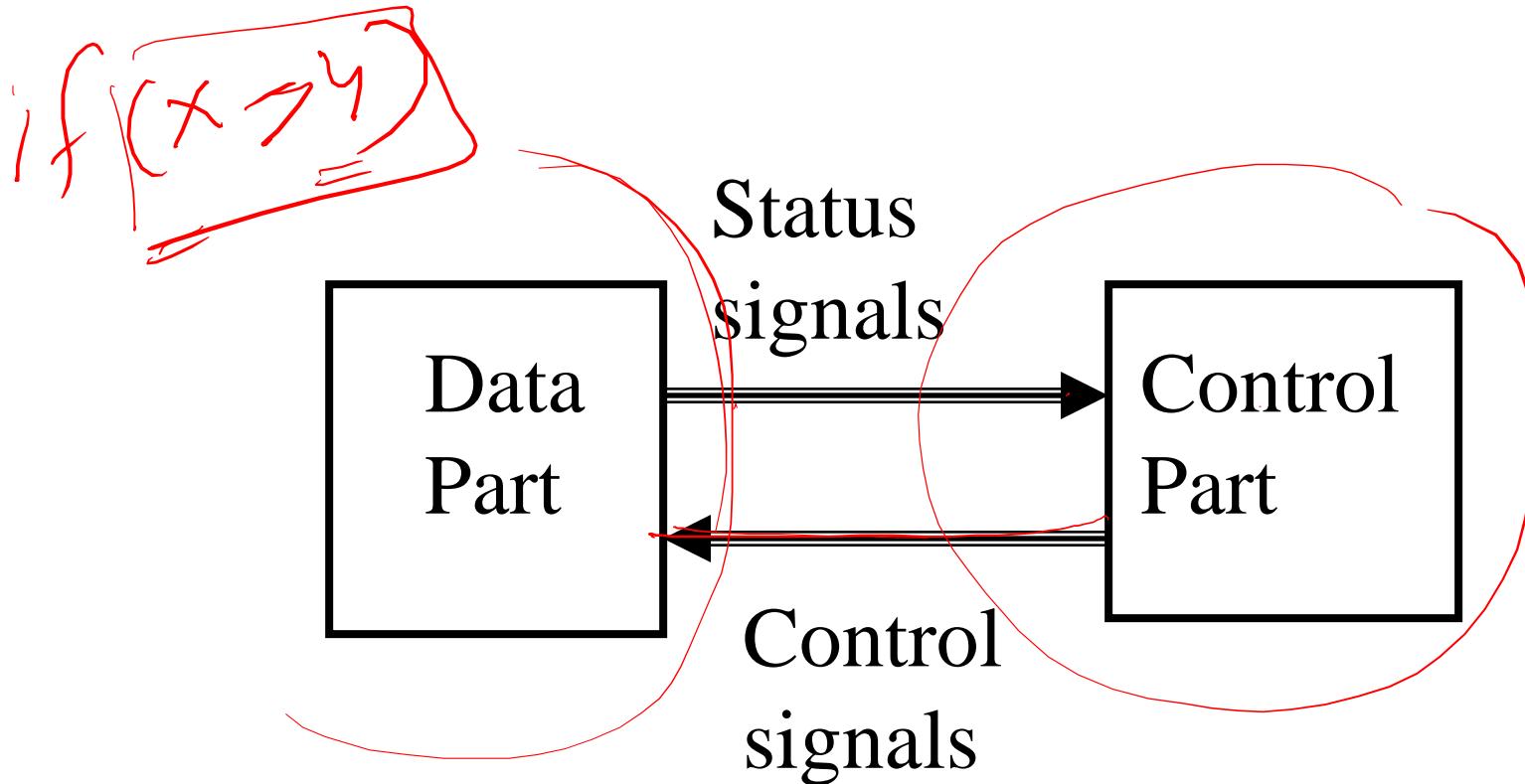
Dept. of Comp. Sci. & Engg.

I.I.T. Delhi



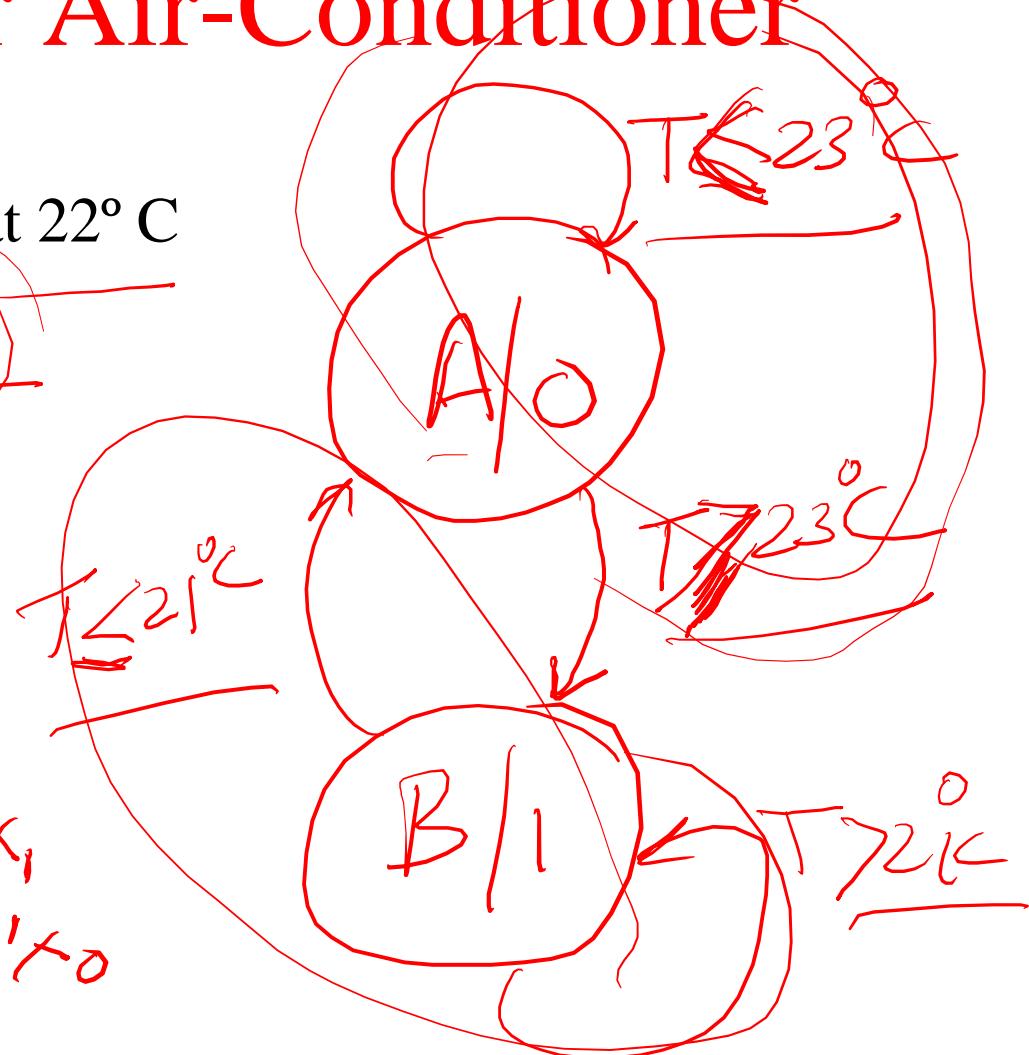
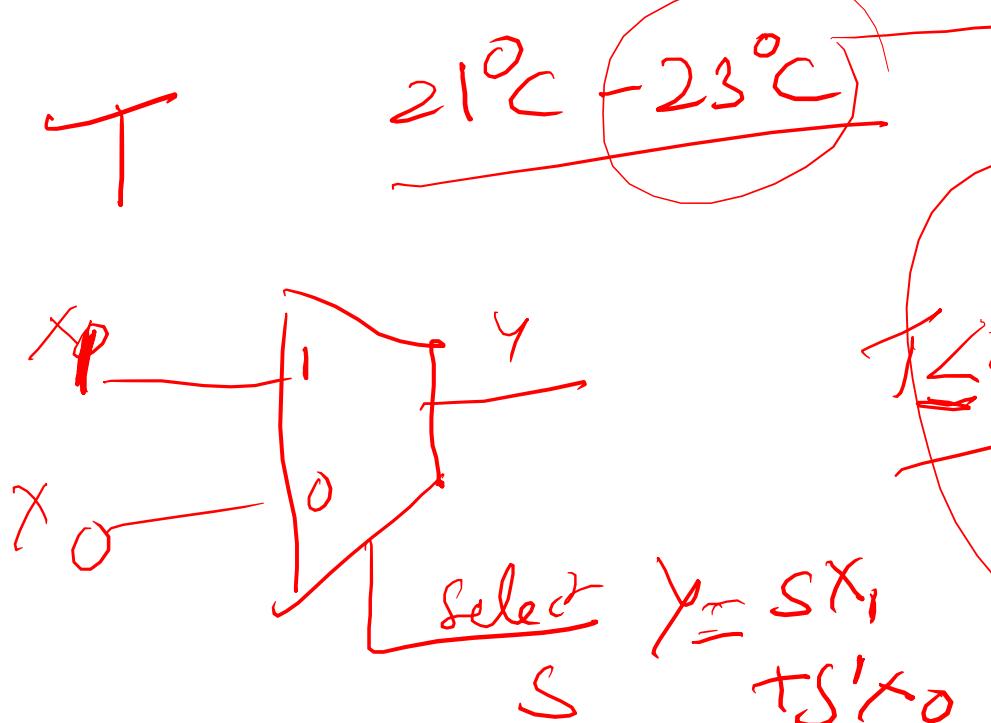
### Section 3: Sequential Circuits

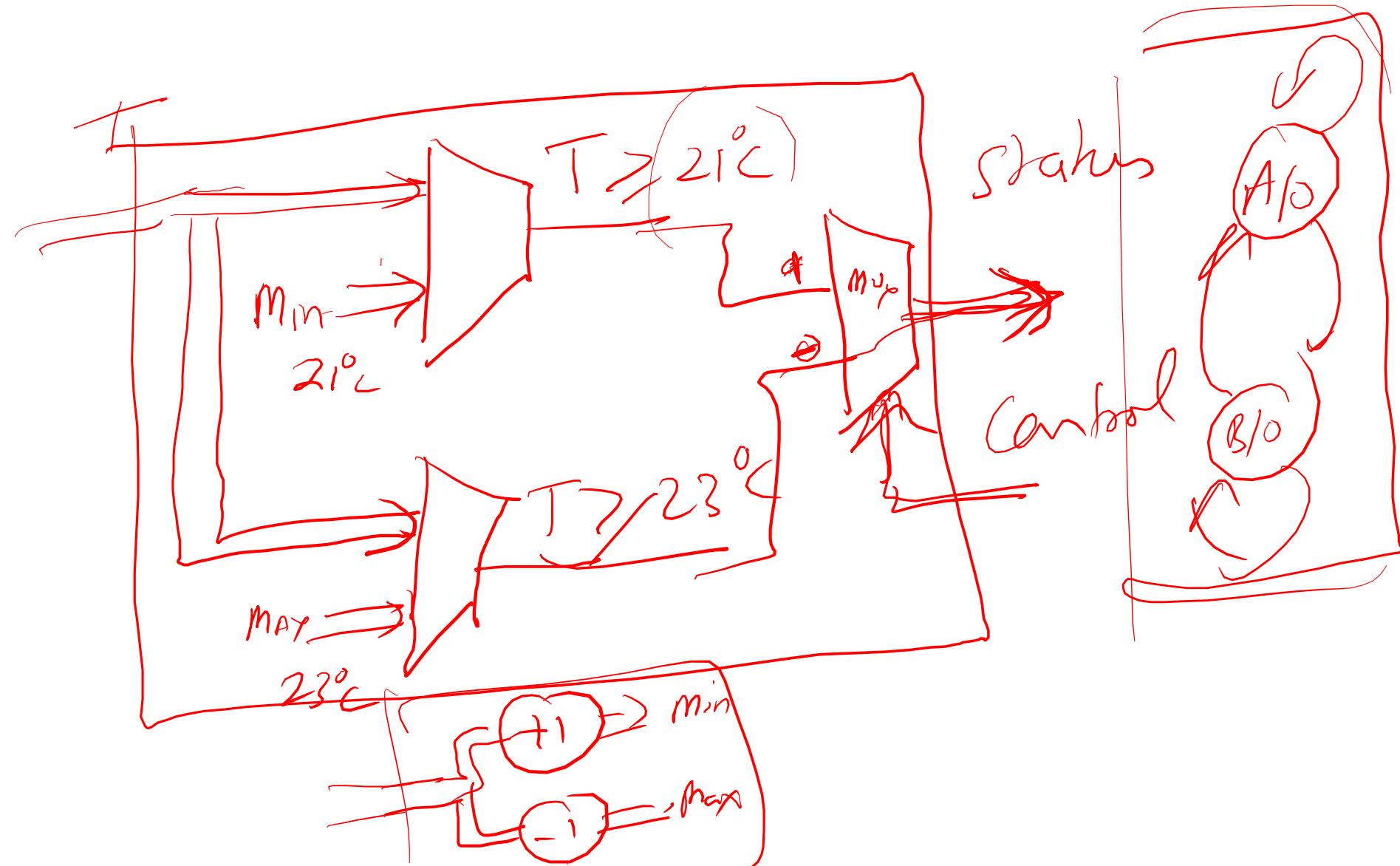
# Data-Control Partition



# Controller for Air-Conditioner

To maintain temperature at  $22^\circ \text{ C}$





### Section 3: Sequential Circuits



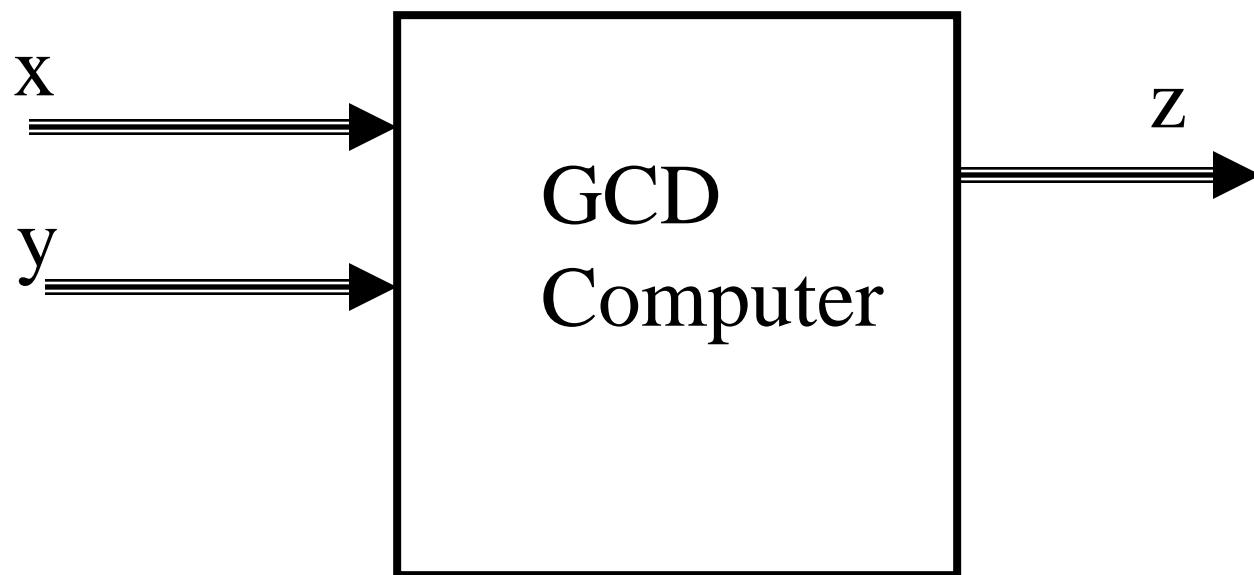
### Section 3: Sequential Circuits

# Steps in System Design

- Choose an algorithm
- Identify the data modules (operators & storage)
- Identify the control signals
  - Status signals
- Extract the state machine for control
- Implement the state machine to complete the design

# Case Study 1: GCD Computer

$$z = \gcd(x, y)$$



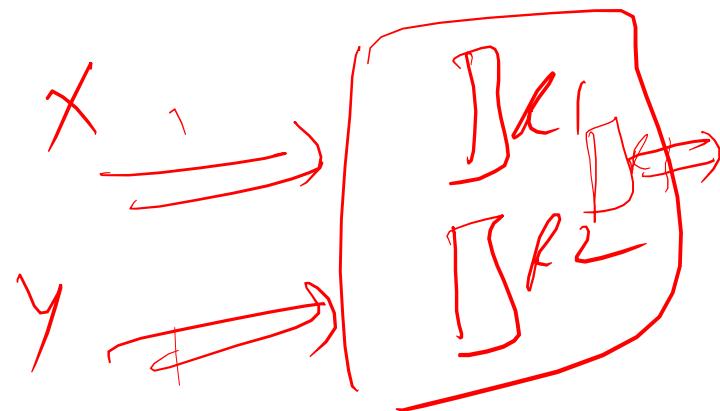
# GCD Algorithm

```
Input x, y;  
while ( x ≠ y ) do  
    if ( x > y ) then x := x - y  
    else y := y - x  
    endif;  
endwhile;  
z := x;  
end.
```

for each step

## Modified GCD Algorithm (RTL)

```
S0:R1:= x, R2:= y;  
S1:while ( R1 ≠ R2 ) do  
  S2: if ( R1 > R2 )  
  S3: then R1:= R1 - R2  
  S4: else R2:= R2 - R1  
  endif;  
endwhile;  
S5: R3:= R1;
```



# GCD Computer: Data Part

R1

R2

Comp

R3

SUB

# Lecture 25

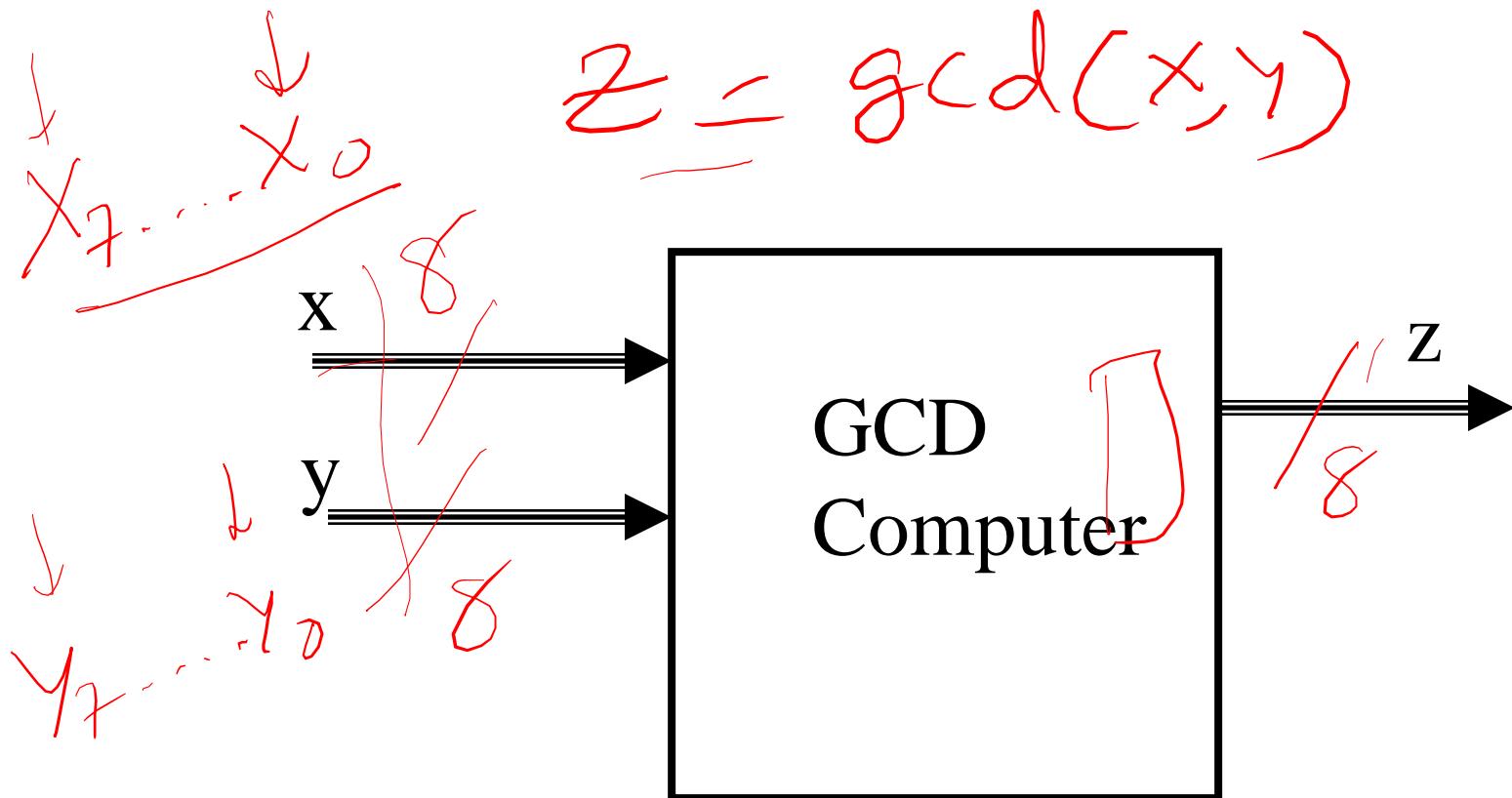
# System Design Case Studies

M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

# Case Study 1: GCD Computer



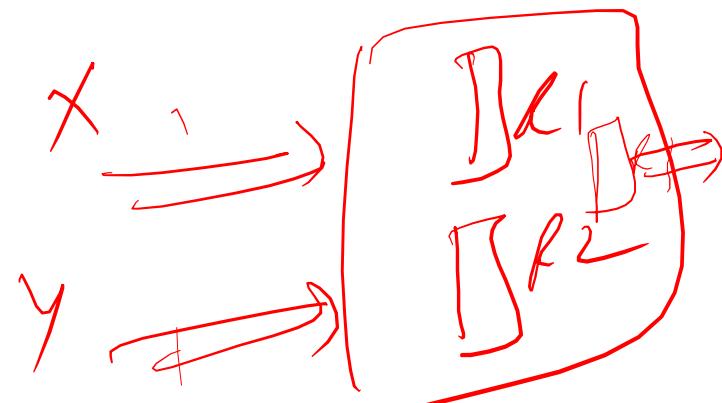
# GCD Algorithm

```
Input x, y;  
while ( x ≠ y ) do  
    if ( x > y )  
        then x := x - y  
    else y := y - x  
    endif;  
endwhile;  
z := x;  
end.
```

# Modified GCD Algorithm (RTL)

~~for  $\left\{ \begin{array}{l} \text{for } \\ \text{do } \end{array} \right\}$  each step~~

```
S0: R1 := x, R2 := y;  
S1: while ( R1 ≠ R2 ) do  
    S2: if ( R1 > R2 )  
    S3: then R1 := R1 - R2  
    S4: else R2 := R2 - R1  
    endif;  
endwhile;  
S5: R3 := R1;
```



# GCD Computer: Data Part

Diagram illustrating the data part of a GCD computer. It shows five components: R1, R2, Comp, R3, and SUB. Red handwritten annotations indicate "Data Path" with arrows pointing from R1 to R2 and from Comp to R3.

R1

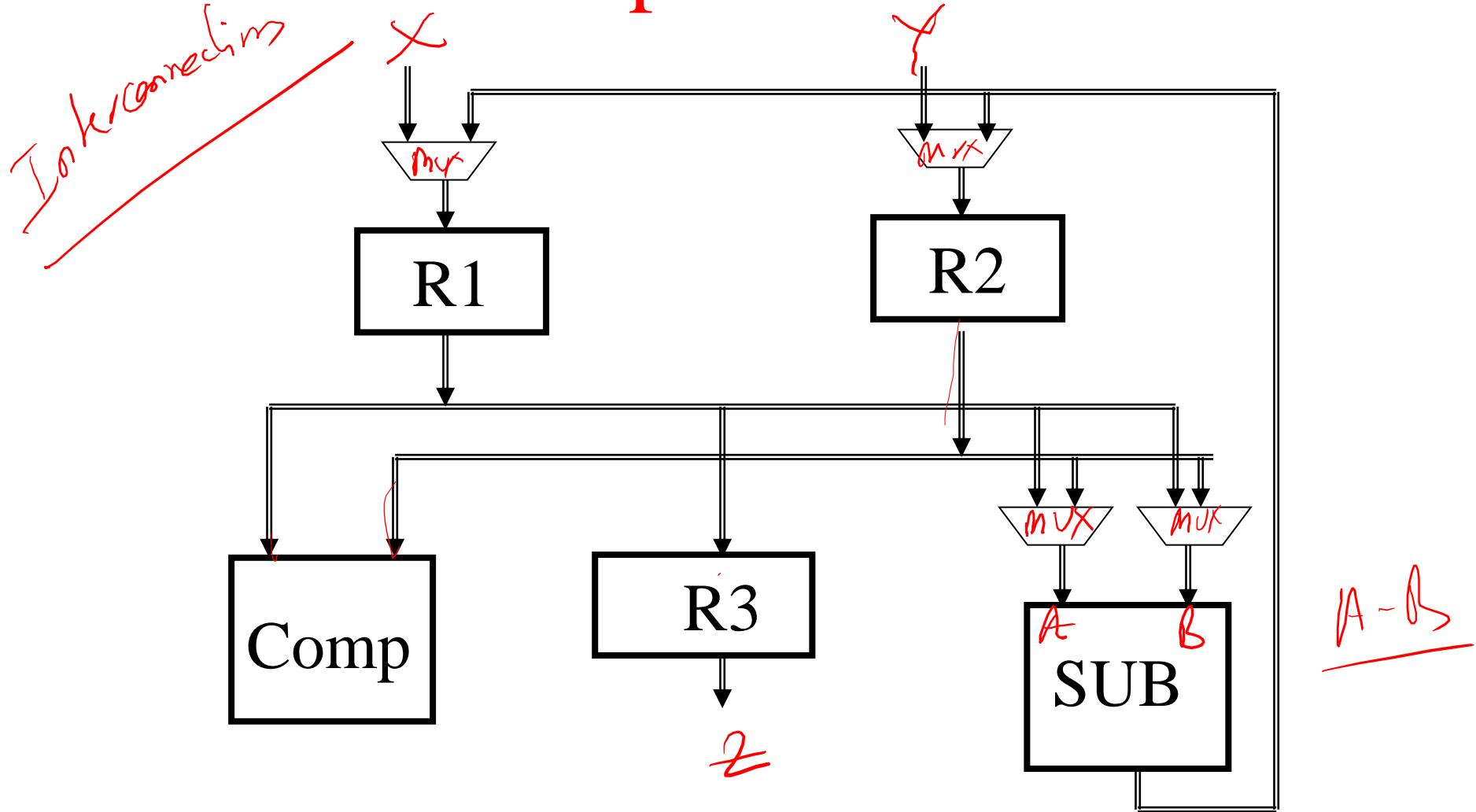
R2

Comp

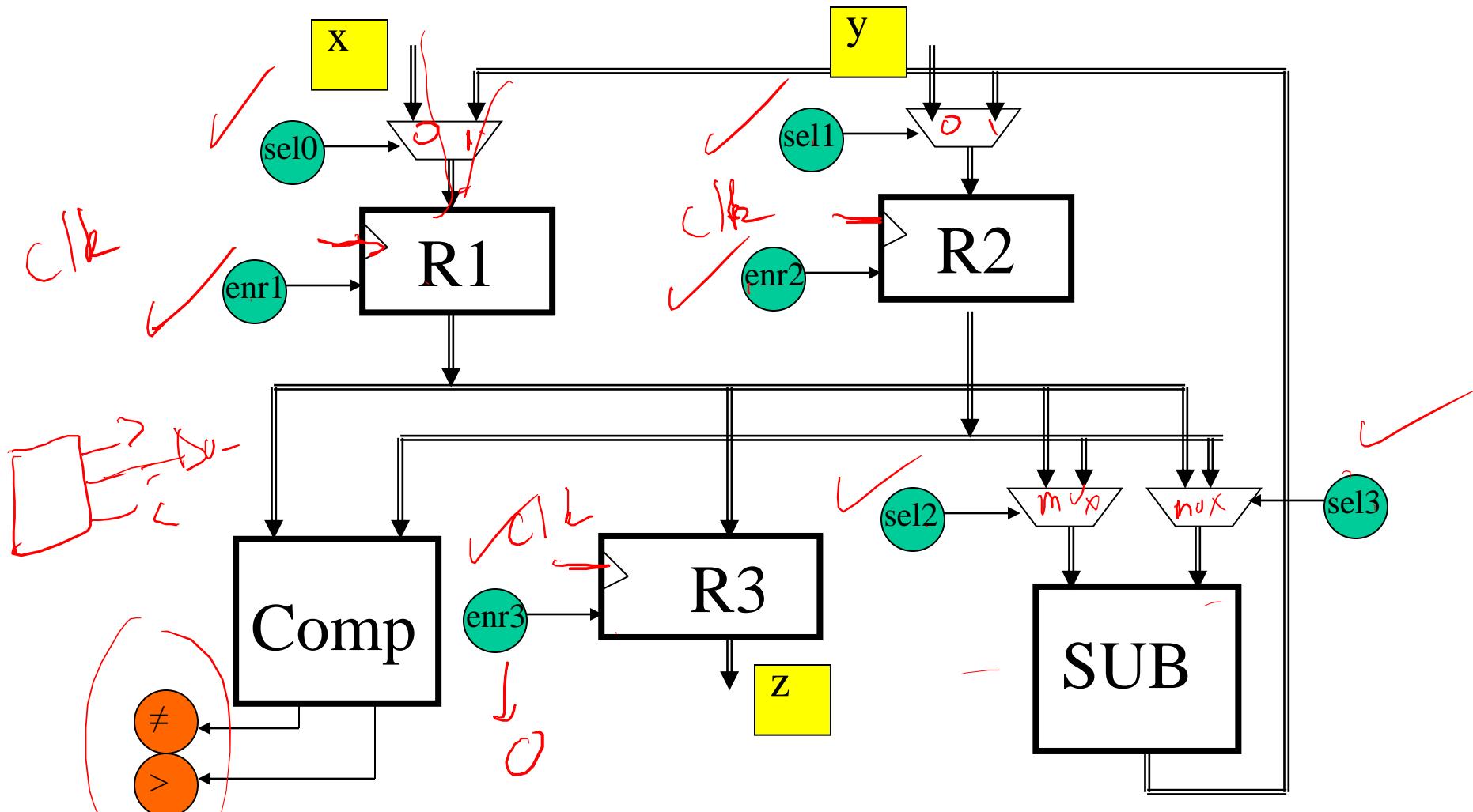
R3

SUB

# GCD Computer: Data Part

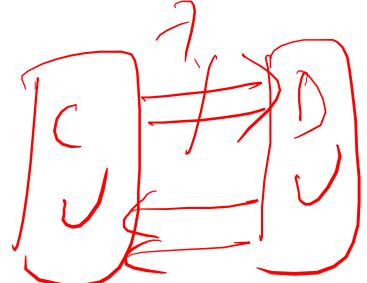


# GCD Computer: Data Part

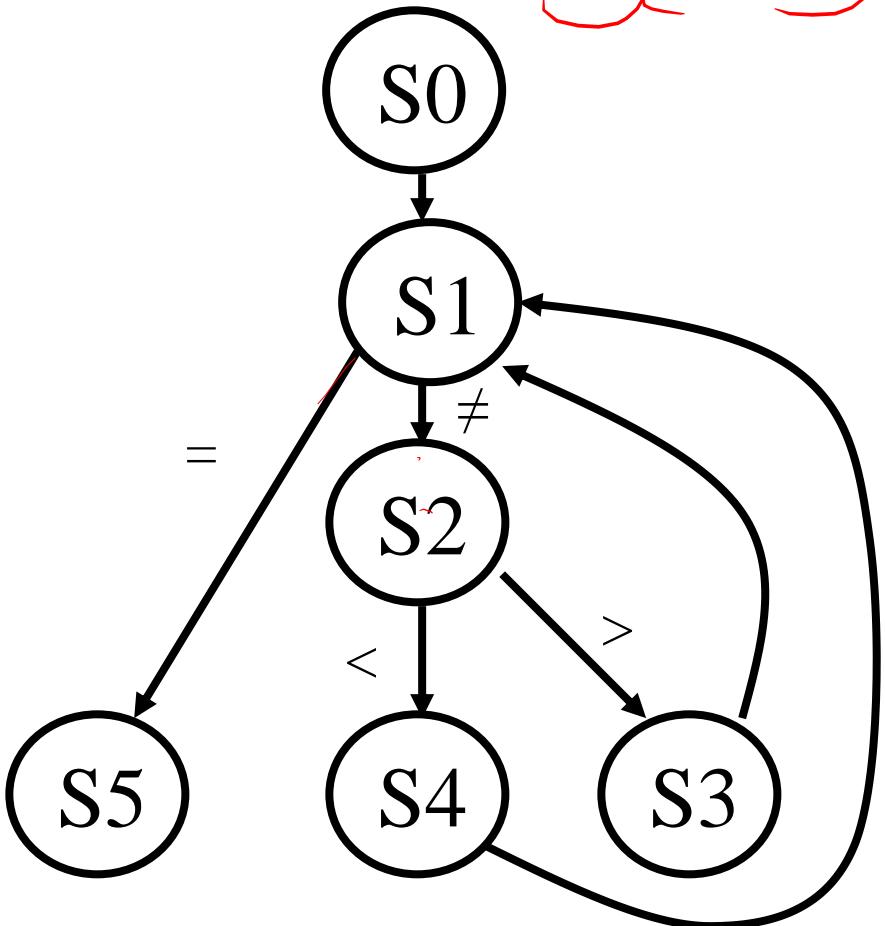


Section 3: Sequential Circuits

# Control Part FSM

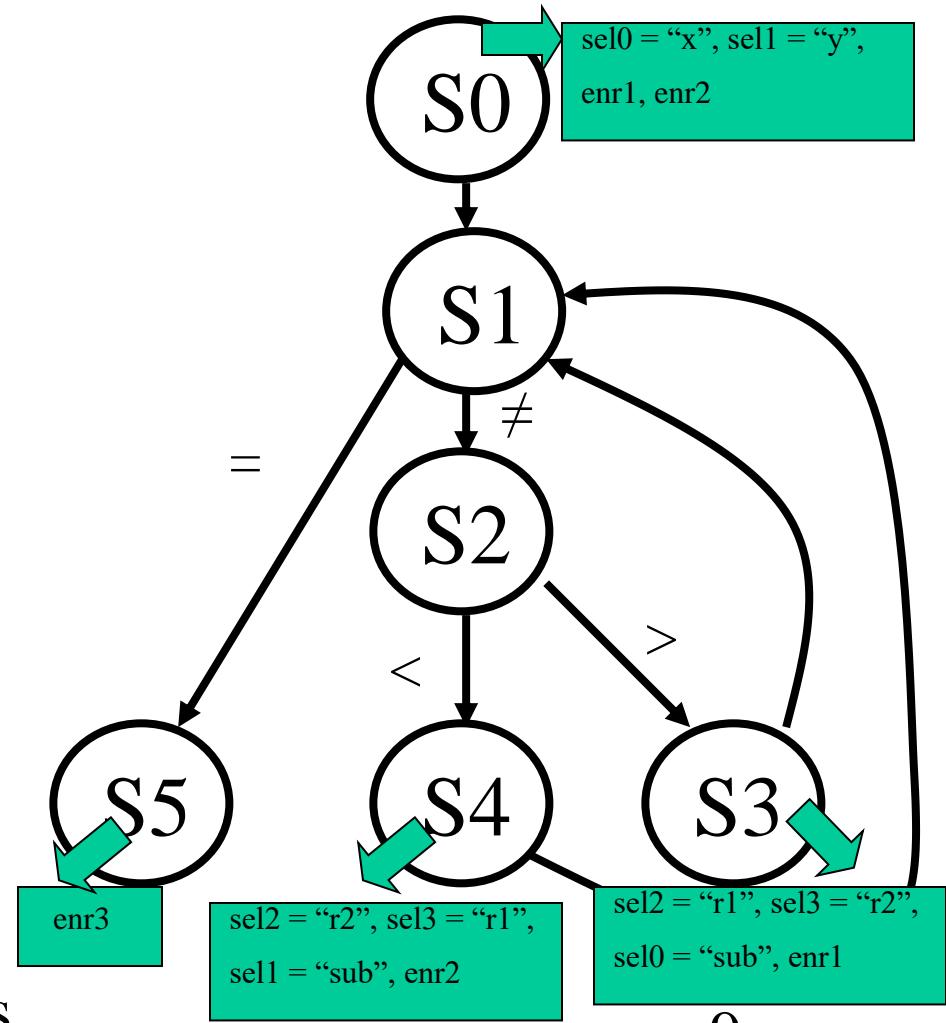


S0:R1:= x, R2:= y;  
S1:while ( R1 ≠ R2 ) do  
    S2: if ( R1 > R2 )  
    S3: then R1:= R1 - R2  
    S4: else R2:= R2 - R1  
    endif;  
endwhile;  
S5: R3:= R1;

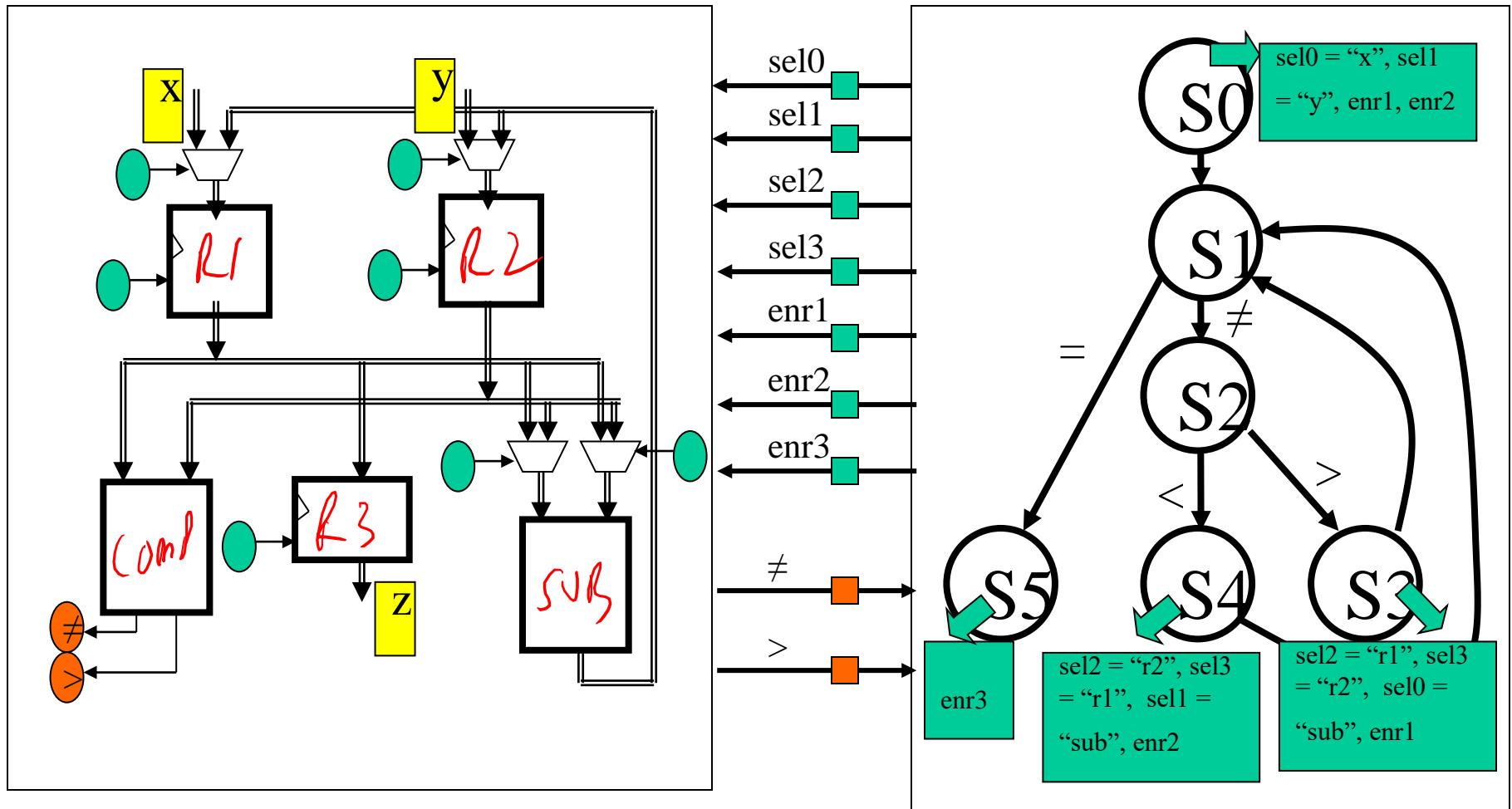


# FSM: Control Signals

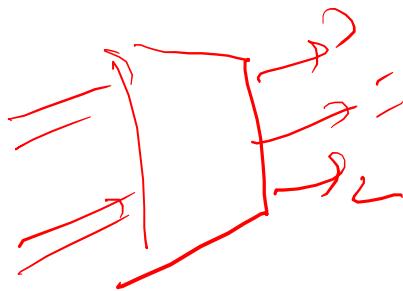
```
S0:R1:= x, R2:= y;  
S1:while ( R1 ≠ R2 ) do  
    S2: if ( R1 > R2 )  
    S3: then R1:= R1 - R2  
    S4: else R2:= R2 - R1  
    endif;  
endwhile;  
S5: R3:= R1;
```



# Data-Control Interface



Section 3: Sequential Circuits



## Modified RTL + FSM

S0: R1 := x, R2 := y;

S1: Case ( R1 “Compare” R2 )

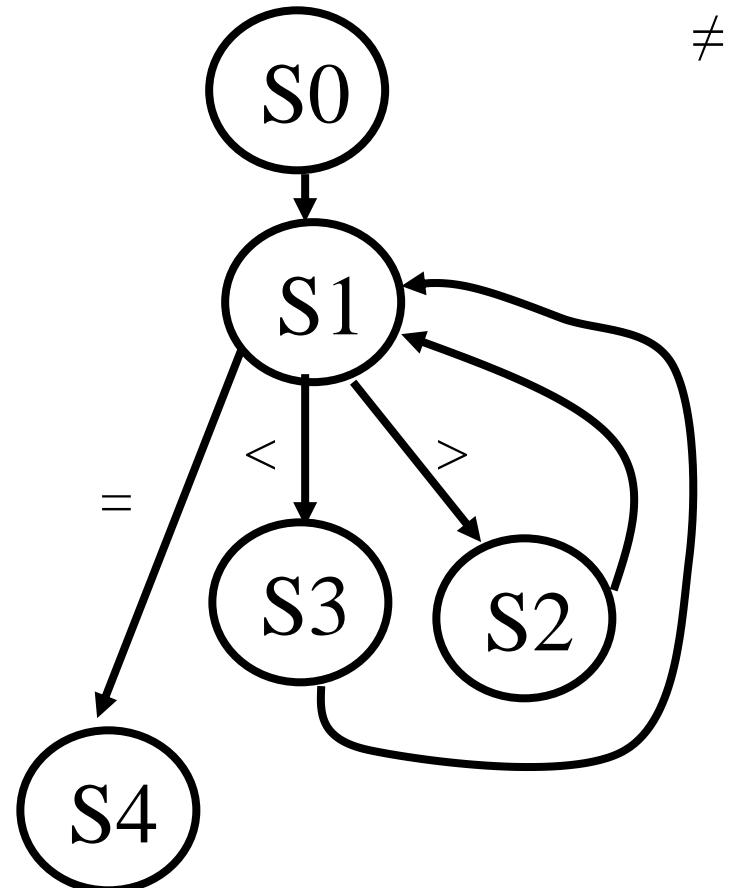
S2: “>” R1 := R1 - R2,  
go to S1;

S3: “<” R2 := R2 - R1,  
go to S1

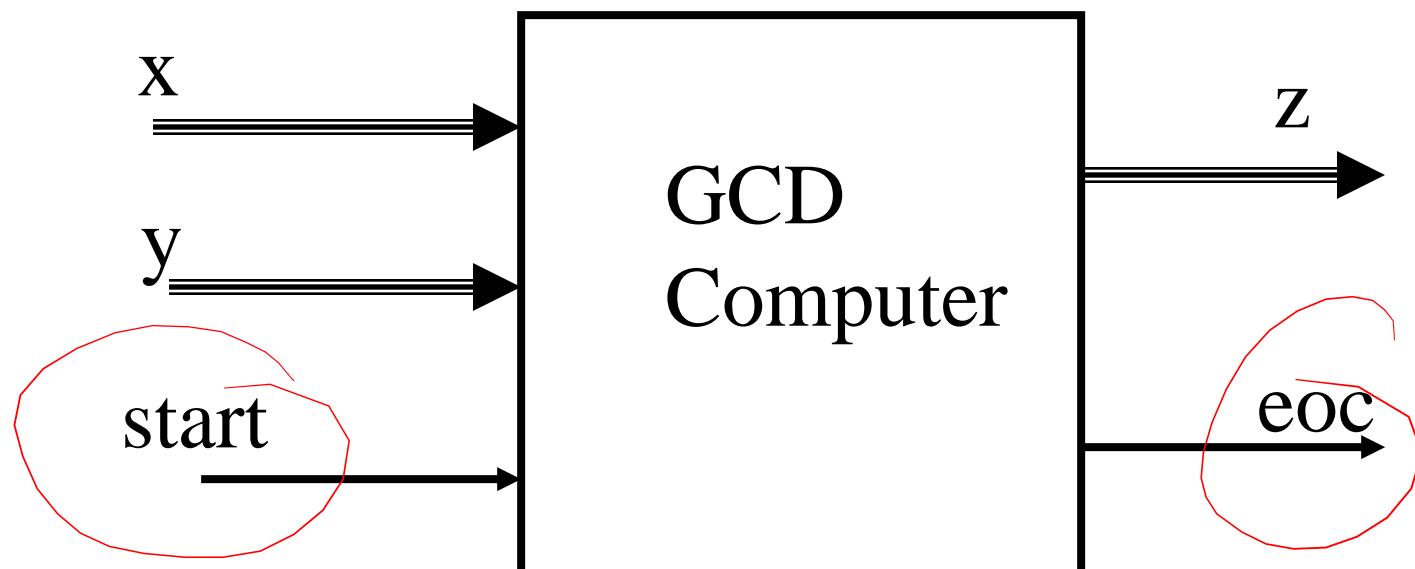
S4: “=” R3 := R1;

endcase;

Section 3: Sequential Circuits



# GCD Computer: Interface



# Modified FSM

Si: wait for “start”

S0: R1:= x, R2:= y, eoc := “0”;

S1: while ( R1  $\neq$  R2 ) do

S2: if ( R1 > R2 )

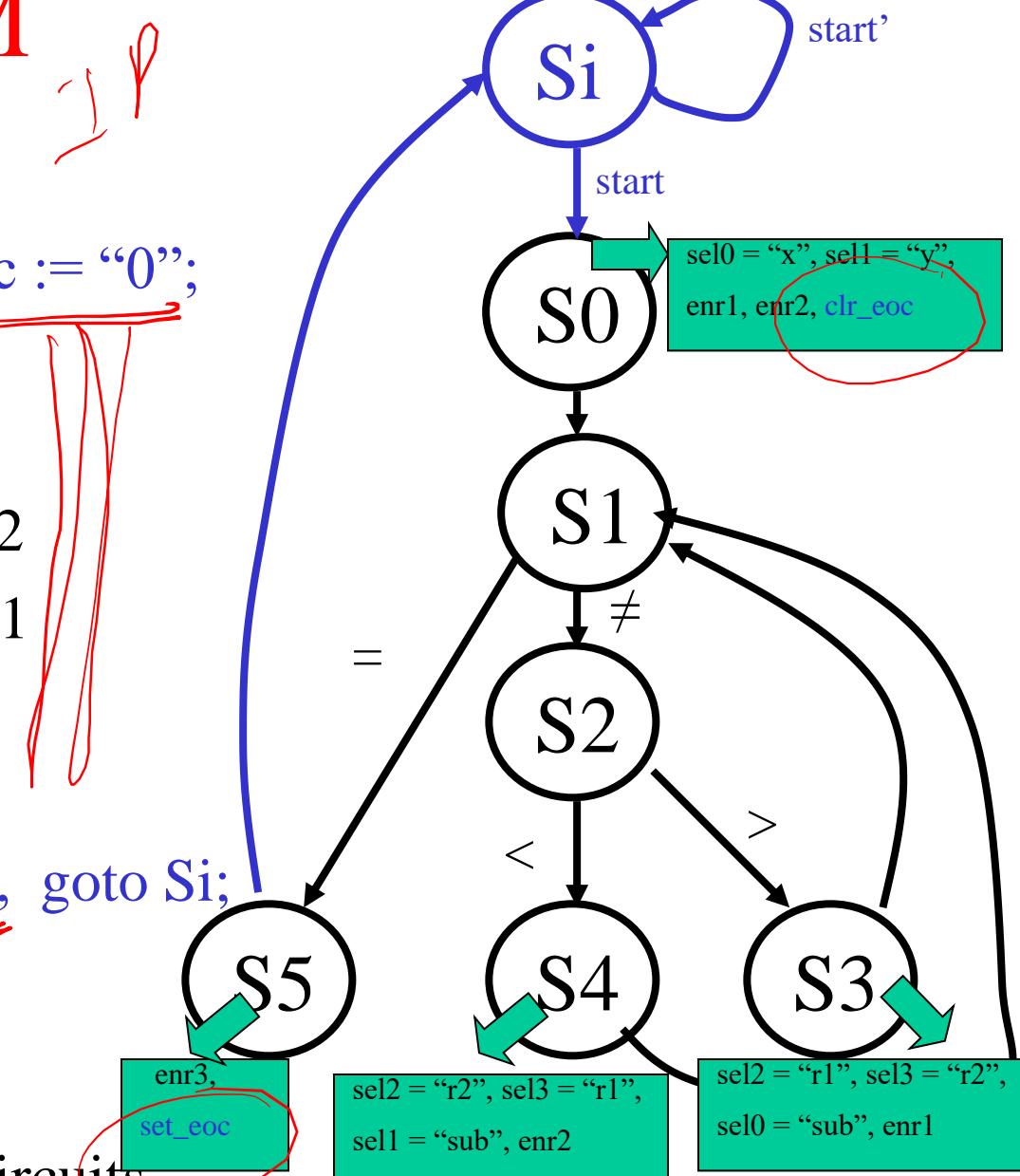
S3: then R1:= R1 - R2

S4: else R2:= R2 - R1

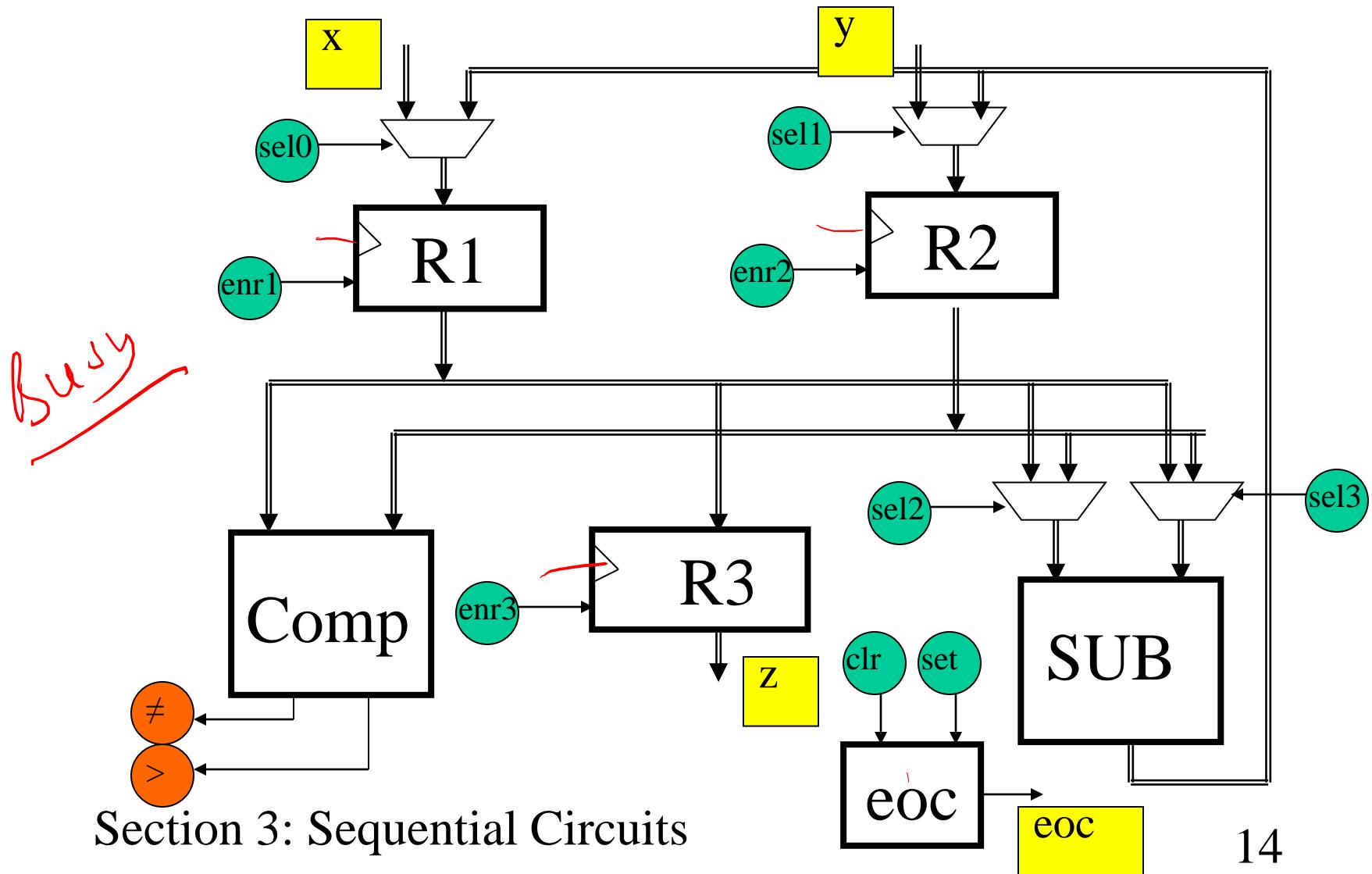
endif;

endwhile;

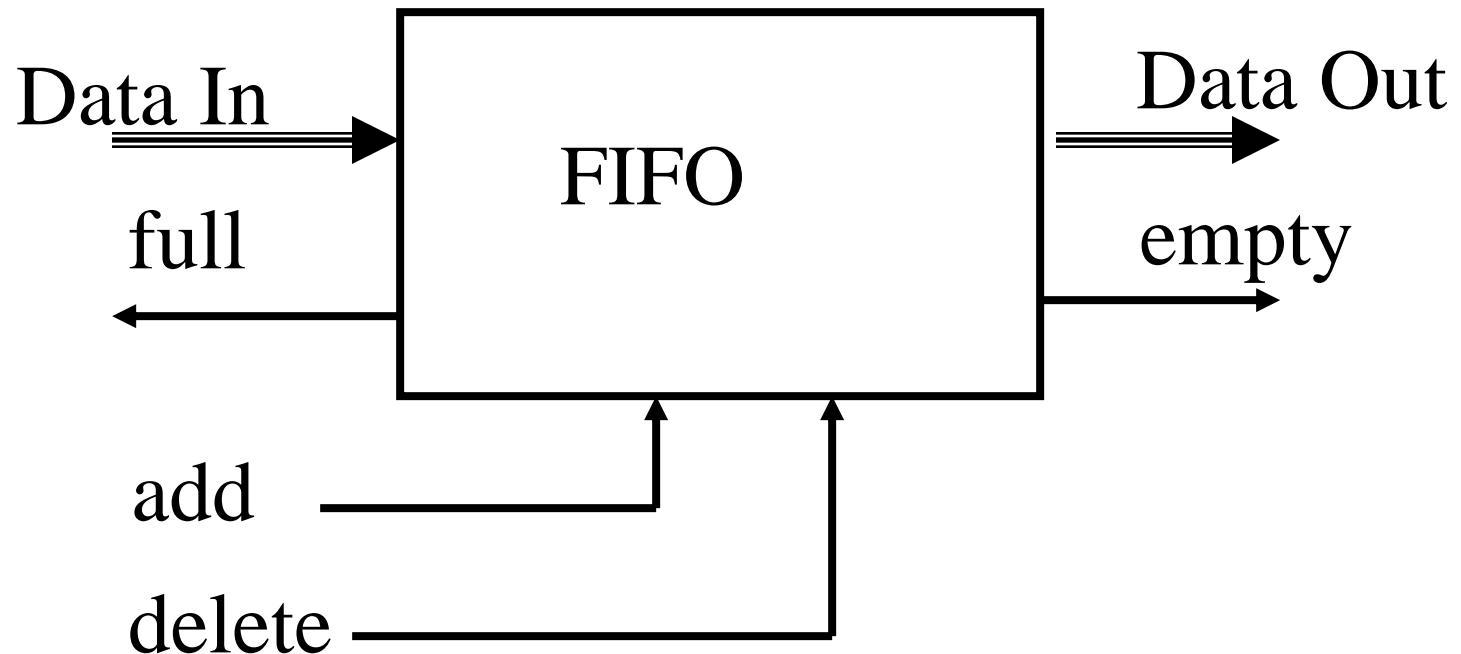
S5: R3:= R1, eoc := “1”, goto Si;



# GCD Computer: Data Part



# Case Study 2: FIFO



# FIFO: Data Part

Head

Memory

Tail

f

e

# FIFO: RTL

Si: head := “0”, tail := “0”, full := “0”, empty := “1”;

S0: If ( add and full’) then

S1: mem(head) := data\_in;

S2: head := head +1;

S3:: full := (head == tail); go to S0;

S4: else if (delete and empty’) then

S5: tail := tail + 1;

S6: data\_out:= mem(tail);

S7: empty := (head == tail); go to S0;

# Design Problem

Design a system which uses GCD module and multiple FIFO modules. An array of numbers are stored in a memory and GCD of these numbers is to be computed and stored back into the memory. FIFO buffers are used to supply the inputs to the GCD module and take the outputs from the GCD module.

# Lecture 26

# Designing with Memories

M. Balakrishnan

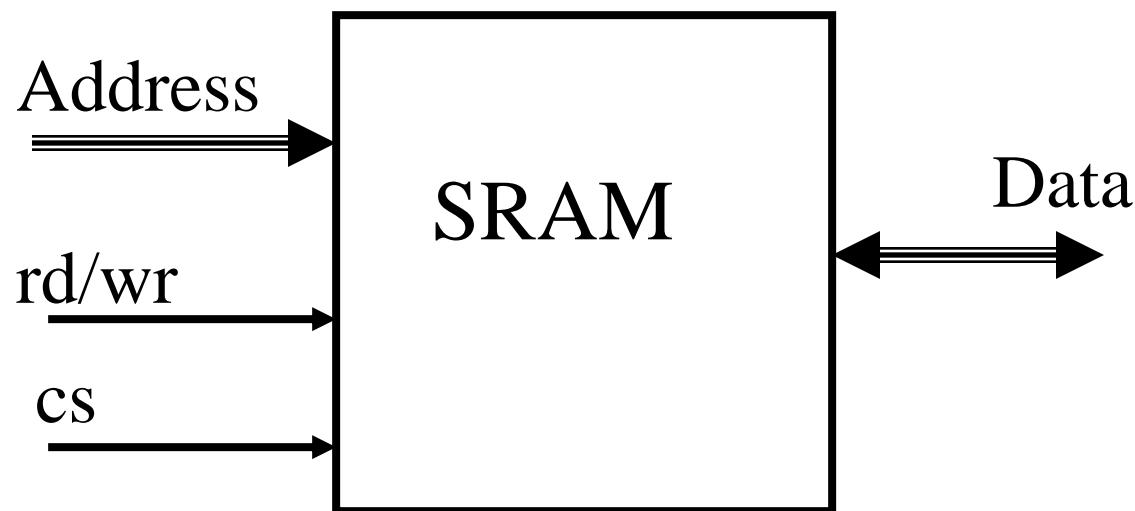
Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

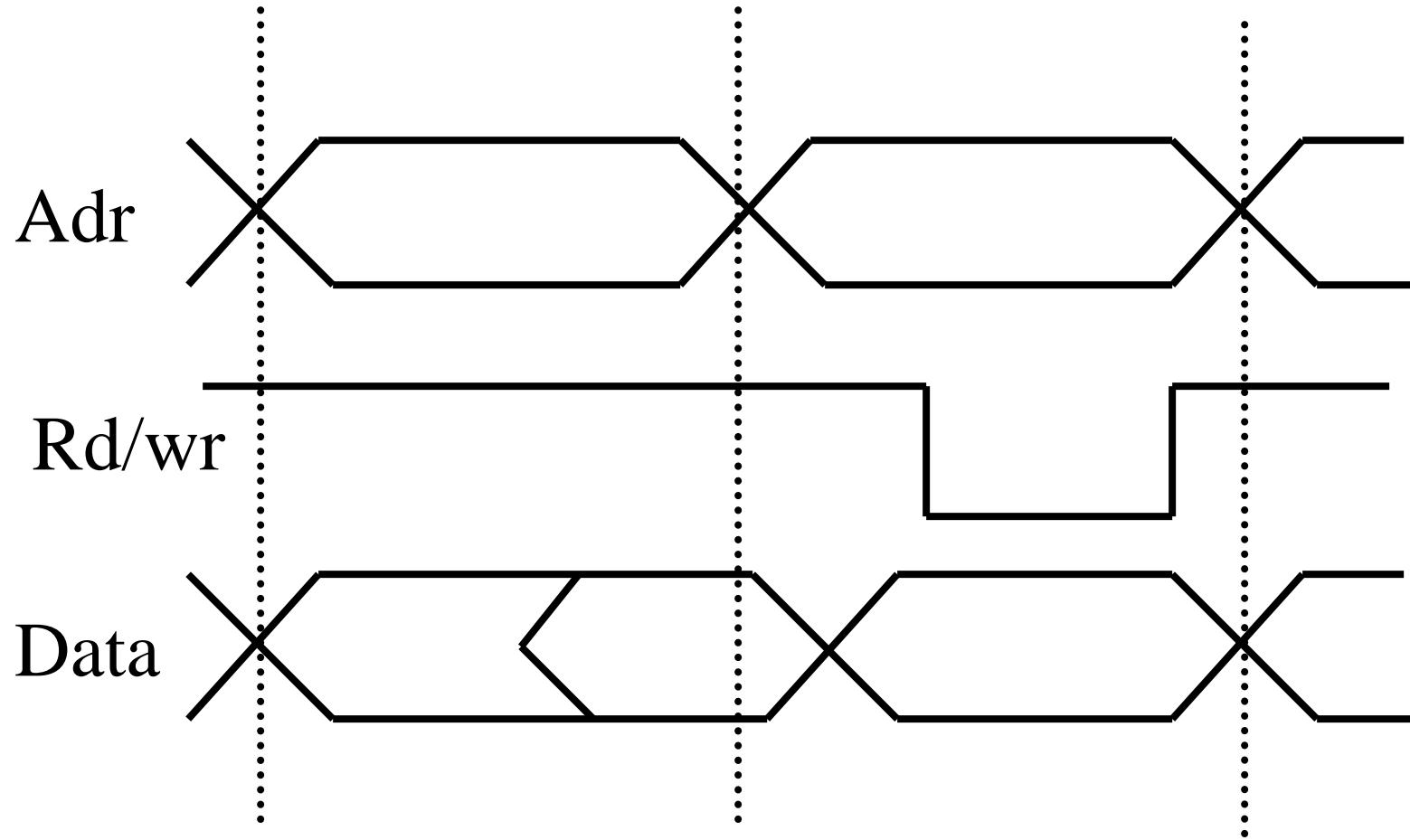
# Classification of Memory Devices

- ROM
  - ROM, PROM, EPROM, EEPROM, UV PROM
- RAM
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)
- Flash memory (Flash ROM)

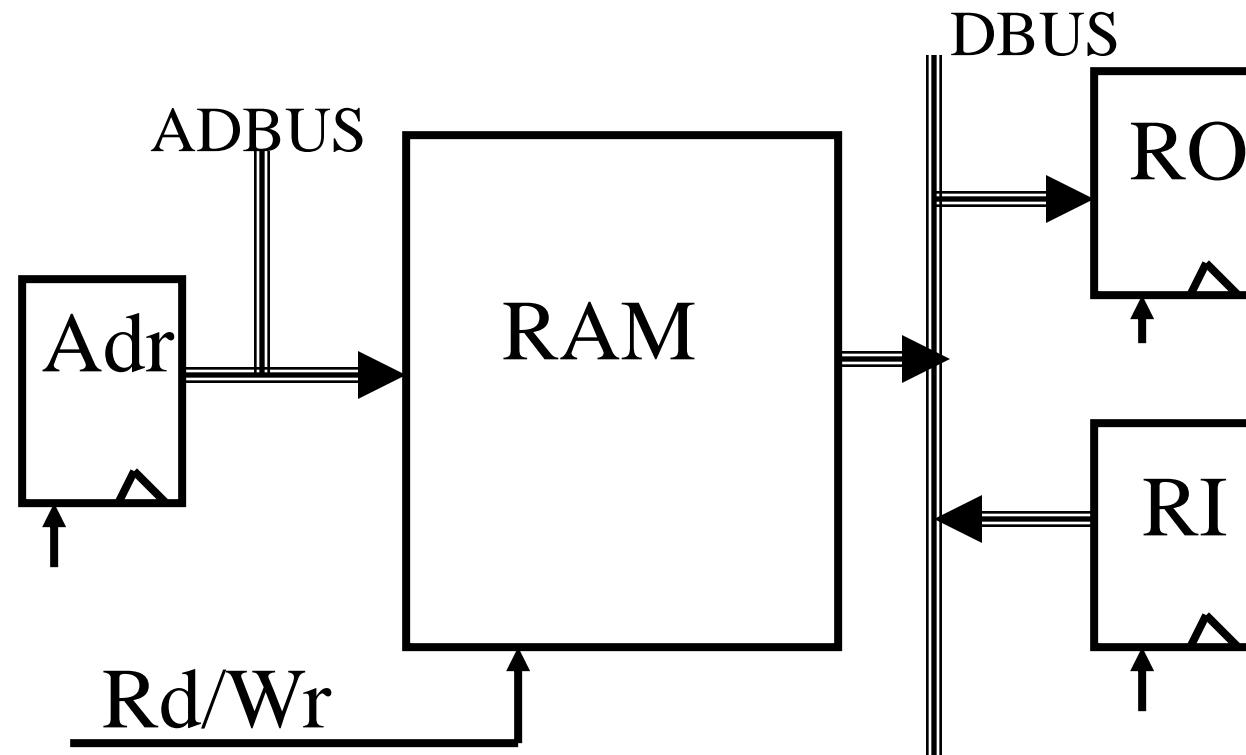
# SRAM Device Signals



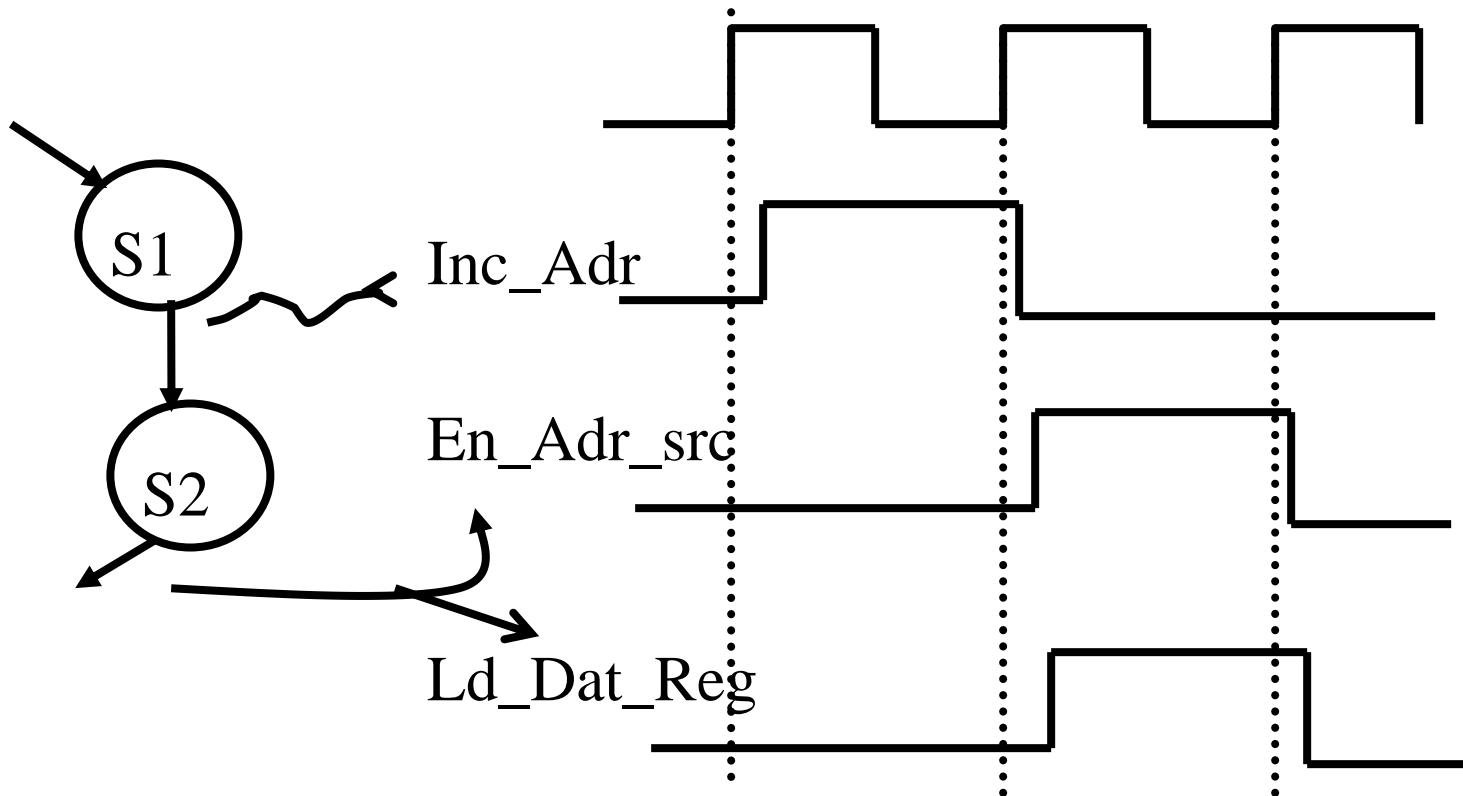
# SRAM Timing



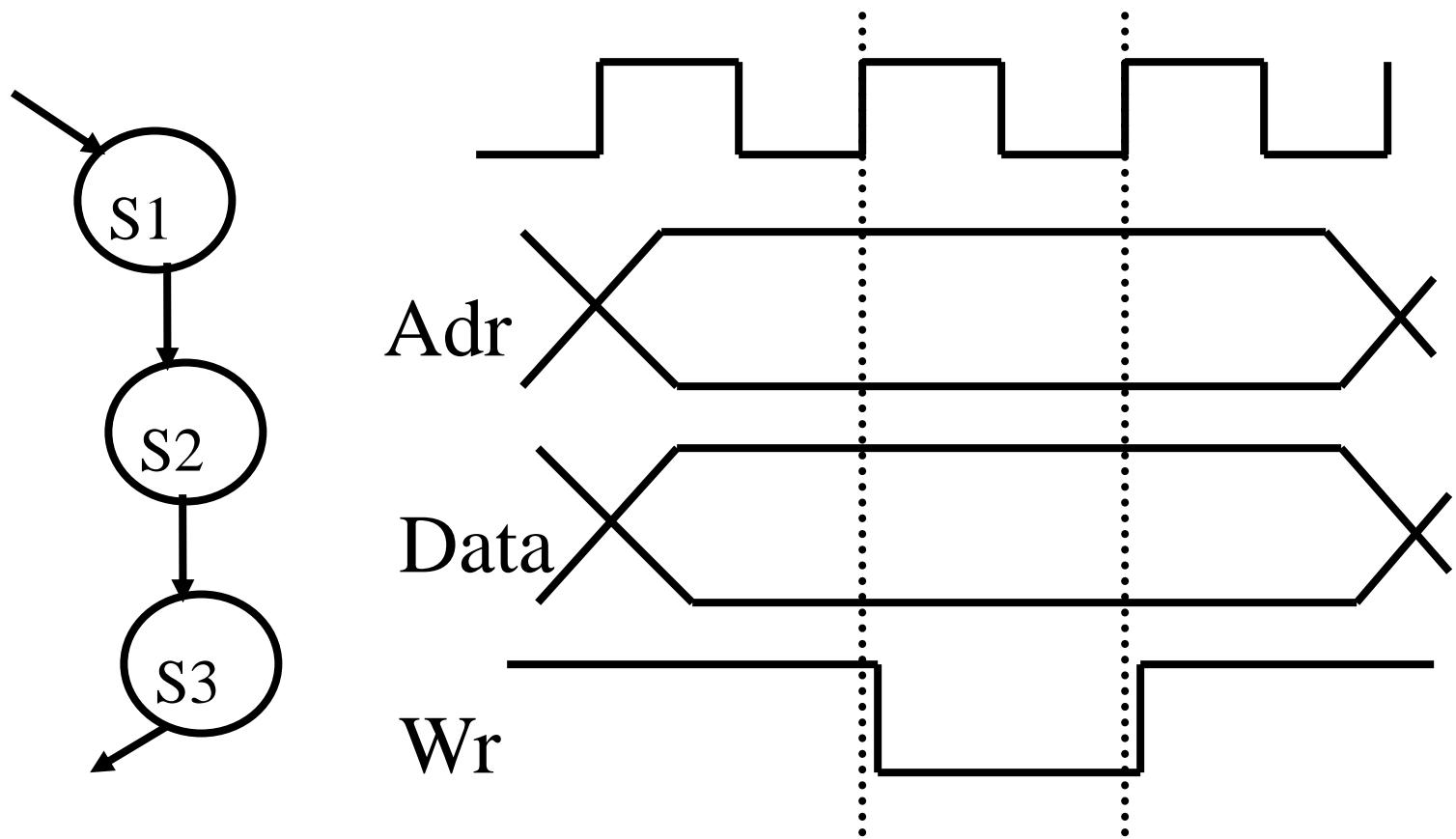
# Circuit Example using Memory

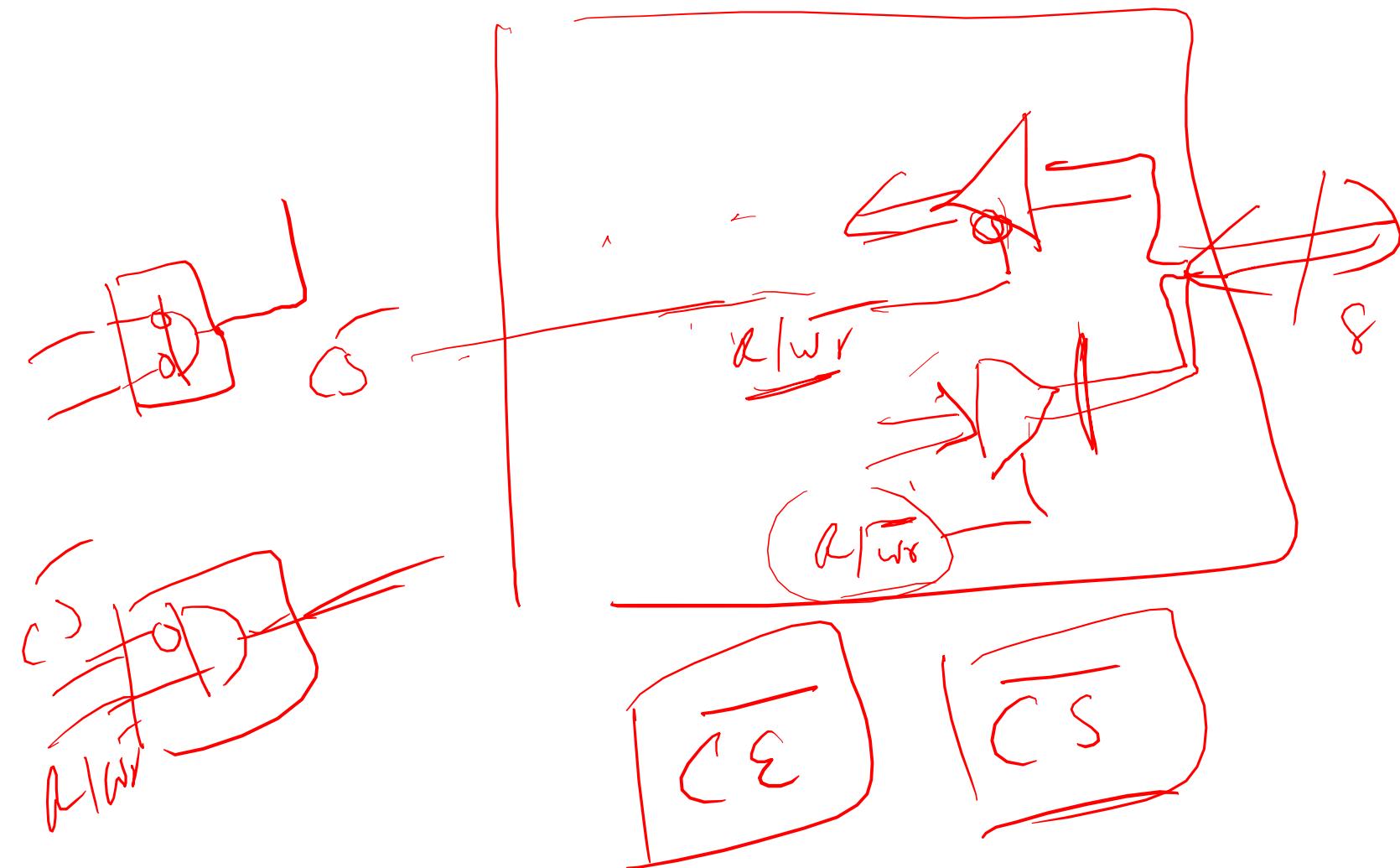


# Reading Memory in a SM

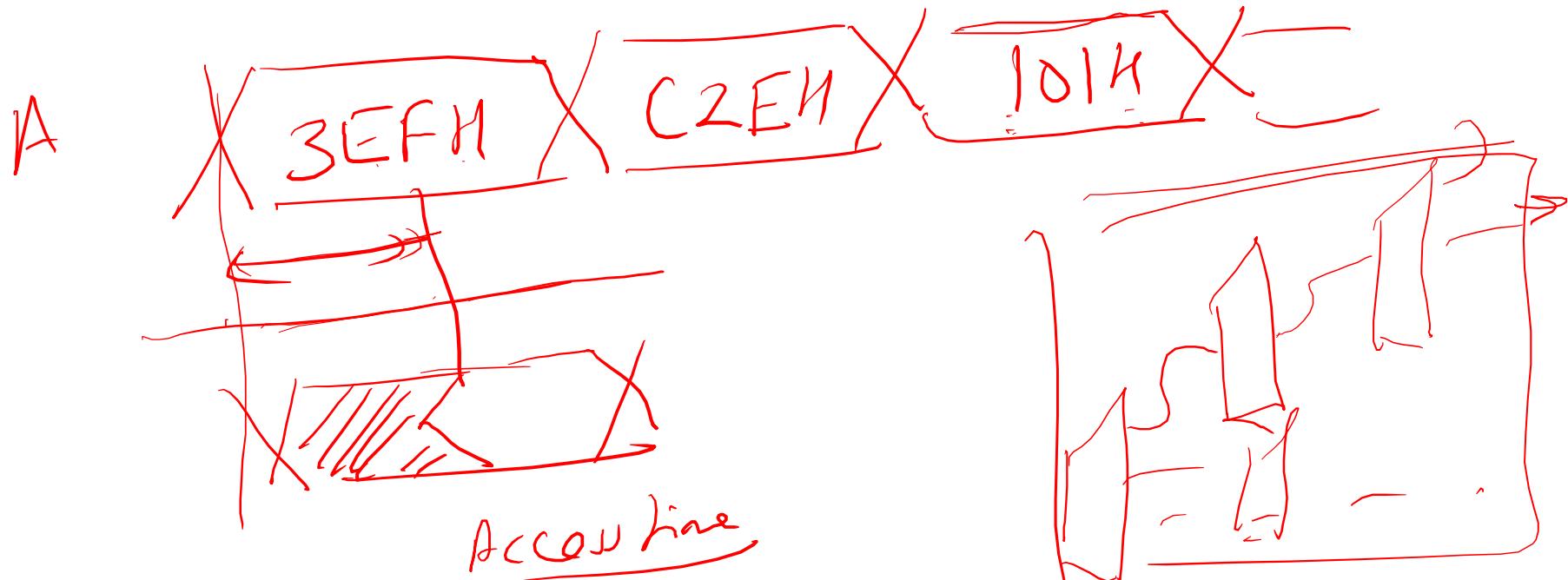
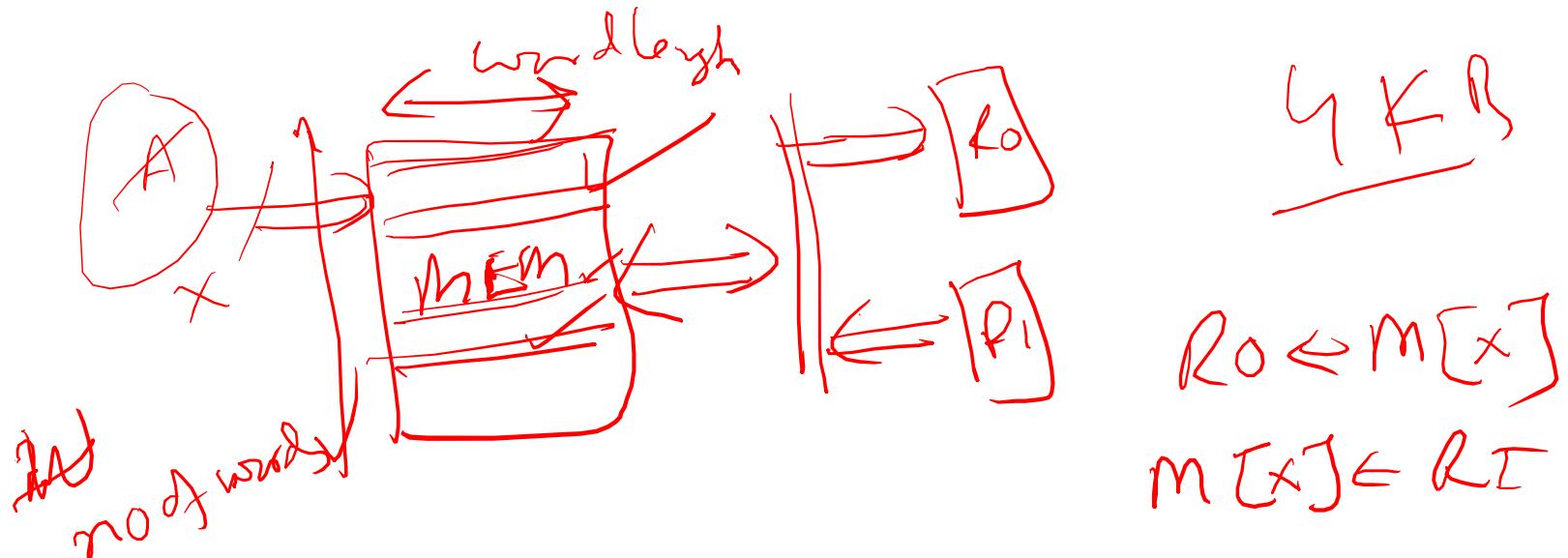


# Writing Memory in a SM

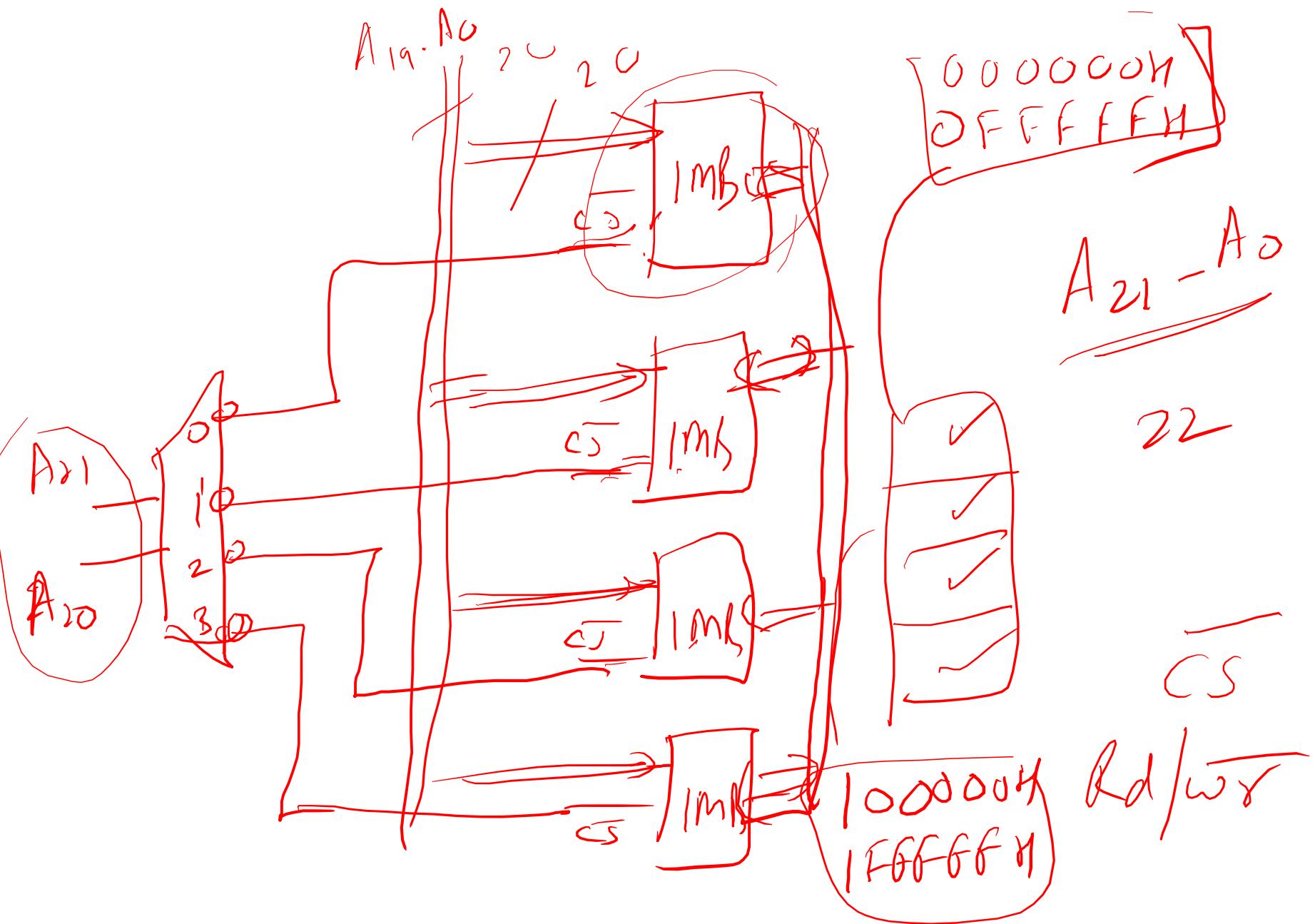




## Section 3: Sequential Circuits

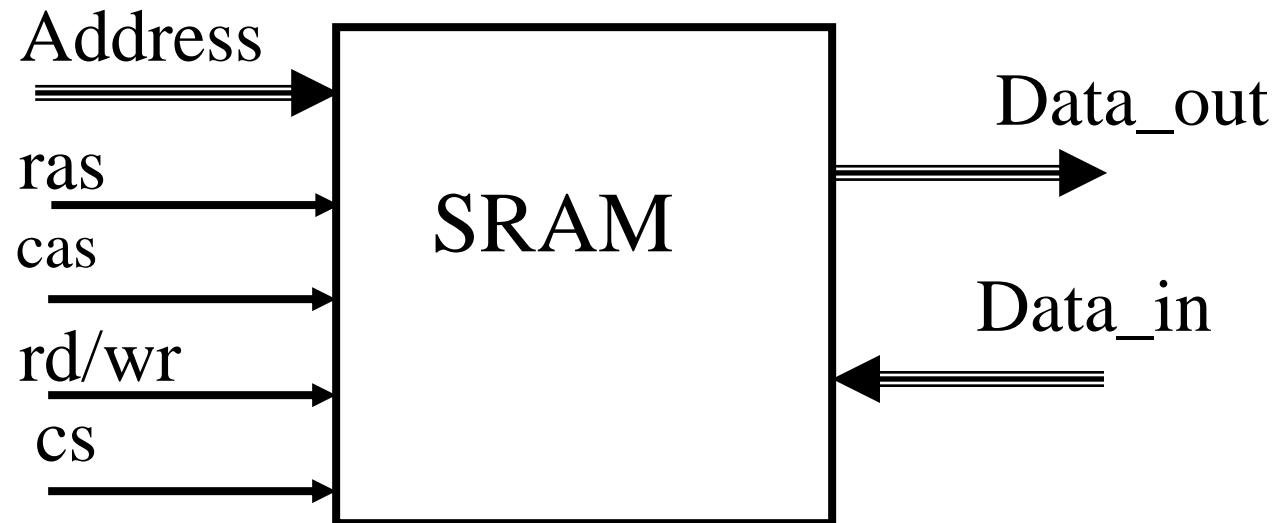


### Section 3: Sequential Circuits

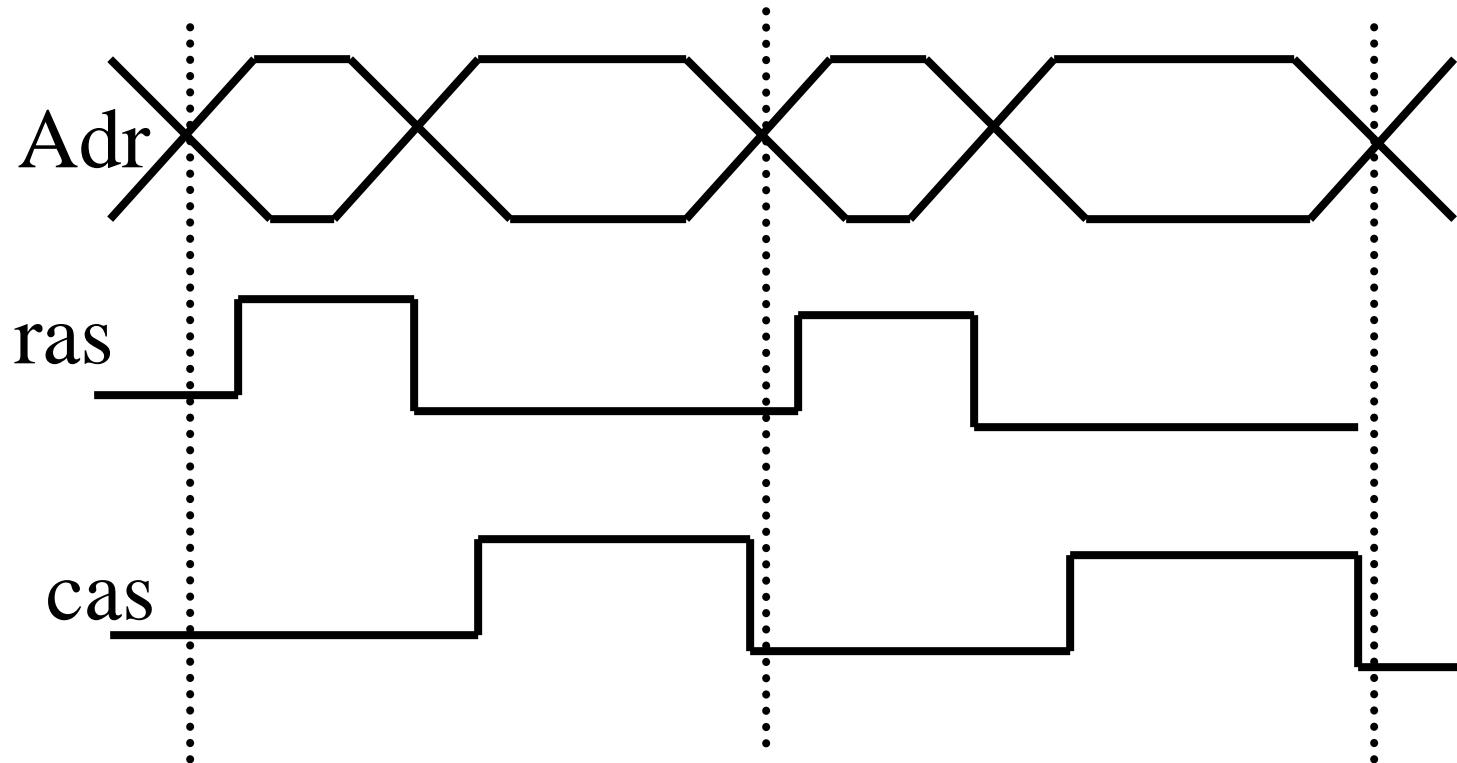


### Section 3: Sequential Circuits

# Dynamic RAM Device Signals



# DRAM Timing



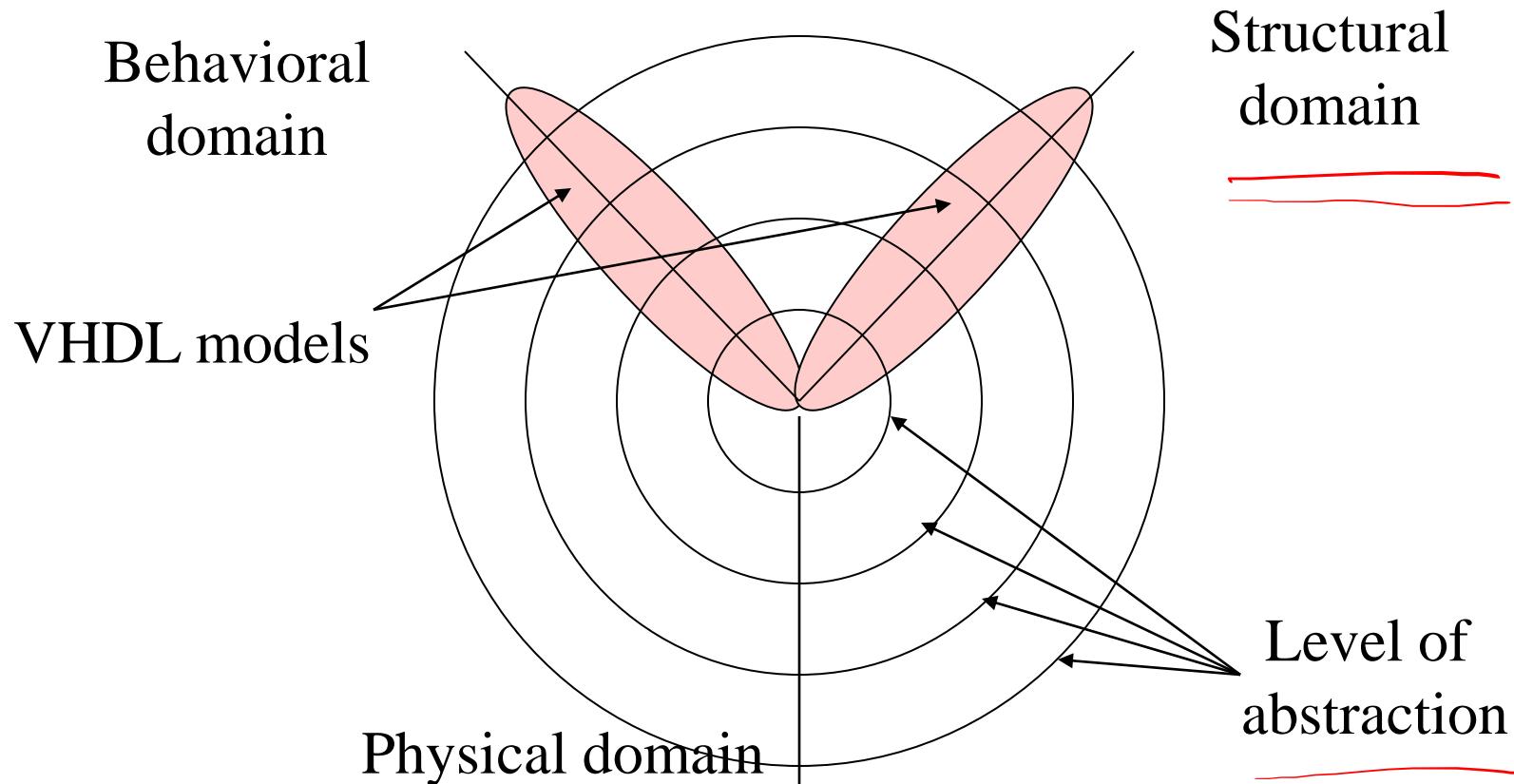
# Lecture 27: Introduction to VHDL

M. Balakrishnan

Dept of Computer Science & Engg.

I.I.T. Delhi

# Domains of Description : Gajski's Y-Chart



# VHDL Development

- US DoD initiated in 80's
- Very High Speed ASIC Description Language
- Initial objective was modeling only and thus only a simulator was envisaged
- Subsequently tools for VHDL synthesis were developed

# HDL Requirements

- Abstraction
- Modularity
- Hierarchy
- Concurrency

# Concurrency in Hardware

# Abstraction

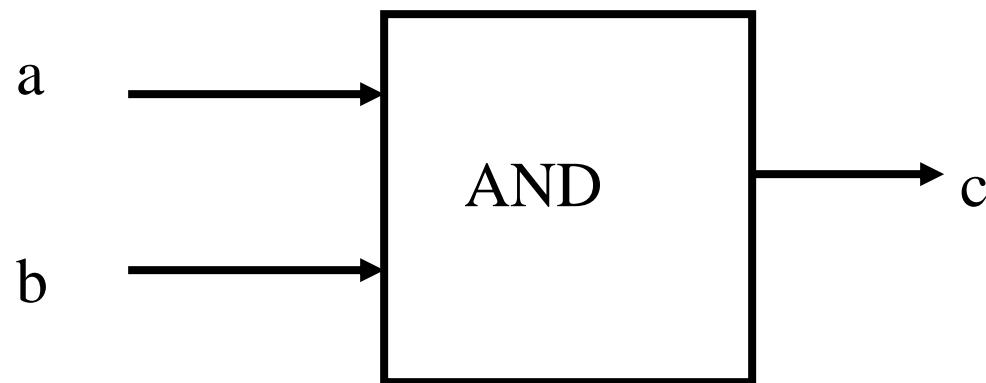
VHDL supports description of components as well as systems at various levels of abstraction

- Gate and component delays
- Clock cycles
- Abstract behavior without any notion of delays

# Modularity

- Every component in VHDL is referred to as an entity and has a clear interface
- The interface is called an entity declaration
- The “internals” of the component are referred to as the architecture declaration
- There can be multiple architectures at even different levels of abstraction associated with the same entity ??

# VHDL Example

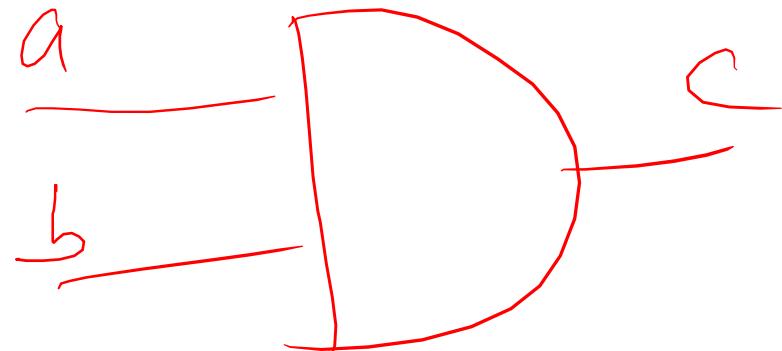


# VHDL Description: AND gate

```
entity AND2 is
  port(a, b: in bit;
       c : out bit);
end AND2;
```

```
architecture beh of AND2 is
begin
```

```
  c <= a and b; after 1ns;
end beh;
```

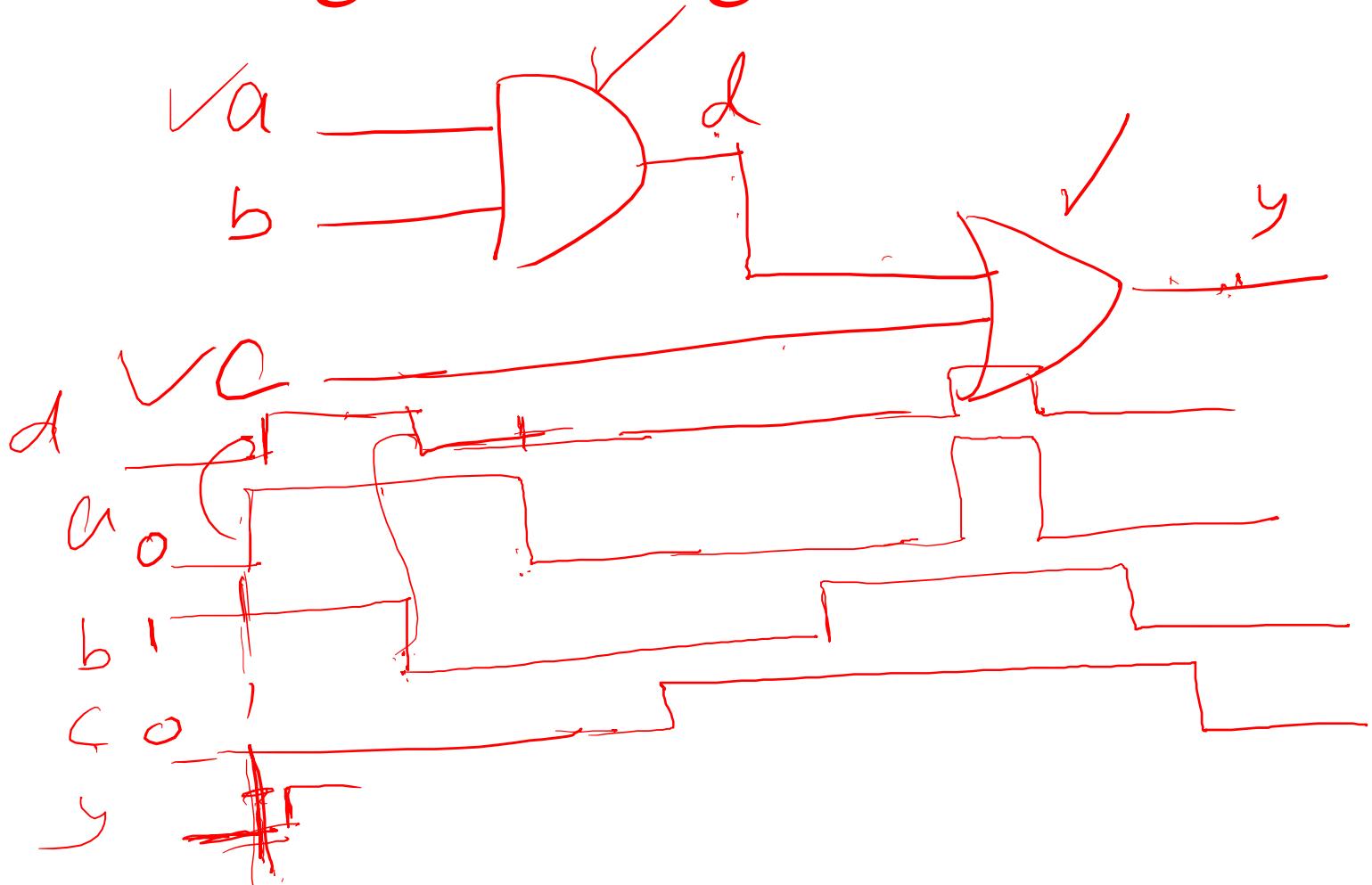


Concurrent assignment

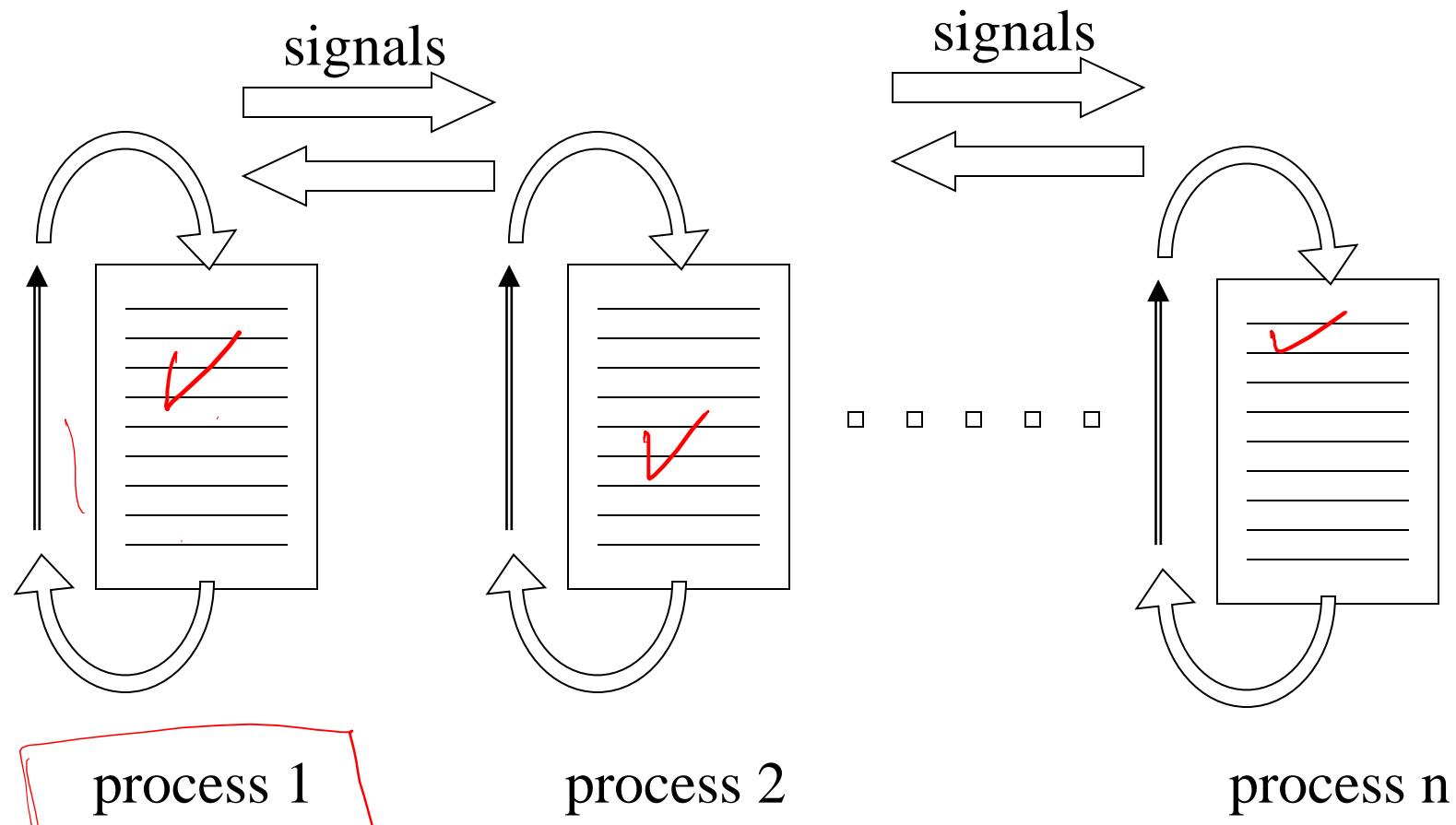
Events  
- tour Travellim

TS delay

# Signal Assignment



# Concurrency in VHDL Descriptions



Section 5: Introduction to VHDL

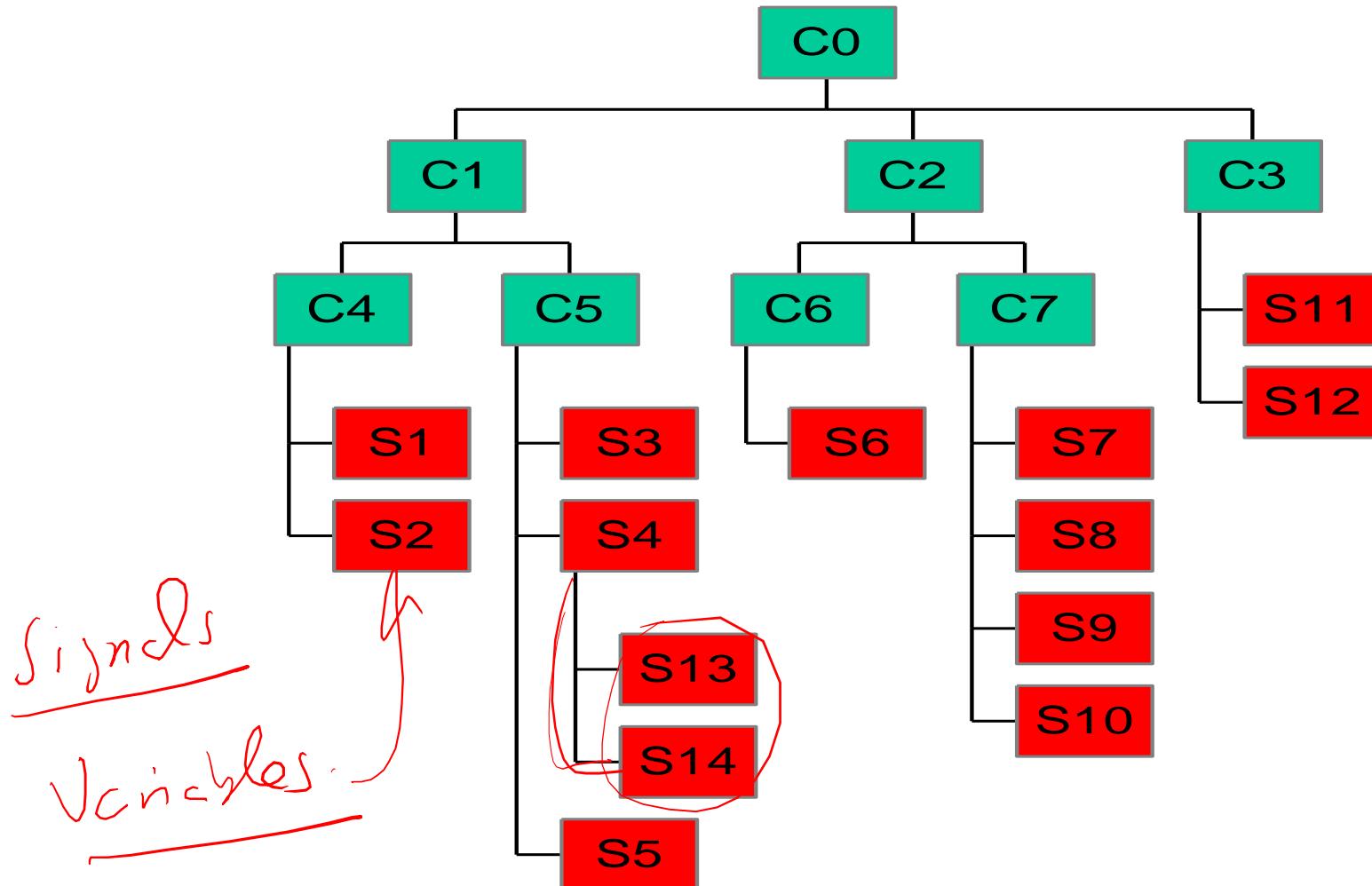
# Concurrent and Sequential Computations

- Processes are concurrent
- Sequential activity within each process

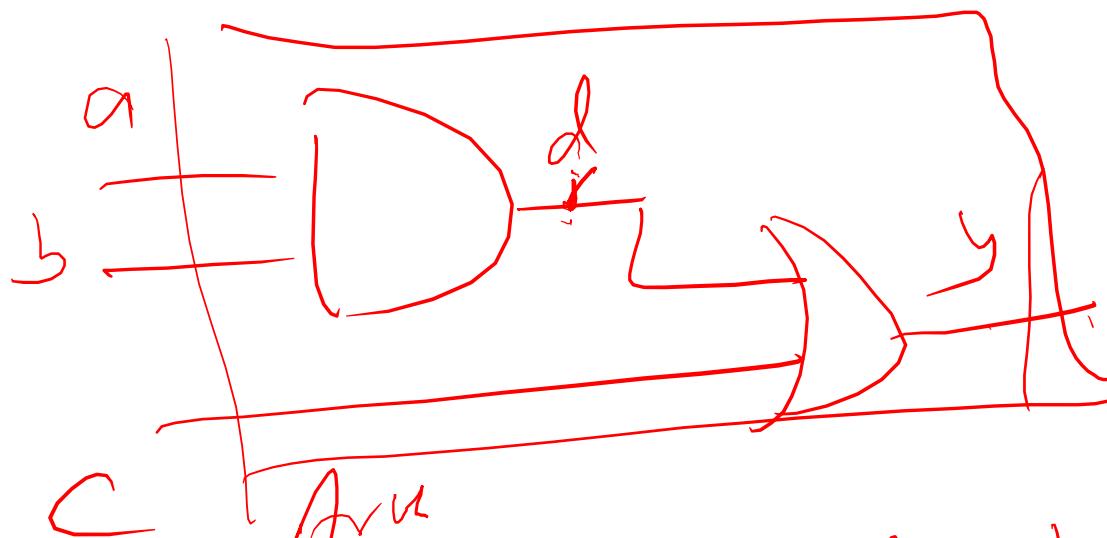
## Nesting of statements :

- Concurrent statements in a concurrent statement
- Sequential statements in a concurrent statement
- Sequential statements in a sequential statement

# Hierarchy in VHDL



# Worksheet



$t+8$

$t+8, E+2\delta$

$\checkmark d \leftarrow a \text{ AND } b;$

$\checkmark y \leftarrow c \text{ OR } d; \quad \checkmark$

$\checkmark d \leftarrow a \text{ AND } b; \quad \checkmark$

# Lecture 28: Modeling Styles in VHDL

M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

# Modeling Styles

- Semantic model of VHDL
- Structural description
- Data Flow description
- Algorithmic description
- RTL description

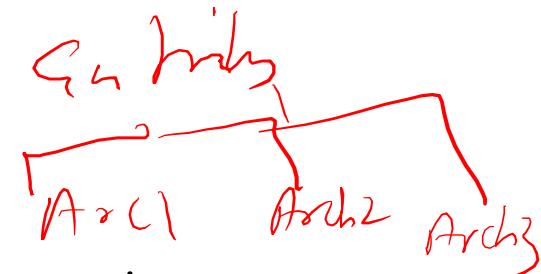
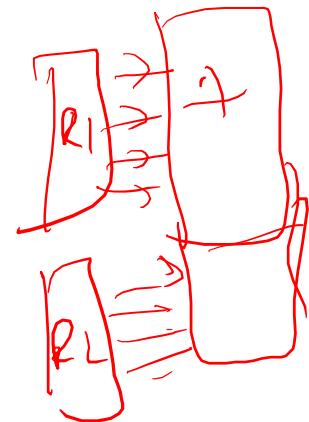
# Modeling Choices in VHDL

- Behavioral and Structural Domains
  - Several Levels of Abstraction
- Multiple Styles of Behavioral Description:
  - Data Flow Style (concurrent) ✓
  - Procedural Style (sequential) ✓
- Combinations, variations and special cases of these, e.g.,
  - special case of data flow style - FSM described using guarded blocks
  - special case of procedural style - FSM described using case statement in a process

Introduction

# Structural Description

- Carries same information as a NET LIST
- Net List = (Component instances) + (Nets)
- Structural Description in VHDL =  
(Signals) + (Component instances + Port maps)
- Many sophisticated features in VHDL to make it more versatile:
  - \* Variety of signal types
  - \* Generic components
  - \* Generate statements for creating arrays of component instances
  - \* Flexibility in binding components to design entities and architectures



# Behavioral Description

- **Procedural**  
(textual order  $\Rightarrow$  execution order)
- **Sequential statements**
- **Control constructs alter normal sequential flow**

Called Behavioral  
description in VHDL

- **Non-procedural**  
(textual order NOT  $\Rightarrow$  execution order)
- **Concurrent statements**
- **Data flow (or rather data dependency restricts concurrency)**

Called Data flow  
description in VHDL

# Concurrent Statements in VHDL

- process statement                         -- behavior
- concurrent procedure call              -- behavior
- concurrent signal assign.              -- data flow
- component instantiation                -- structure
- generate statement                      -- structure
- block statement                         -- nesting
- concurrent assertion stmt            -- error check

# Example: 1-bit Full Adder

```
entity FullAdder is  
port (X, Y, Cin: in bit; -- Inputs  
      Cout, Sum: out bit); -- Outputs  
end FullAdder;
```

FA

Architecture

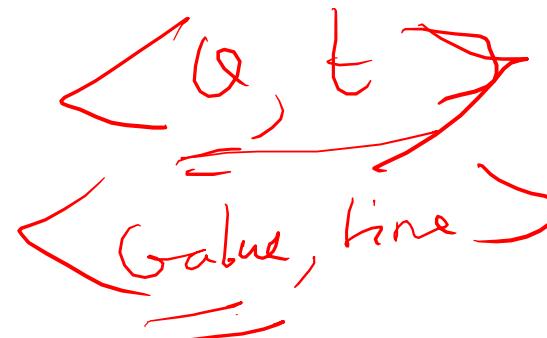


# Example: 1-bit Full Adder (contd.)

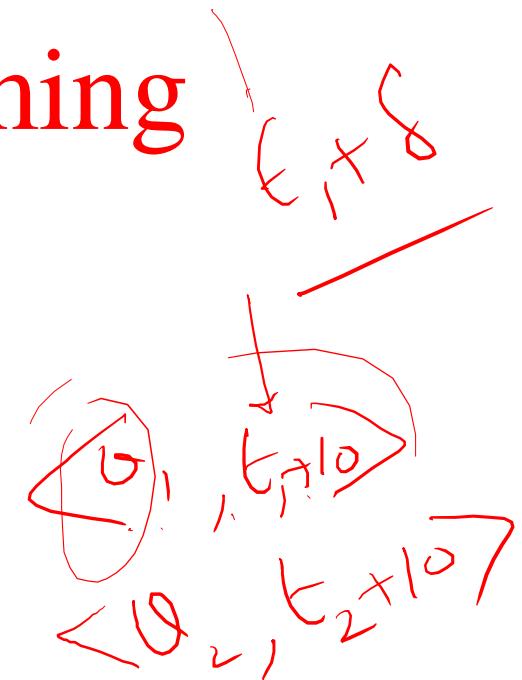
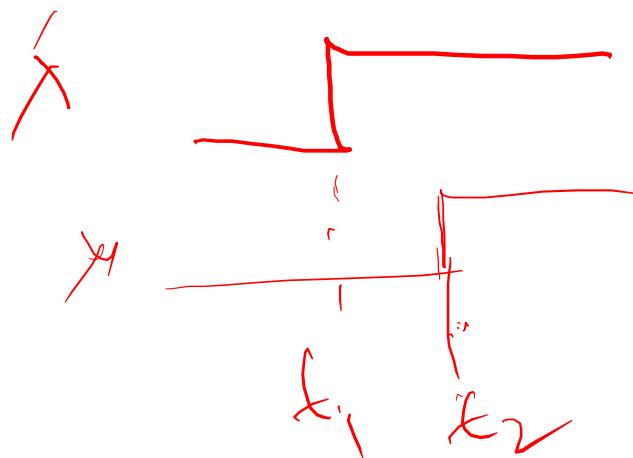
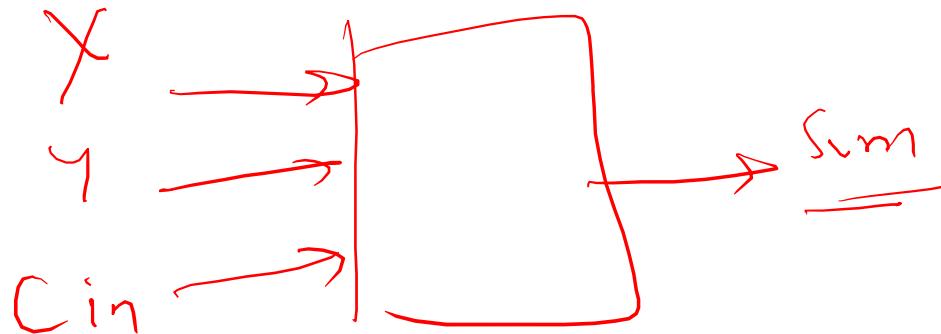
Architecture Equations of FullAdder is

```
begin      -- Concurrent Assignment
    Sum <= X xor Y xor Cin after 10 ns;
    Cout <= (X and Y) or (X and Cin) or (Y
        and Cin) after 15 ns;
end Equations;
```

Entity  
declaration

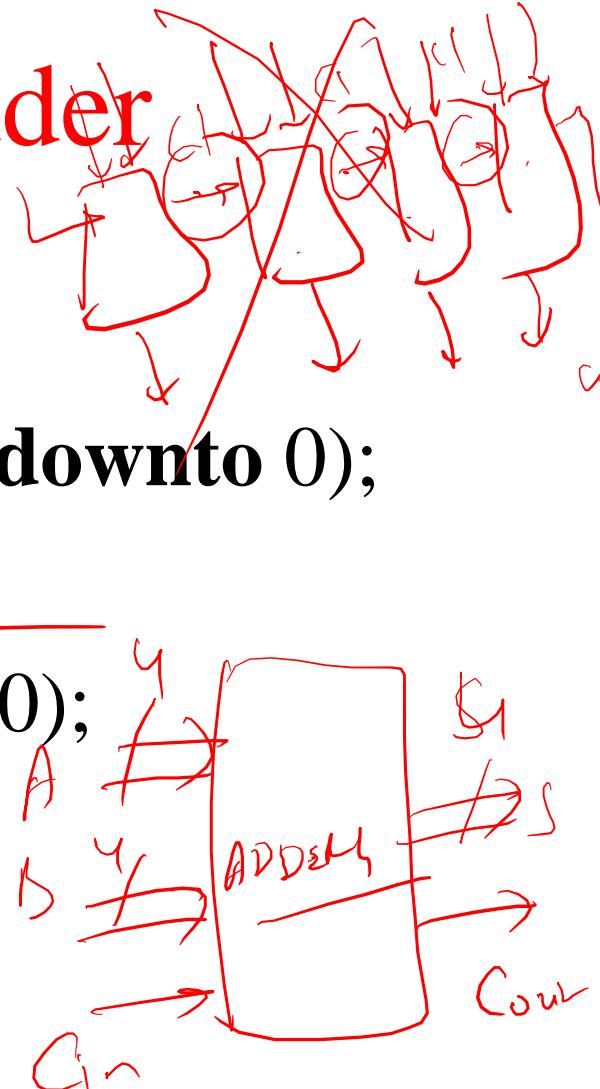
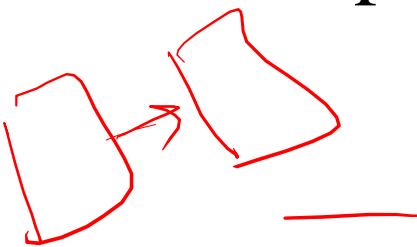


# 1-bit Full Adder Timing



# Example: 4-bit Adder

```
entity Adder4 is
    port (A, B: in bit_vector(3 downto 0);
          Ci: in bit;           -- Inputs
          S: out bit_vector(3 downto 0);
          Co: out bit);        -- Outputs
end Adder4;
```



# Example: 4-bit Adder (contd.)

Architecture Structure of Adder4 is  
Component FullAdder

port (X, Y, Cin: **in** bit; Cout, Sum: **out** bit);

signal C: bit\_vector (3 downto 1);

**begin** -- Instantiations

FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));

FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));

FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));

FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));

**end Structure;**

Section 5: Introduction to VHDL

pushing  
making

C1

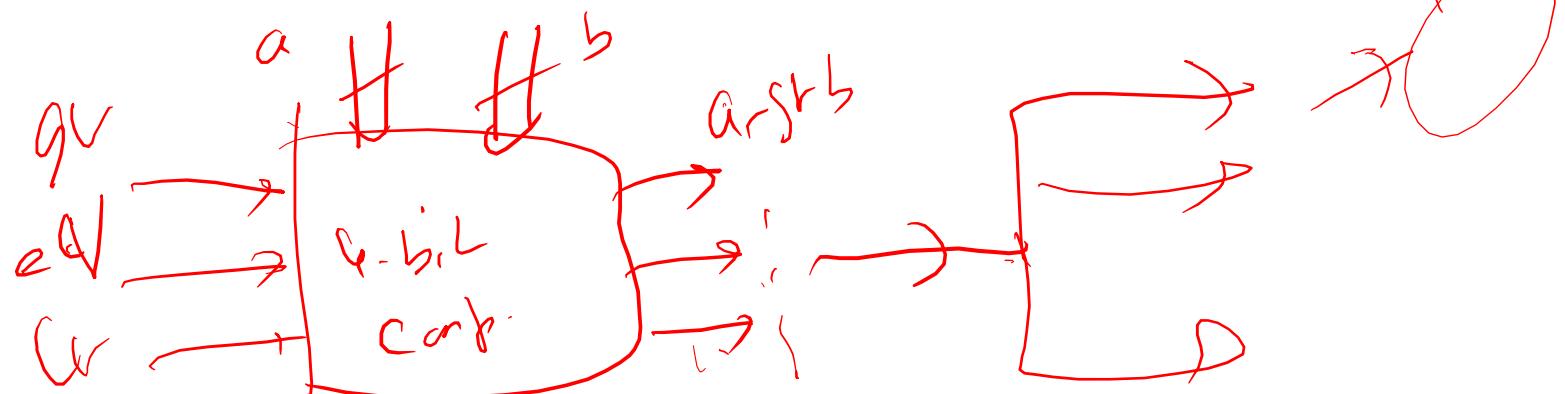
A0  
C2

C3

structural

# Example: 4-bit Comparator

```
entity nibble_comparator is
    port (a, b: in bit_vector (3 downto 0);
          gt, eq, lt : in bit;
          a_gt_b, a_eq_b, a_lt_b : out bit);
end nibble_comparator;
```



Doubt in Syntax ???

# Structural Description (contd.)

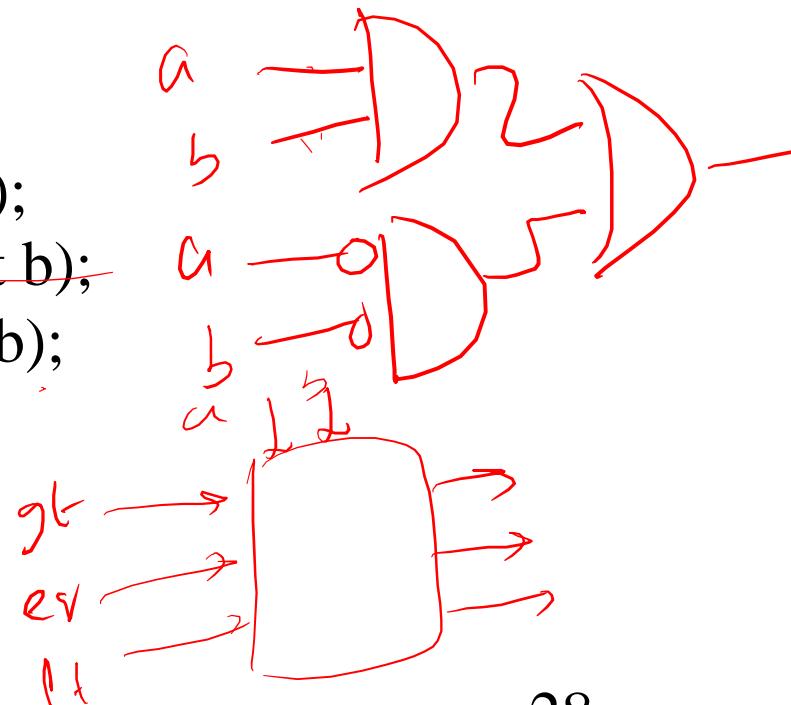
```
architecture iterative of nibble_comparator is
component comp1
    port (a, b, gt, eq, lt : in bit; a_gt_b, a_eq_b, a_lt_b : out bit);
end component;
for all : comp1 use entity work.bit_comparator(gate_level);
signal im: bit_vector (0 to 8);
begin
    c0:comp1 port map(a(0),b(0), gt, eq, lt, im(0), im(1), im(2));
    c1toc2: for i in 1 to 2 generate
        c:comp1 port map(a(i),b(i),im(i*3-3),im(i*3-2),im(i*3-1),
                           im(i*3+0),im(i*3+1),im(i*3+2));
    end generate;
    c3: comp1 port map(a(3),b(3),im(6),im(7),im(8),
                        a_gt_b, a_eq_b, a_lt_b);
end nibble_comparator;
```

Section 5: Introduction to VHDL

# Example: 1-bit Comparator (data flow)

LSB  $\rightarrow$  MSB

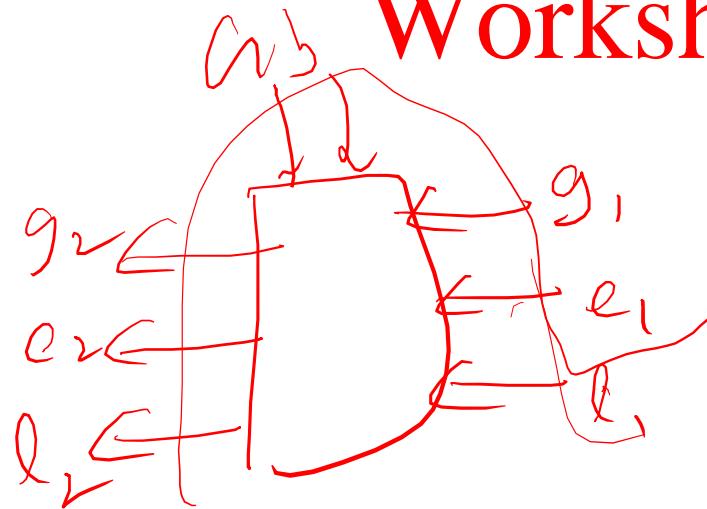
```
entity comp1 is
    port (a, b, gt, eq, lt : in bit; a_gt_b, a_eq_b, a_lt_b : out bit);
end comp1;
architecture dataflow of comp1 is
signal s : bit;
begin
    s <= (a and b) or (not a and not b);
    a_gt_b <= (gt and s) or (a and not b);
    a_lt_b <= (lt and s) or (not a and b);
    a_eq_b <= eq and s;
end dataflow;
```



# References

- Digital Systems Design Using VHDL  
Charles H. Roth, Jr., PWS Publishing Co.  
Chapter 2 (pp. 44 to 84)
- The Designer's Guide to VHDL  
Peter J. Ashenden, Morgan Kaufmann

# Worksheet



$$\left. \begin{array}{l} g_2 = \check{g}_1 + e_1 \cdot (\overline{a}b + \overline{a}\overline{b}) \\ e_2 = e_1 \cdot (ab + \overline{a}b) \\ l_2 = l_1 + e_1 \cdot \overline{a}b \end{array} \right\}$$

# Lecture 29: Behavioral Description in VHDL

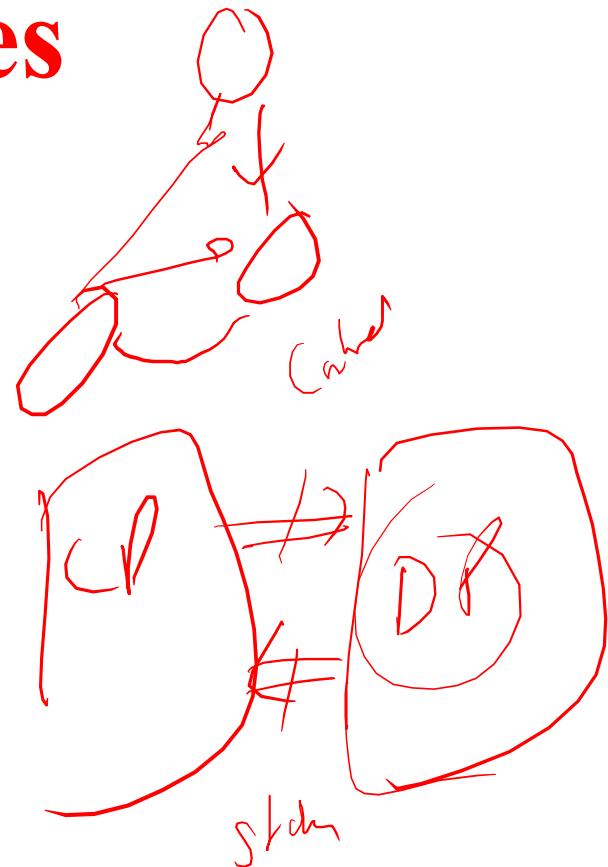
M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

# Modeling Styles

- Semantic model of VHDL
- Structural description ✓
- Data Flow description ✓
- Algorithmic description ✓
- RTL description



# Concurrent Statements in VHDL

- process statement
  - concurrent procedure call
  - ✓ • concurrent signal assign.
  - ✓ • component instantiation
  - ✓ • generate statement
  - block statement
  - concurrent assertion stmt
- behavior ✓
  - behavior
  - data flow
  - structure
  - structure
  - nesting
  - error check

# Example: D Flip-Flop

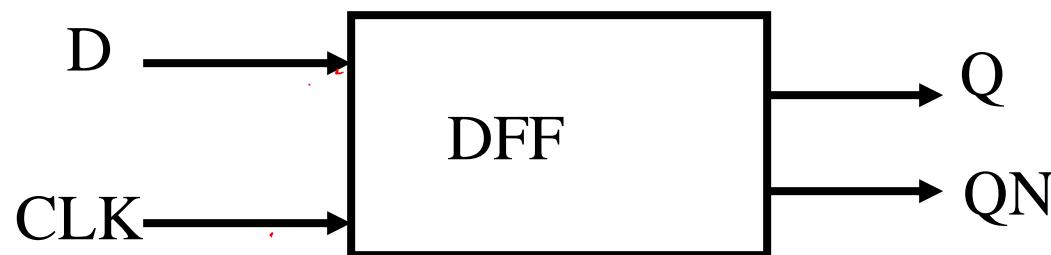
entity ~~DFF~~ is

~~port (D, CLK: in bit;~~

~~Q: out bit; QN: out bit := '1' );~~

end ~~DFF;~~

~~:= '0' )~~



## Example: DFF (contd.)

Architecture Beh of DFF is

~~begin process (CLK)~~

~~begin if (CLK = '1') then~~

~~Q <= D after 10 ns;~~

~~QN <= not D after 10 ns;~~

~~endif;~~

~~endprocess;~~

~~end Beh;~~

*Sensitivity list*

Section 5: Introduction to VHDL

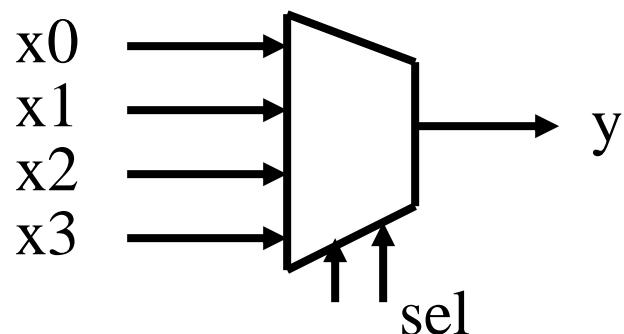
# Timing Diagram

# Data Flow & Process Statement



# Concurrent Conditional Assignment: 4 to 1 Multiplexer

```
y <=      x0    when      sel = 0  
      else x1    when      sel = 1  
      else x2    when      sel = 2  
      else x3    when      sel = 3
```



# CASE Statement: 4 to 1 Multiplexer

**Case** sel is

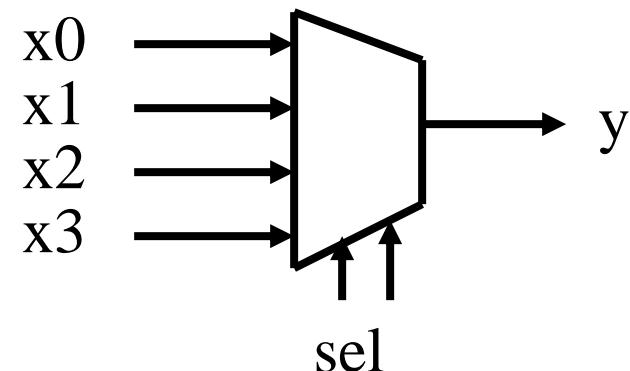
**when** 0 => y <= x0

**when** 1 => y <= x1

**when** 2 => y <= x2

**when** 3 => y <= x3

**end case**



**Note** allowed only within a Process statement

# Variables And Signals

Architecture var of dummy is  
signal trigger, sum: integer := 0;

begin process

variable var1: integer:= 1;

variable var3, var2: integer:= 2;

begin wait on trigger;

var3 := var1 + var2;

var1 := var3;

sum <= var1;

end process; end var;

: bit;

:=

:=

var3 3

var1 3

sum 3

$A = b + c$

# Variables and Signals

Architecture sig of dummy is

```
signal trigger, sum: integer := 0;  
signal sig1: integer:= 1;  
signal sig3, sig2: integer:= 2;
```

begin process

begin wait on trigger;

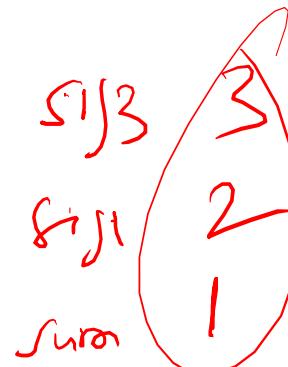
sig3 <= sig1 + sig2;

sig1 <= sig3 + sig2;

sum <= sig1;

end process; end sig;

: bit



# Inertial and Transport Delays

~~y1 <= transport x after 5 ns; -- transport delay~~

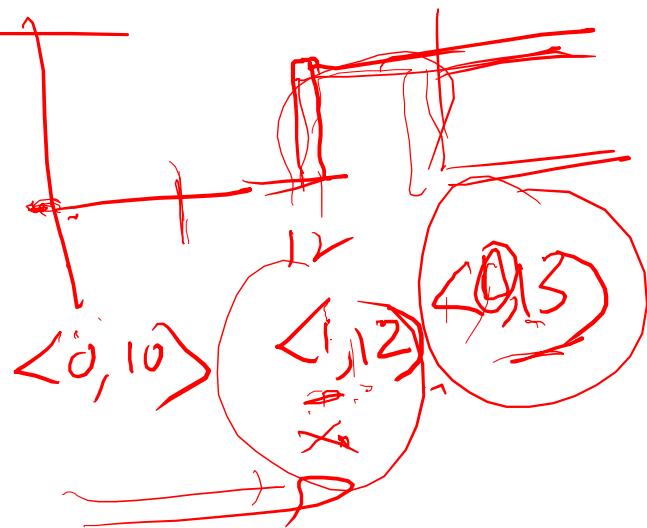
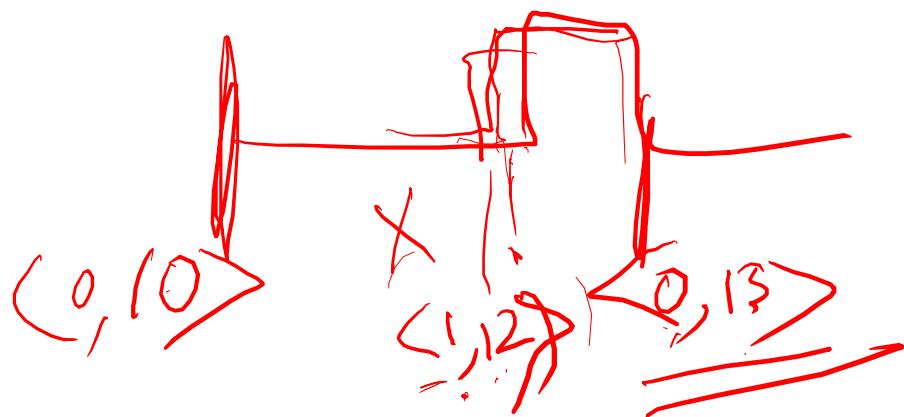


~~y2 <= x after 5 ns; -- inertial delay~~

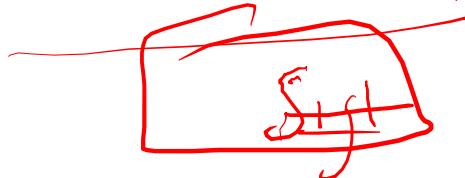
~~y3 <= reject 2 ns x after 5 ns; -- mix of inertial and transport delay but rejection of pulse less than 2 ns~~

# Inertial & Transport Delays (contd)

- ✓ Inertial delay can reject narrow input pulses/spikes whereas transport delay would preserve them in the output



Proses (trigger)



Endproses

$\langle v_1, t_1 \rangle$

$\langle v_2, t_2 \rangle$

h:  
;

# Worksheet

Borne



uitvoering

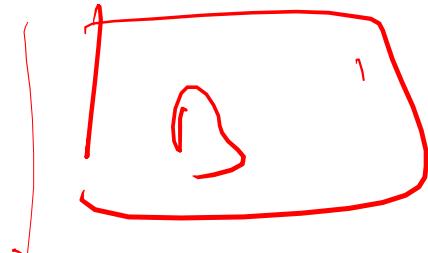
Endproses

Son

2ns

3ns

Process(trigger)



Endprocess



Process

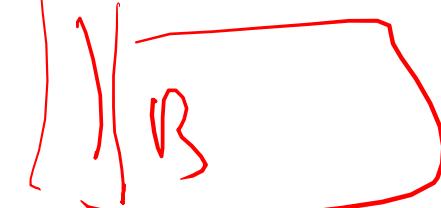
wait on trigger



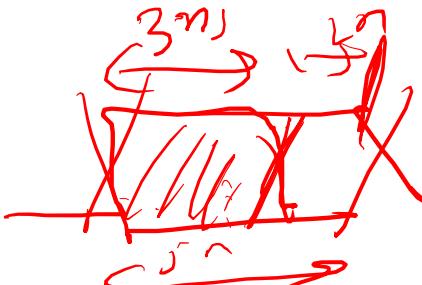
Endprocess

begin  
wait

Process



wait on trigger  
Endprocess



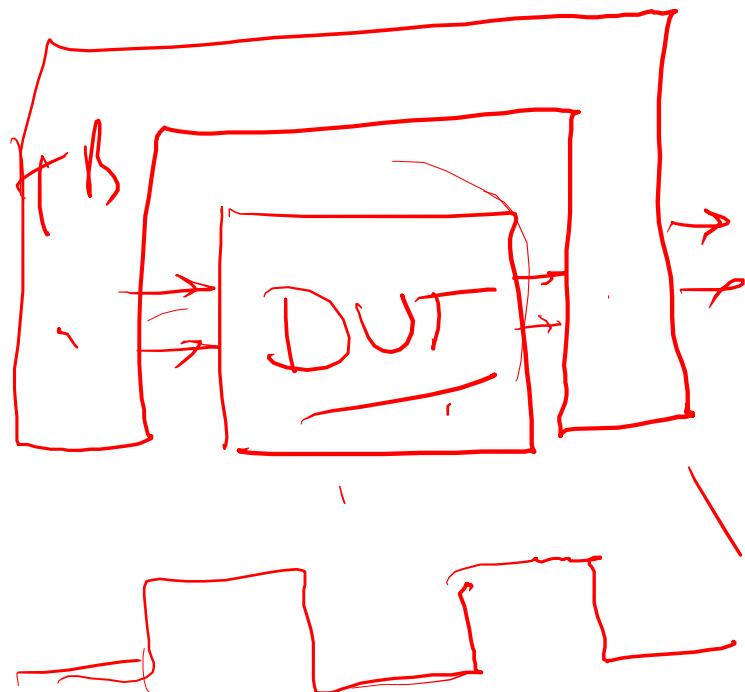
- 2 ns 3 ns

## Section 5: Introduction to VHDL

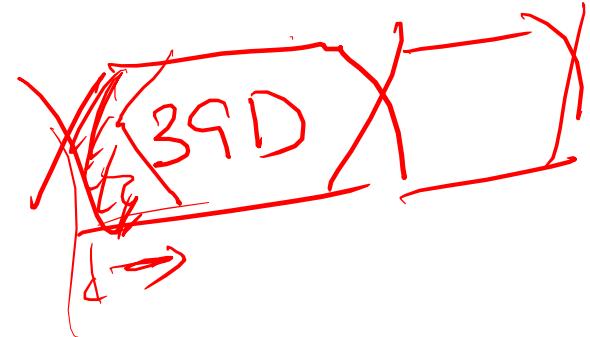
# Worksheet

Synthesizable

Testbench



Process



$a \leftarrow 0;$

wait for 5ns

$a \leftarrow 1$

wait for 5ns

- end process

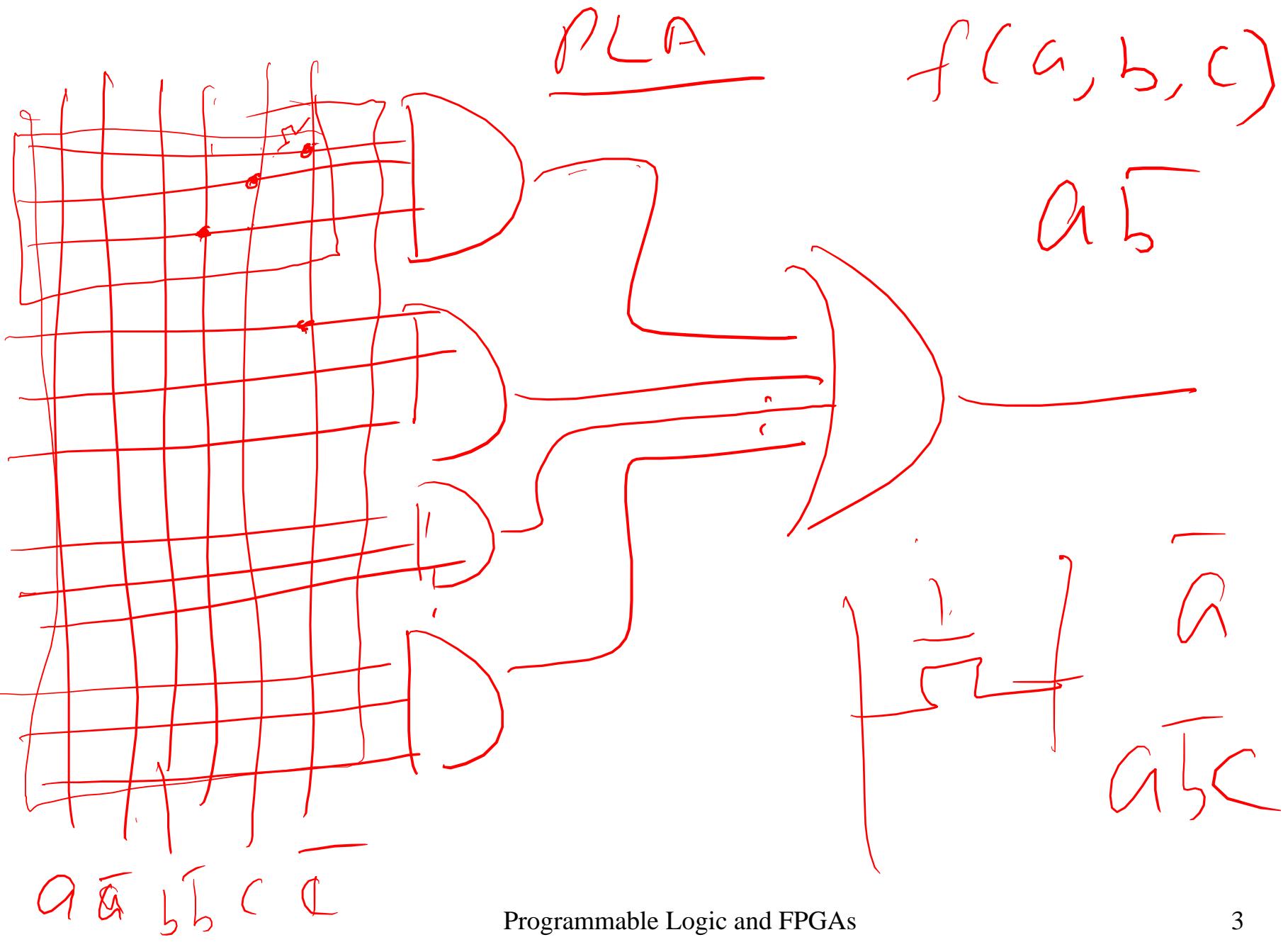
# Lecture 32: Programmable Logic

M. Balakrishnan

# Logic Implementation Options

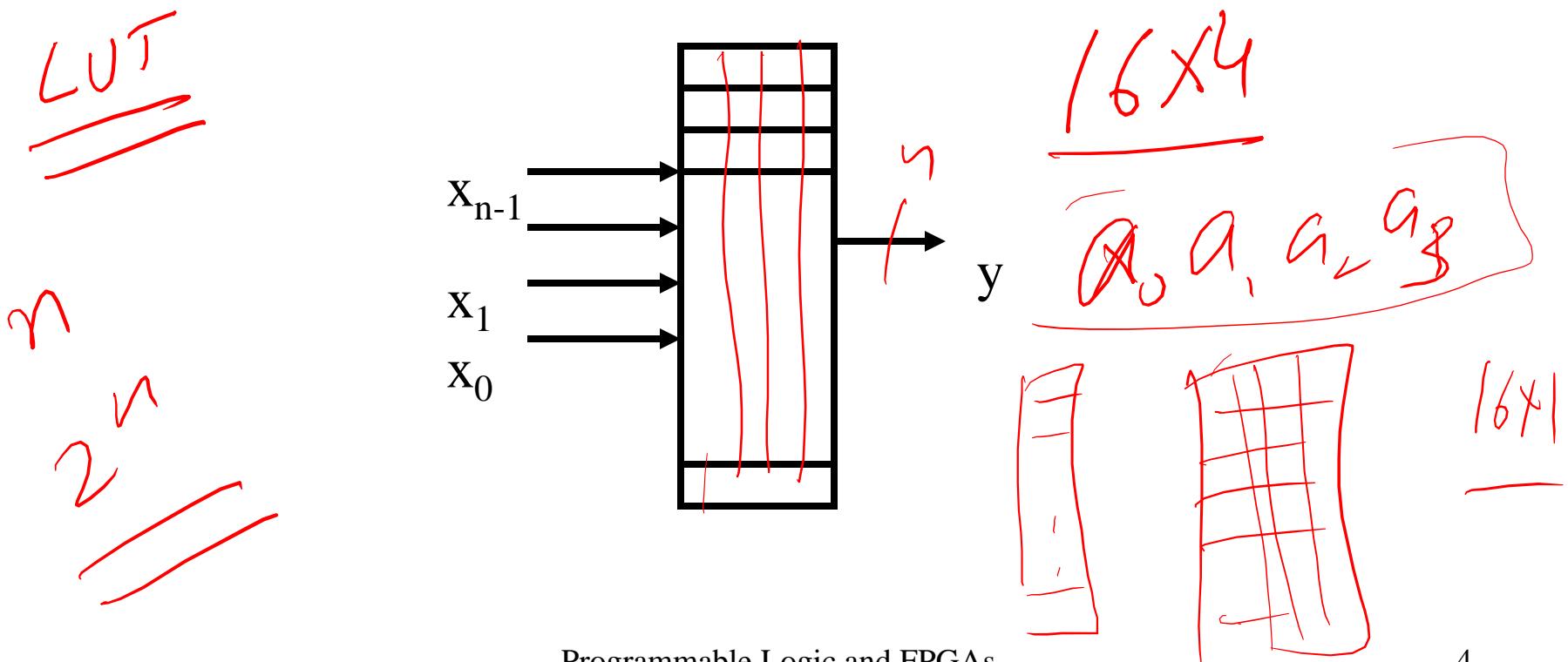
To implement a n-variable function (with k minterms)

- n to  $2^n$  decoder + k input OR gate
- $2^n:1$  multiplexer
- $2^{(n-1)}:1$  multiplexer + 1 inverter
- $2^n \times 1$  ROM
- $n \times p \times 1$  PLA with  $p \leq k$
- **FPGA**

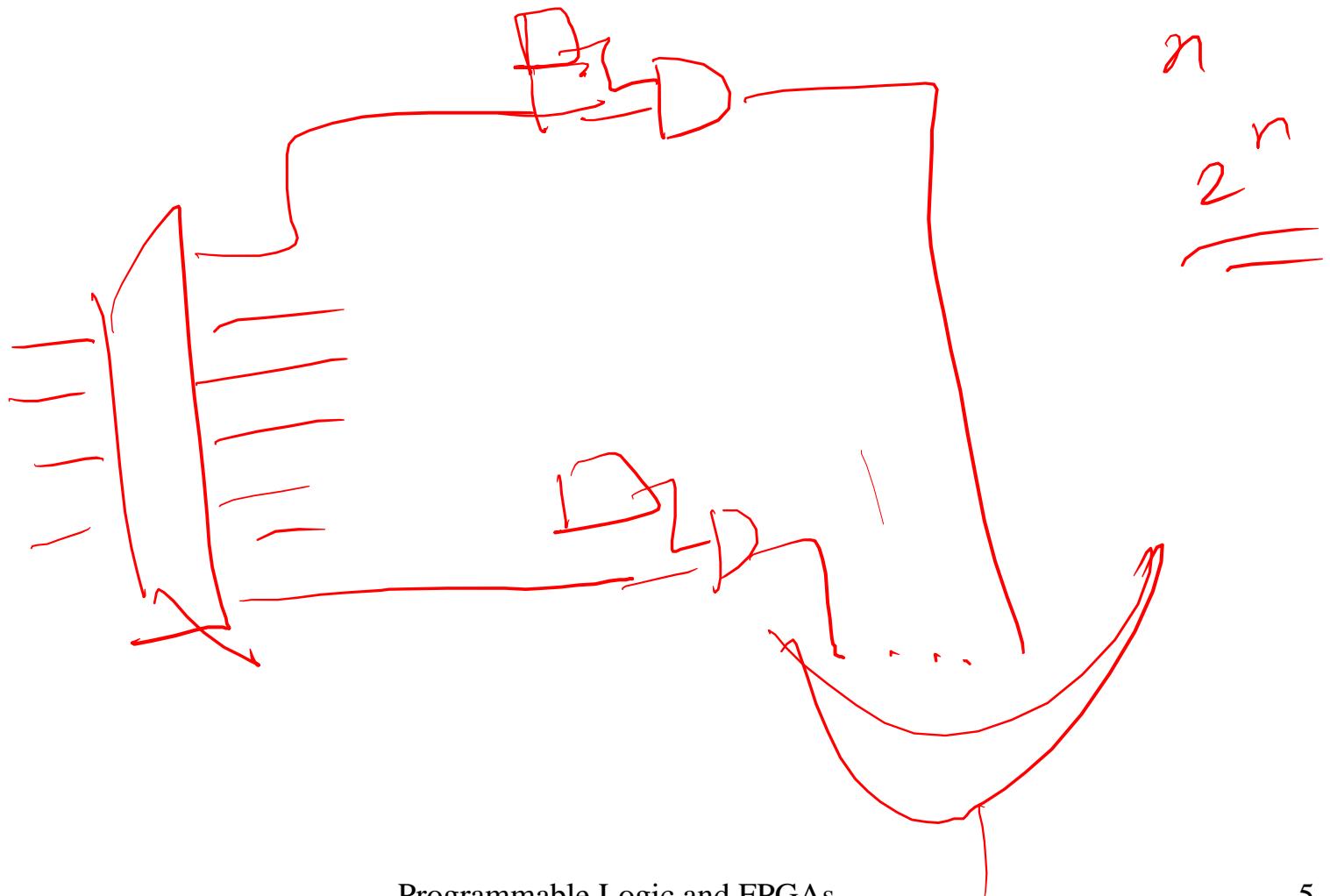


# ROM (Read Only Memory)

$2^n \times m$  ROM (n address and m outputs)

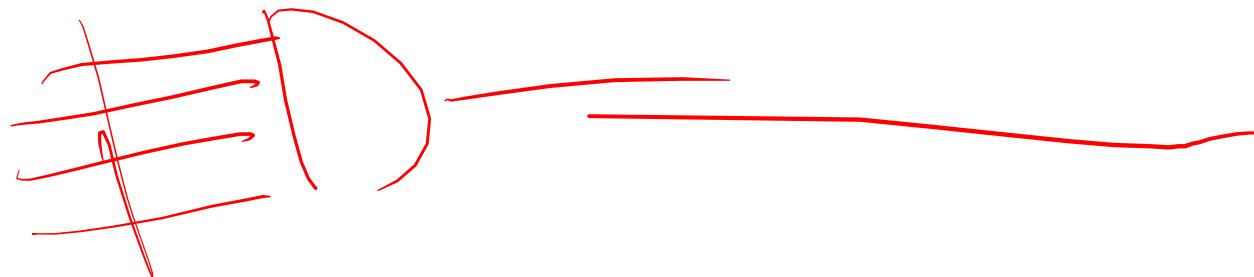


# ROM Design Example



# Implementing Logic Using ROMs

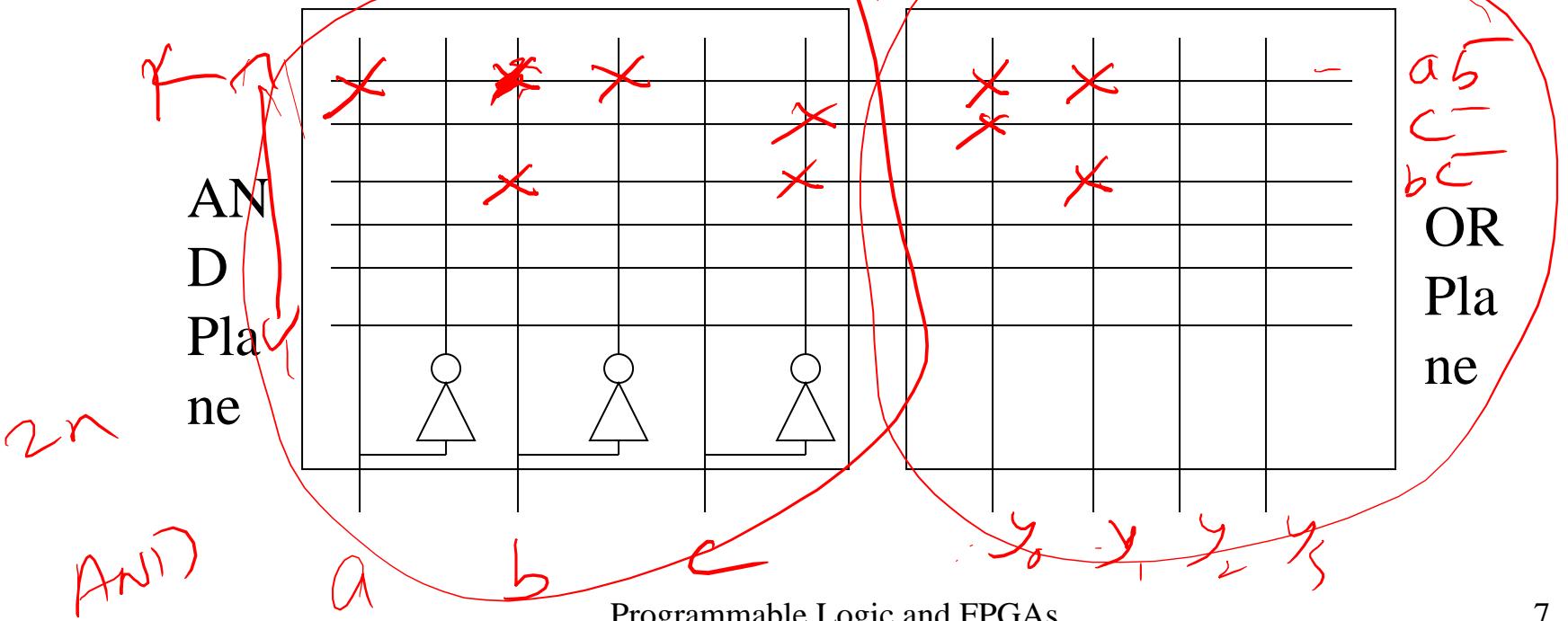
- Direct implementation of the function
- Extremely flexible as reprogramming can implement a completely different function



# PLA (Programmable Logic Arrays)

PLA Specification:  $n \times k \times m$

(N inputs, K product terms and M outputs)



# Implementing Logic Using PLAs

$N_{AND}$   $N_{OR}$

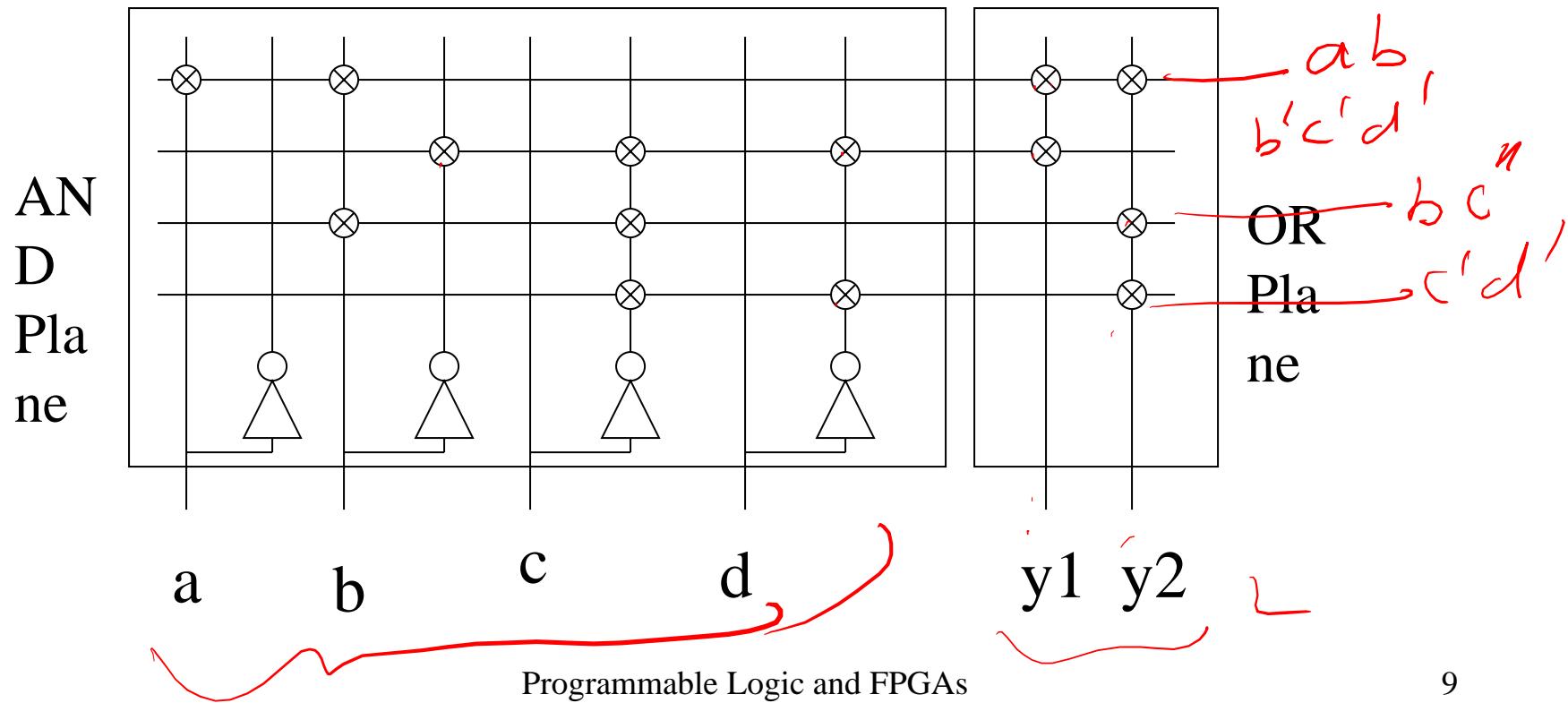
- Direct implementation of SoPs
- Implement product terms in the AND plane
- Implement sum in the OR plane

$K$  AND Gates with  $\underline{2n}$  inputs —  $\boxed{K \times 2n}$   
 $m$  OR Gates with  $\underline{k}$  inputs —  $\boxed{m \times K}$

# Implementing Logic Using PLAs

(Example)

$$y_1 = ab + b'c'd'$$
$$y_2 = ab + bc' + c'd'$$



# Programmable Devices

- Prefabricated Silicon
- Logic implemented by programming the basic cells and the interconnect
- Very fast *turnaround* time
- Limited design flexibility
- Low development *time* and *cost*

ASIC

# Why FPGA?

ASIC

Custom logic implemented as **ASICs** have the following merits and demerits

## Pros:

- a. Reduced system complexity.
- b. Improved performance.

## Cons:

- a. Very expensive to develop.
- b. Delay introduction of product to market (time to market) because of increased design time.

# Why FPGA (contd.)?

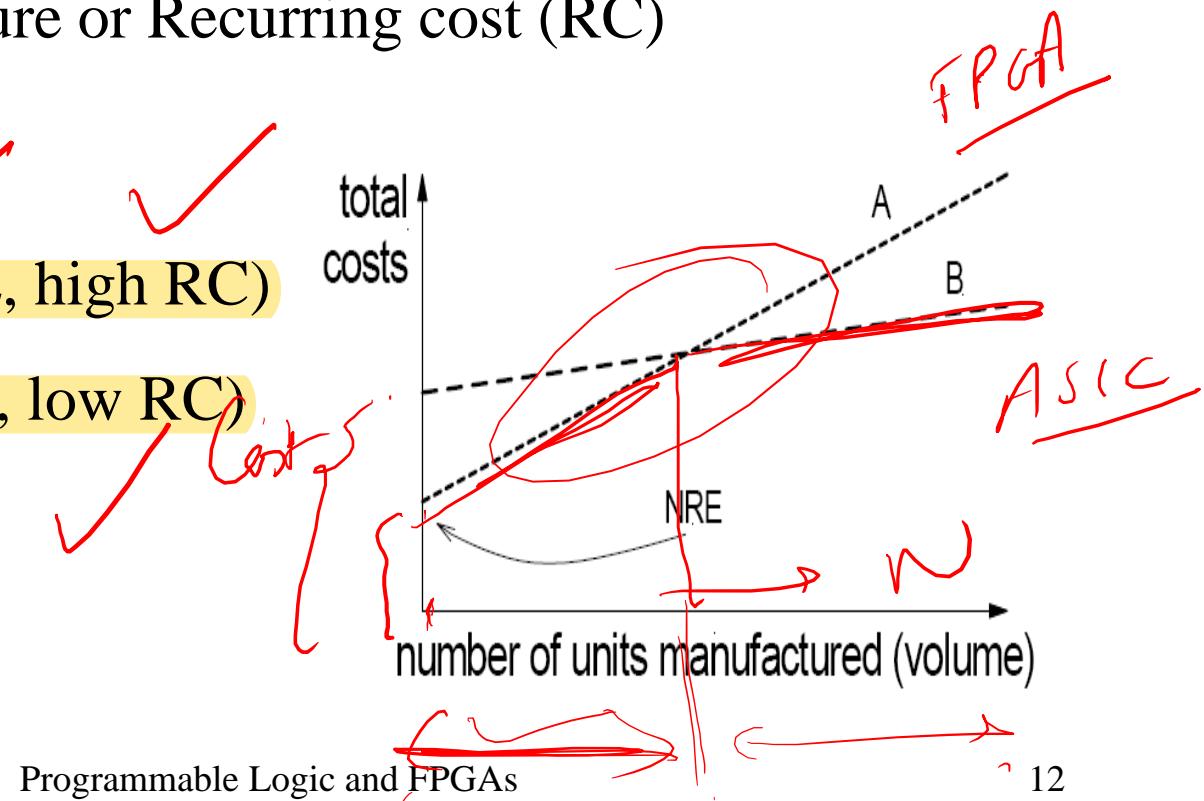
Need to worry about two kinds of costs:

- Cost of development, sometimes called non-recurring engineering (NRE)
- Cost of manufacture or Recurring cost (RC)

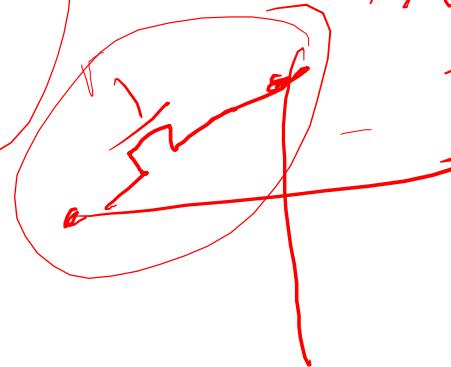
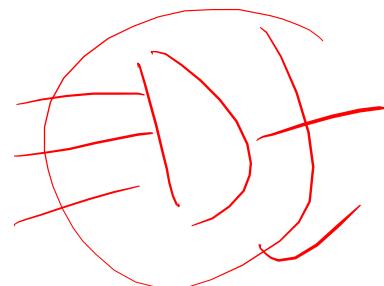
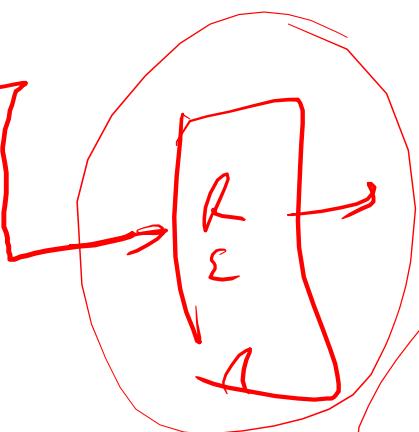
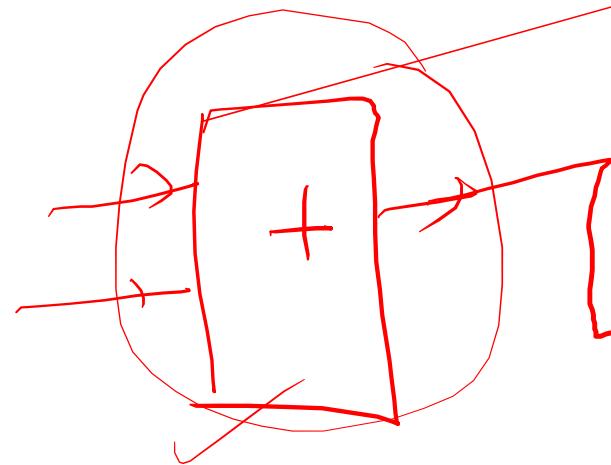
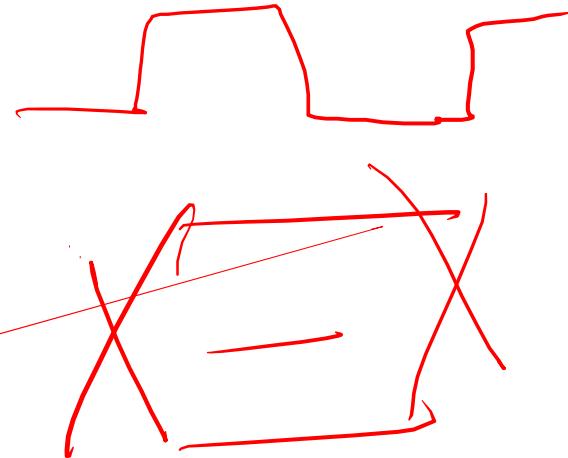
A: FPGAs (low NRE, high RC)

B: ASICs (high NRE, low RC)

PLW  
Process

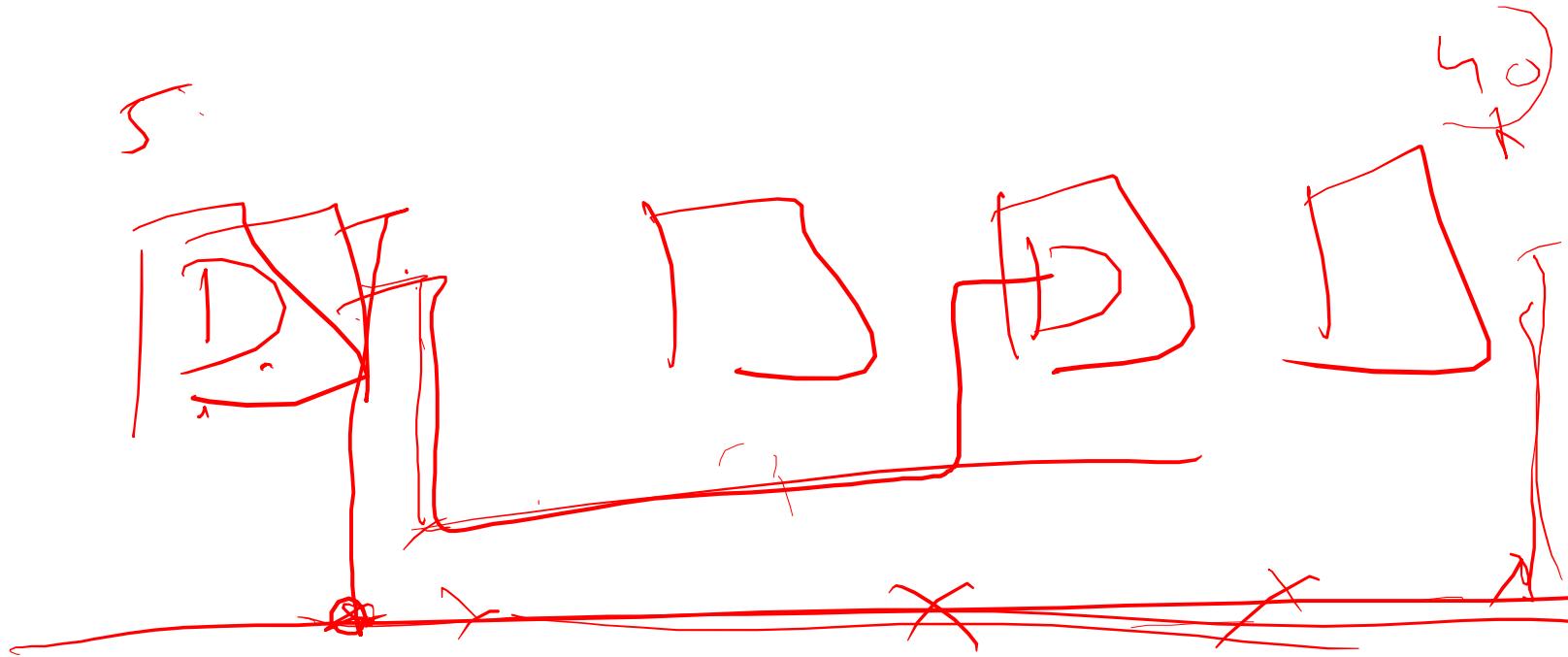


# Lecture 33



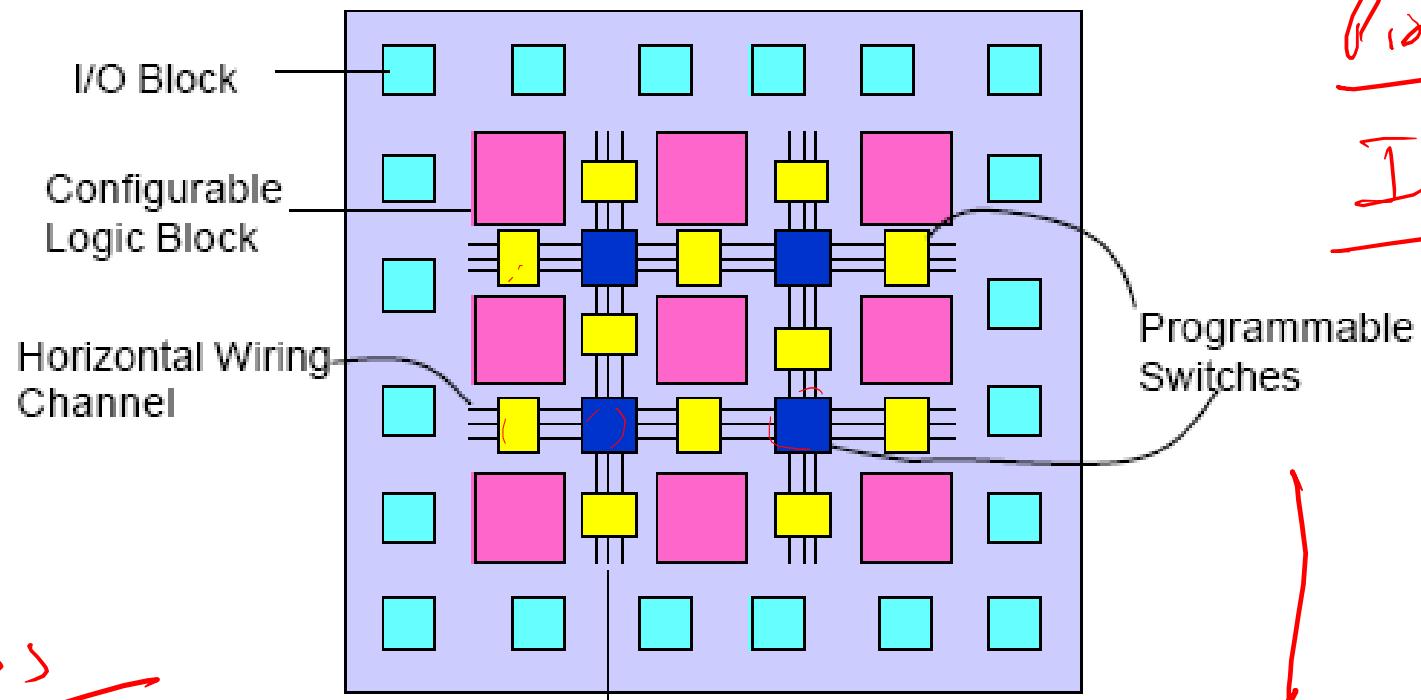
CUT

Truth table

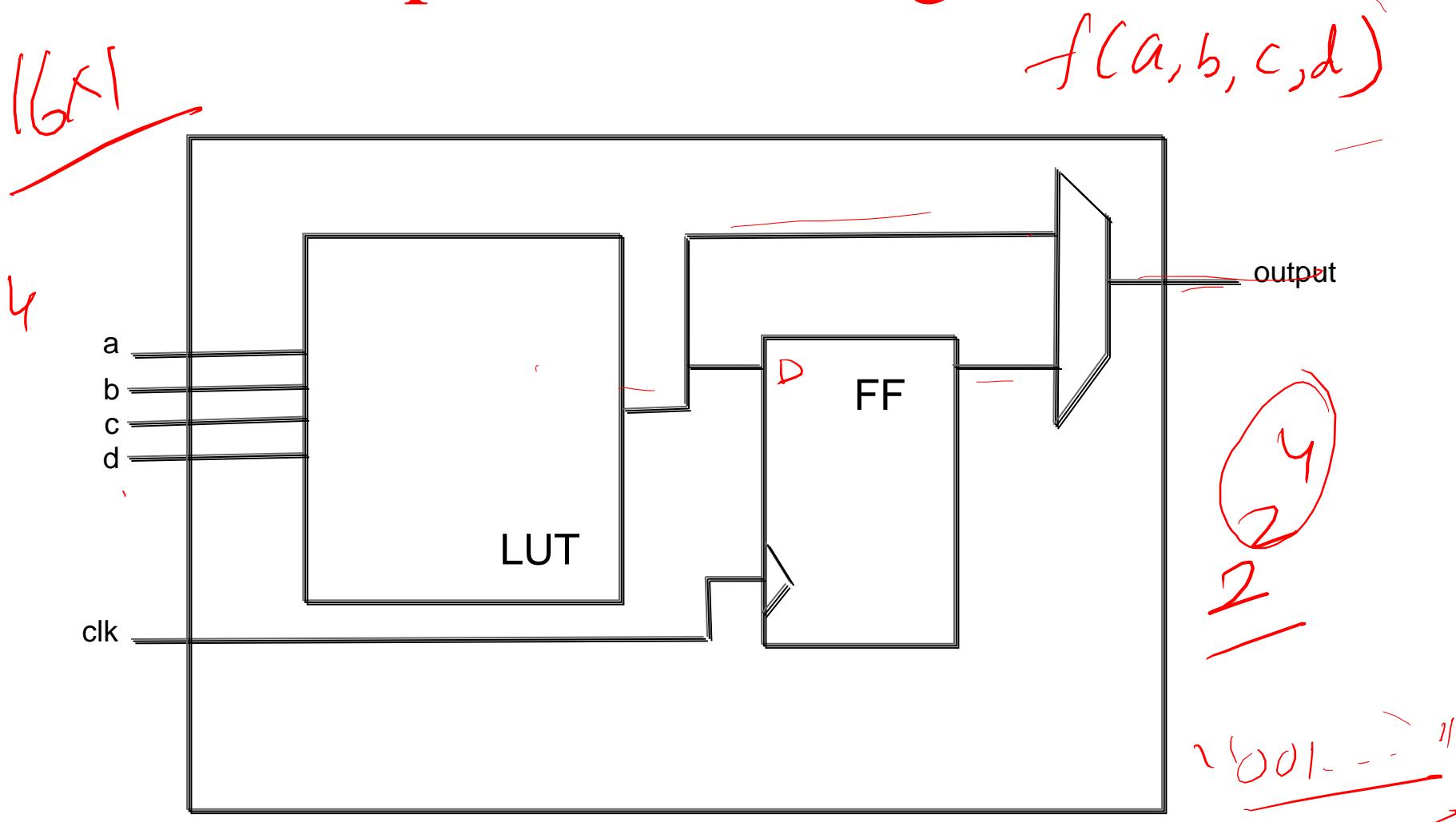


XILINX

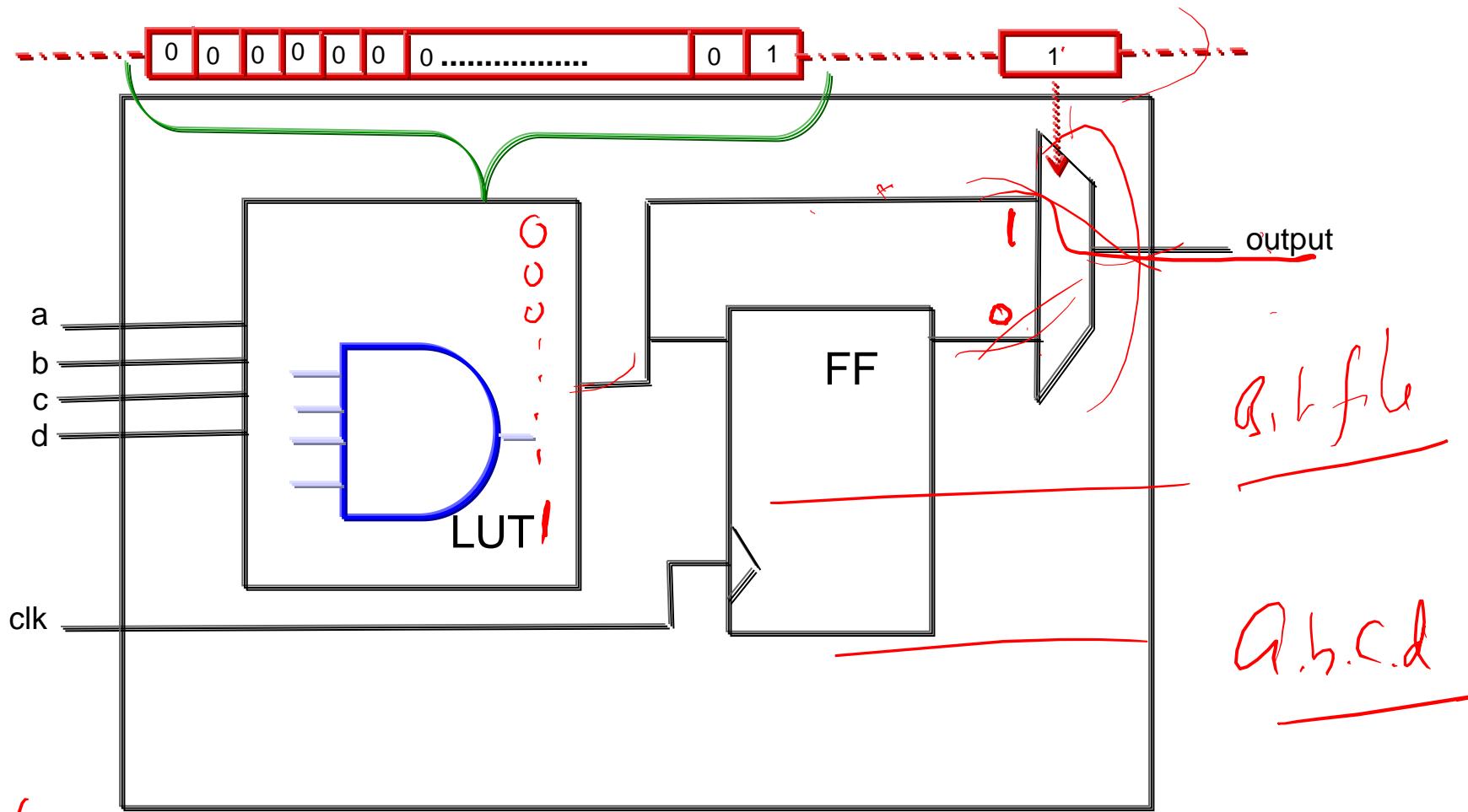
# FPGA Blocks



# A Simple FPGA Logic Block

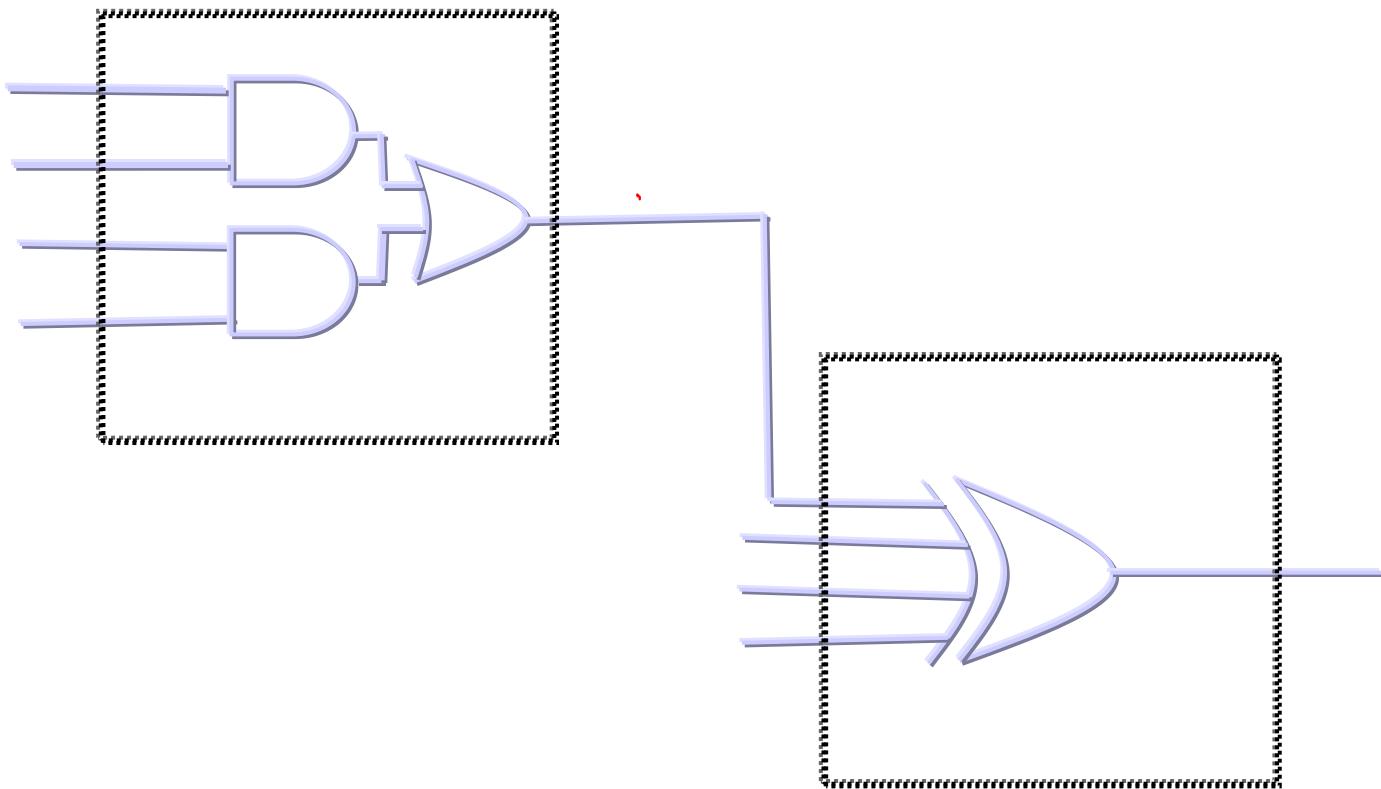


# A Simple FPGA Logic Block

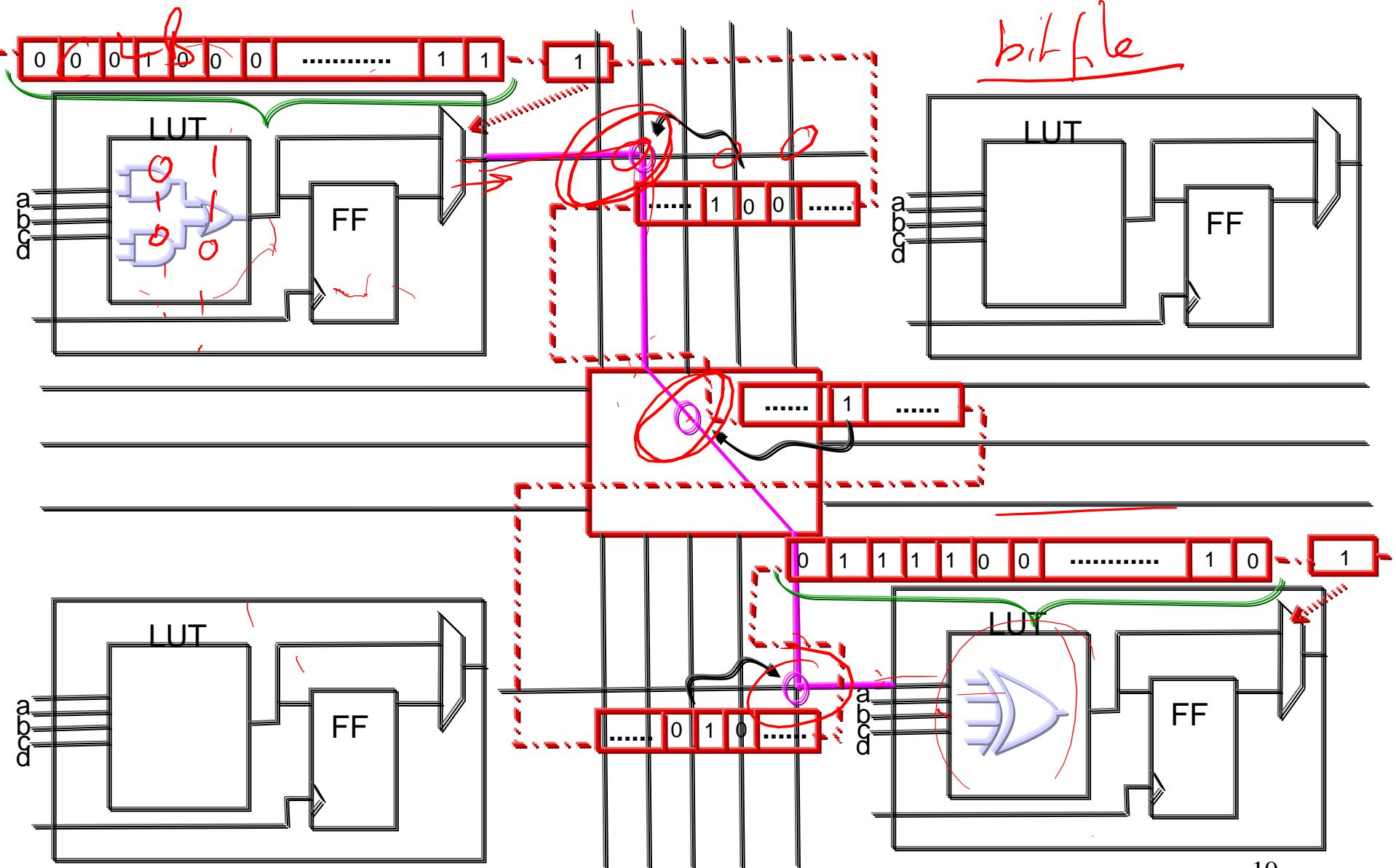


MEC  
Decoder ✓

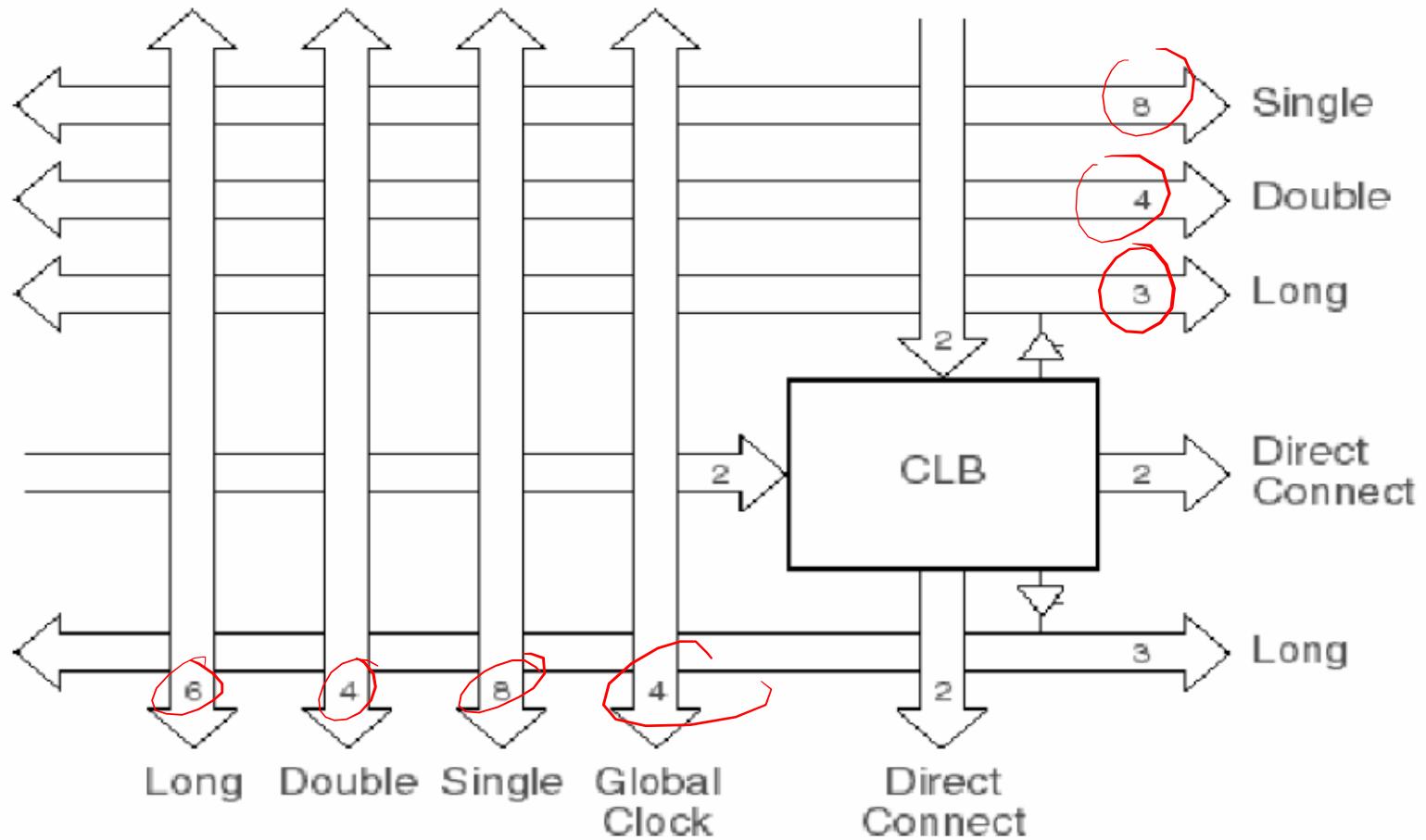
# Simple Circuit



# LUT Configuration bits

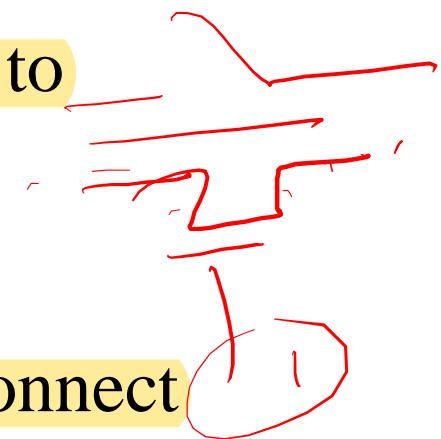


# Interconnections

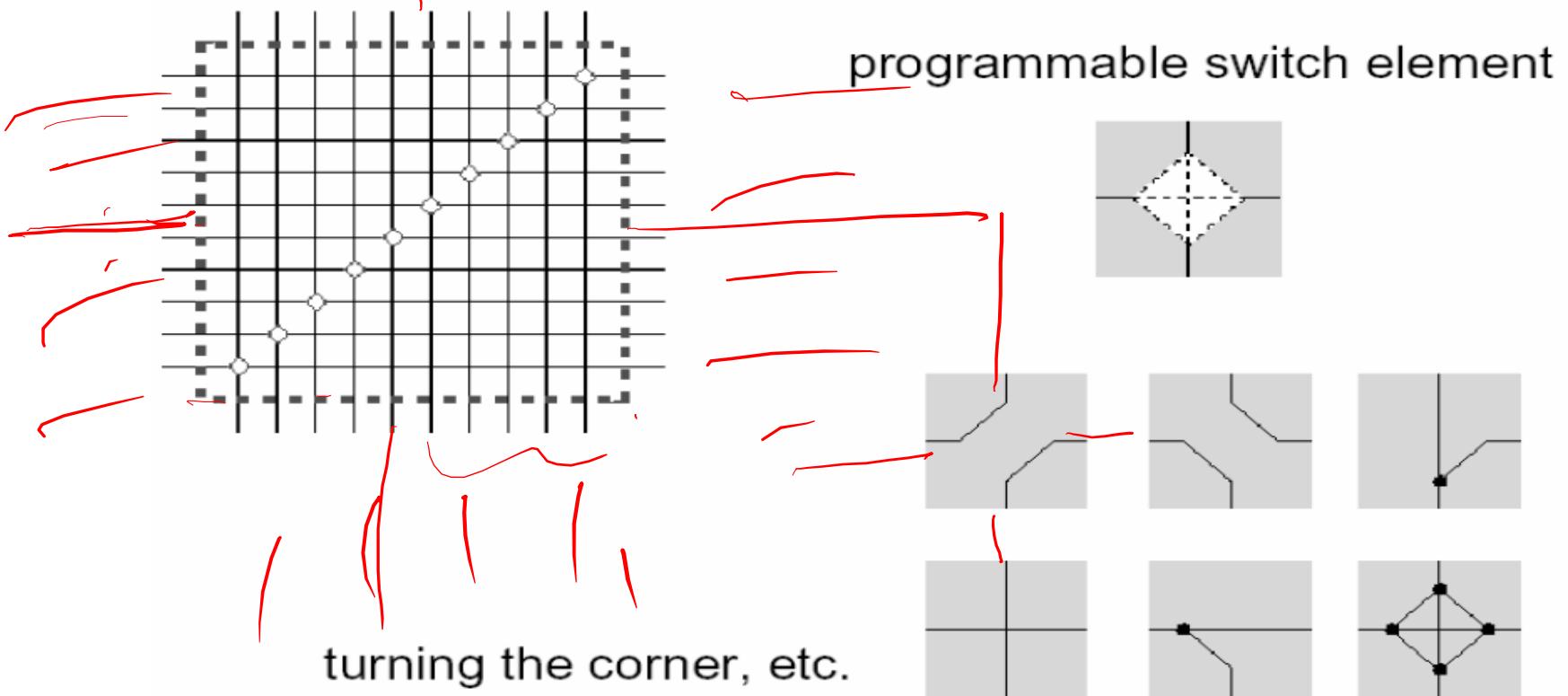


# Programmable Interconnects

- **Connection box**
  - Connects input/output of logic block to interconnect channels.
- **Switch box**
  - Enables the connection of two interconnect lines.
- **Transmission gate (or a pass transistor) is used for each connection.**



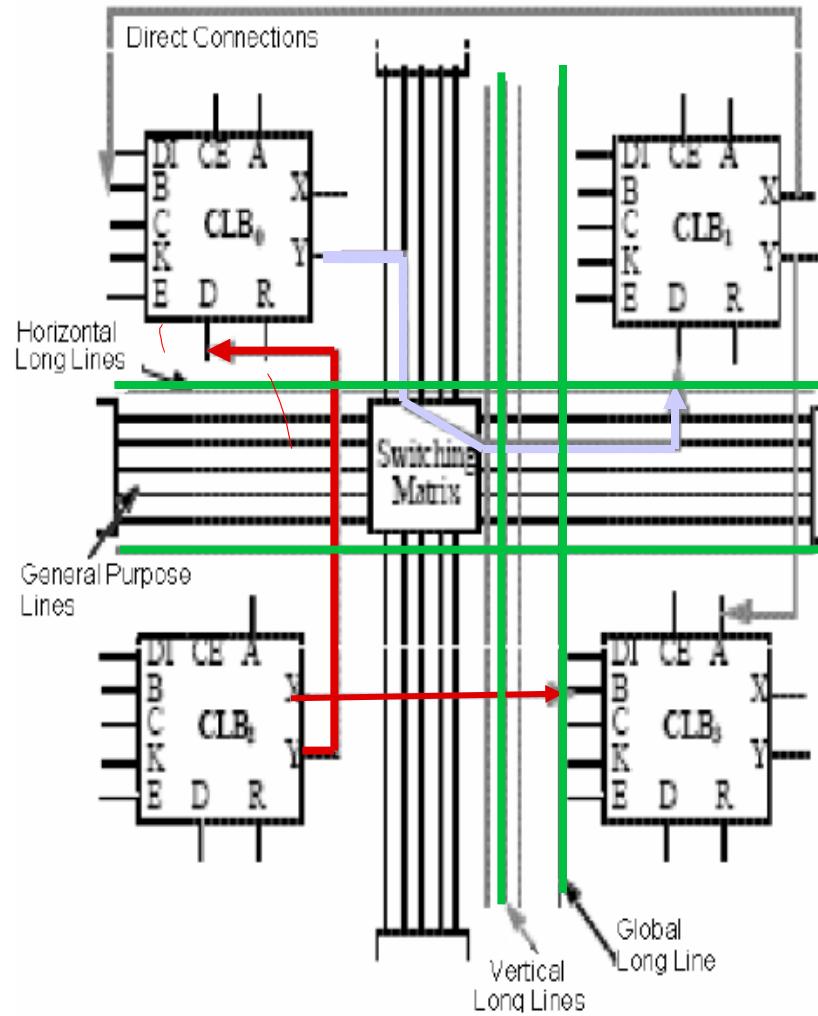
# Programmable Switching Matrix



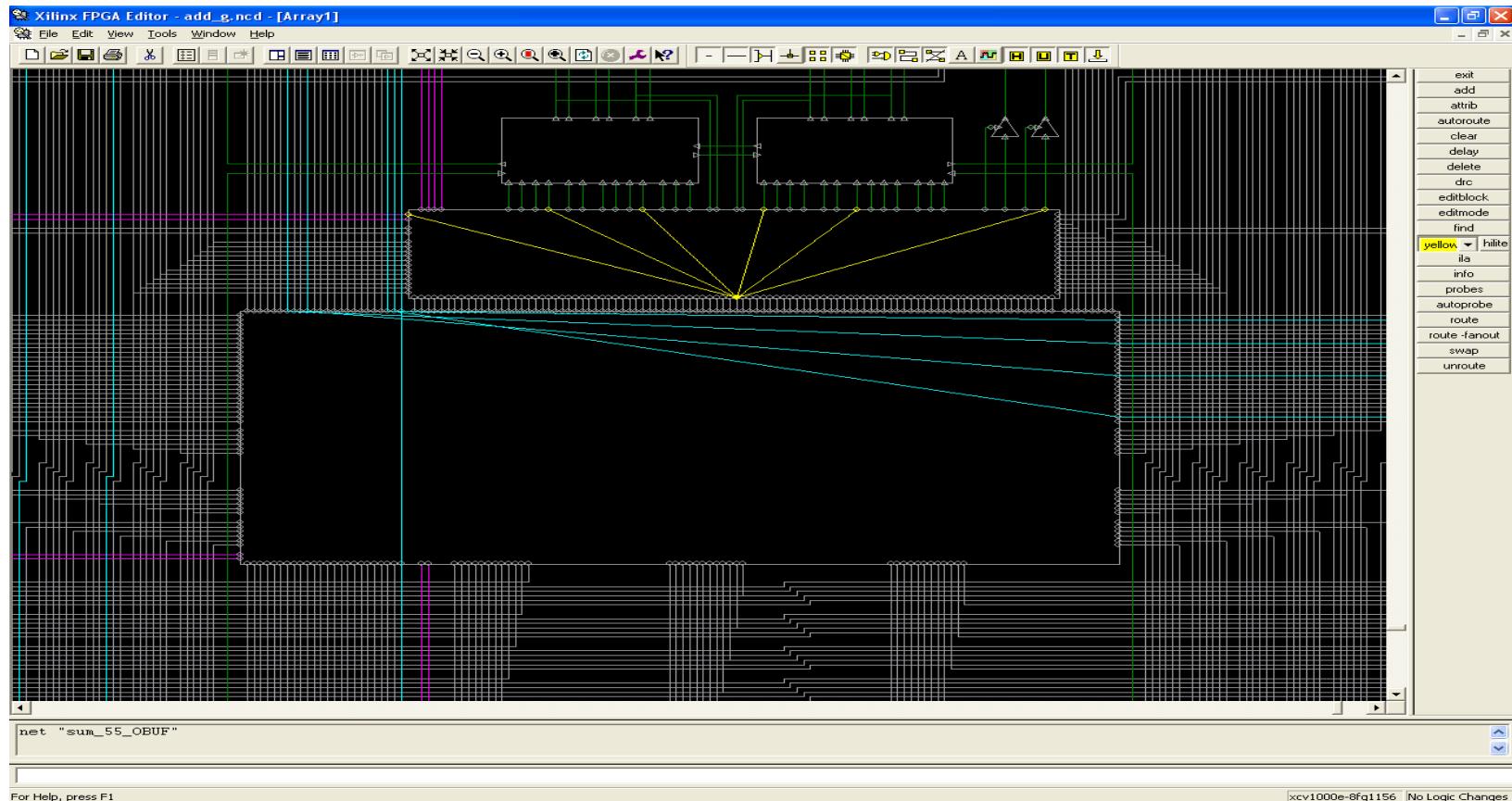
# Methods of Interconnection

## Direct Interconnect

- Geared for purpose
- Independent of logic
- Long line interconnect through direct interconnects
- Time critical signals
- Interconnections and without going through switch boxes
- Horizontal, vertical & global long lines → very fast
- Global long lines for clocks and resets

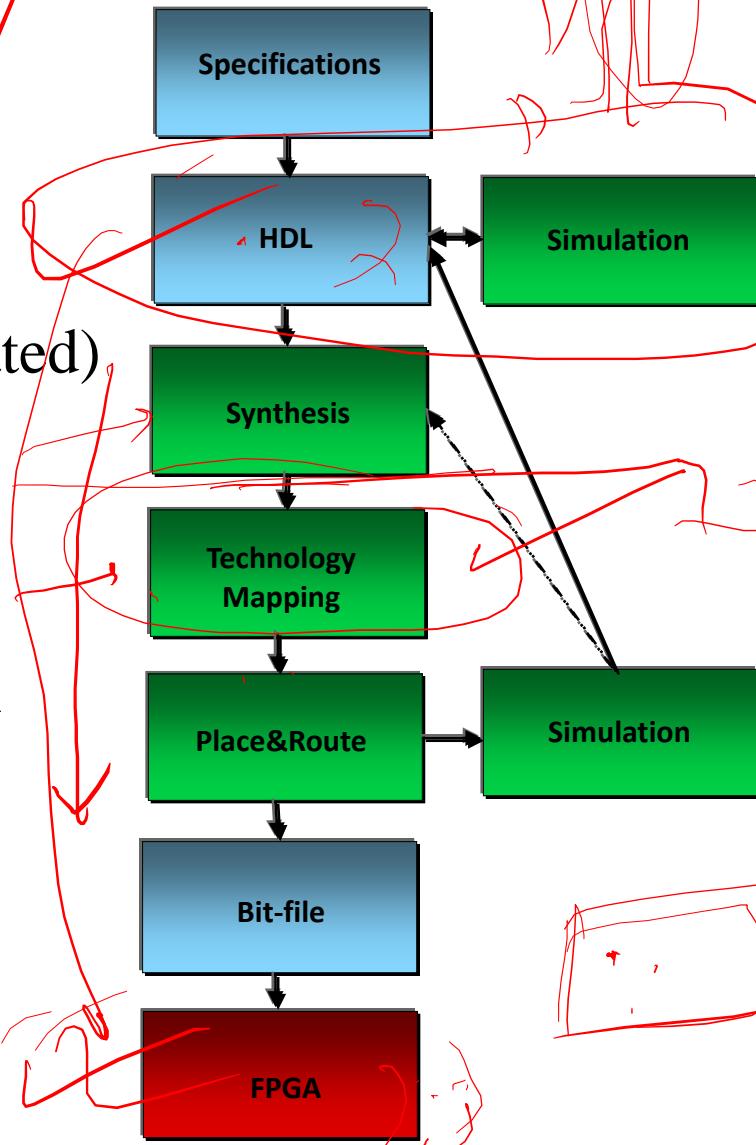


# Programmable Interconnects



# FPGA Design Flow

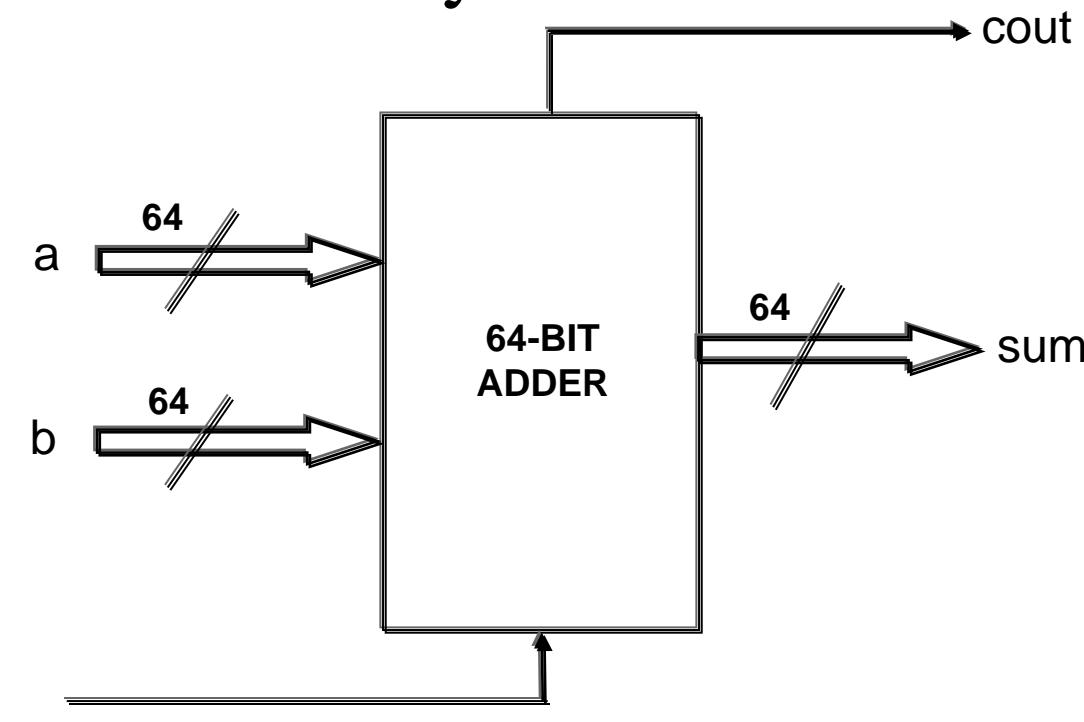
- Specifications of Design.
- Converting into HDL.
- Synthesis steps (mostly automated)
  - Synthesize Design
  - Map design
  - Placing design inside FPGA
  - Routing design inside FPGA
  - Generate bit-stream



Programmable Logic and FPGAs

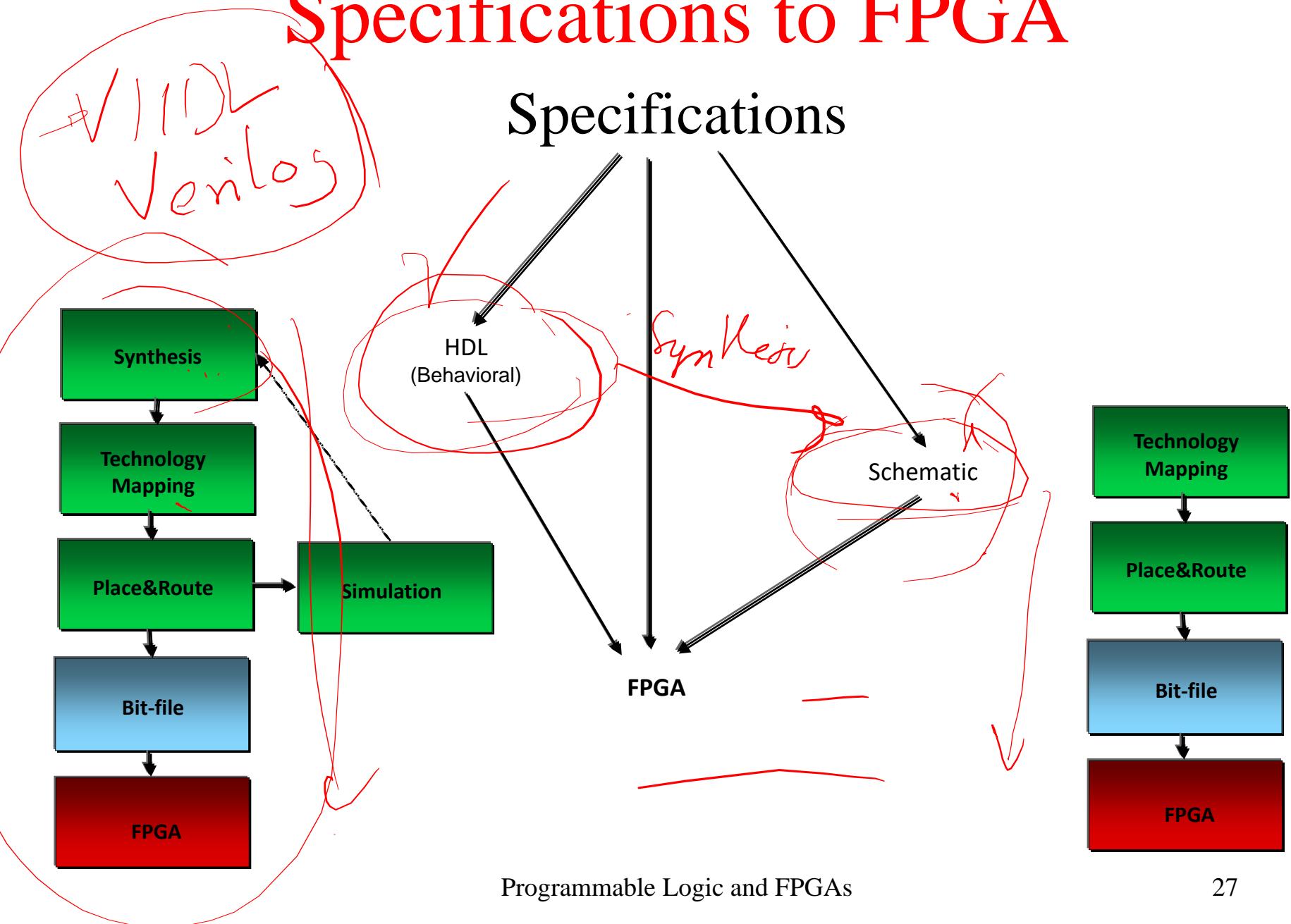
# Specifications

- To add two 64-bit binary numbers with an additional carry in and generate a 64-bit output and 1-bit carry out.



# Specifications to FPGA

## Specifications



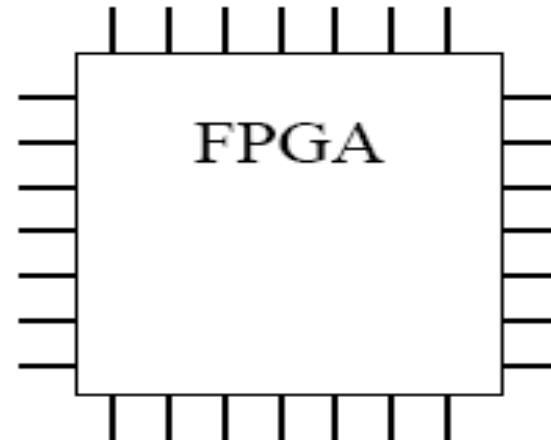
# Specs (VHDL) to FPGA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

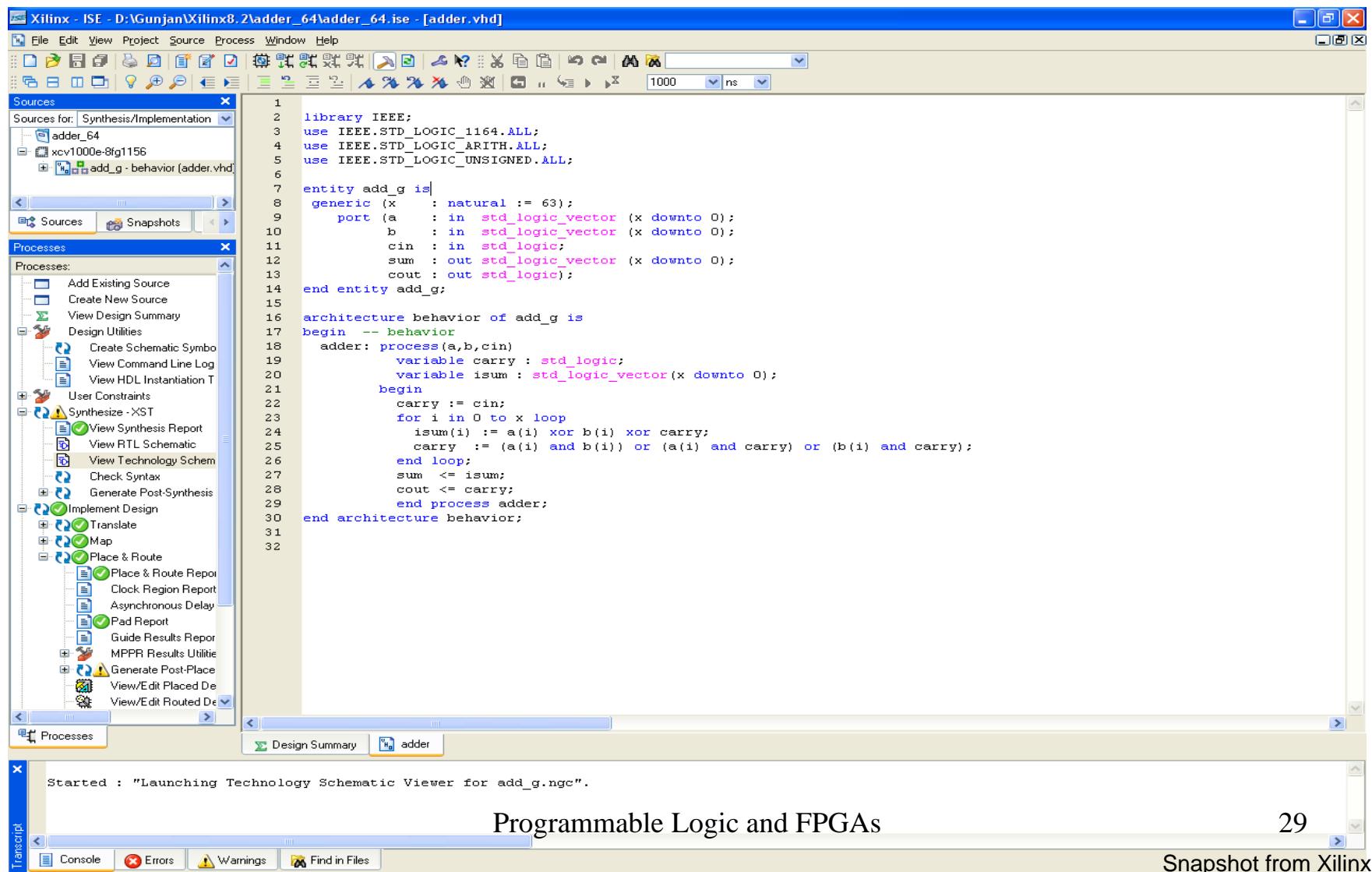
entity add_g is
generic (x : natural := 63);

port (a : in std_logic_vector (x downto 0);
      b : in std_logic_vector (x downto 0);
      cin : in std_logic;
      sum : out std_logic_vector (x downto 0);
      cout : out std_logic);
end entity add_g;

architecture behavior of add_g is
begin -- behavior
  adder: process(a,b,cin)
  variable carry : std_logic;
  variable isum : std_logic_vector(x downto
    0);
  begin
    carry := cin;
    for i in 0 to x loop
      isum(i) := a(i) xor b(i) xor carry;
      carry := (a(i) and b(i)) or (a(i) and
        carry) or (b(i) and carry);
    end loop;
    sum <= isum;
    cout <= carry;
  end process adder;
end architecture behavior;
```



# Design Entry



# Synthesizing the Design

- Synthesis : Optimization process of adapting a logic design to the logic resources available on the chip, like lookup tables, Long line, and dedicated carry.
- It means analyzing the whole design, and selecting which logic resources available in the FPGA will be used to perform the task.
- Gate level netlist is the output file.

# Synthesis - Example

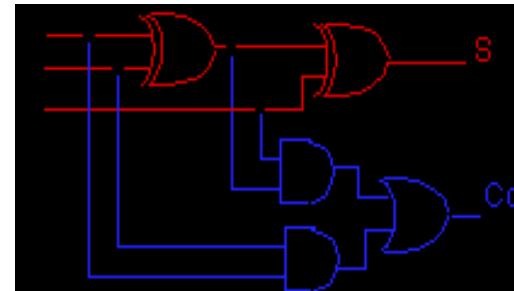
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add_g is
generic (x : natural := 63);

port (a : in std_logic_vector (x downto 0);
      b : in std_logic_vector (x downto 0);
      cin : in std_logic;
      sum : out std_logic_vector (x downto 0);
      cout : out std_logic);
end entity add_g;

architecture behavior of add_g is
begin -- behavior
  adder: process(a,b,cin)
  variable carry : std_logic;
  variable isum : std_logic_vector(x downto
    0);
  begin
    carry := cin;
    for i in 0 to x loop
      isum(i) := a(i) xor b(i) xor carry;
      carry := (a(i) and b(i)) or (a(i) and
        carry) or (b(i) and carry);
    end loop;
    sum <= isum;
    cout <= carry;
  end process adder;
end architecture behavior;
```

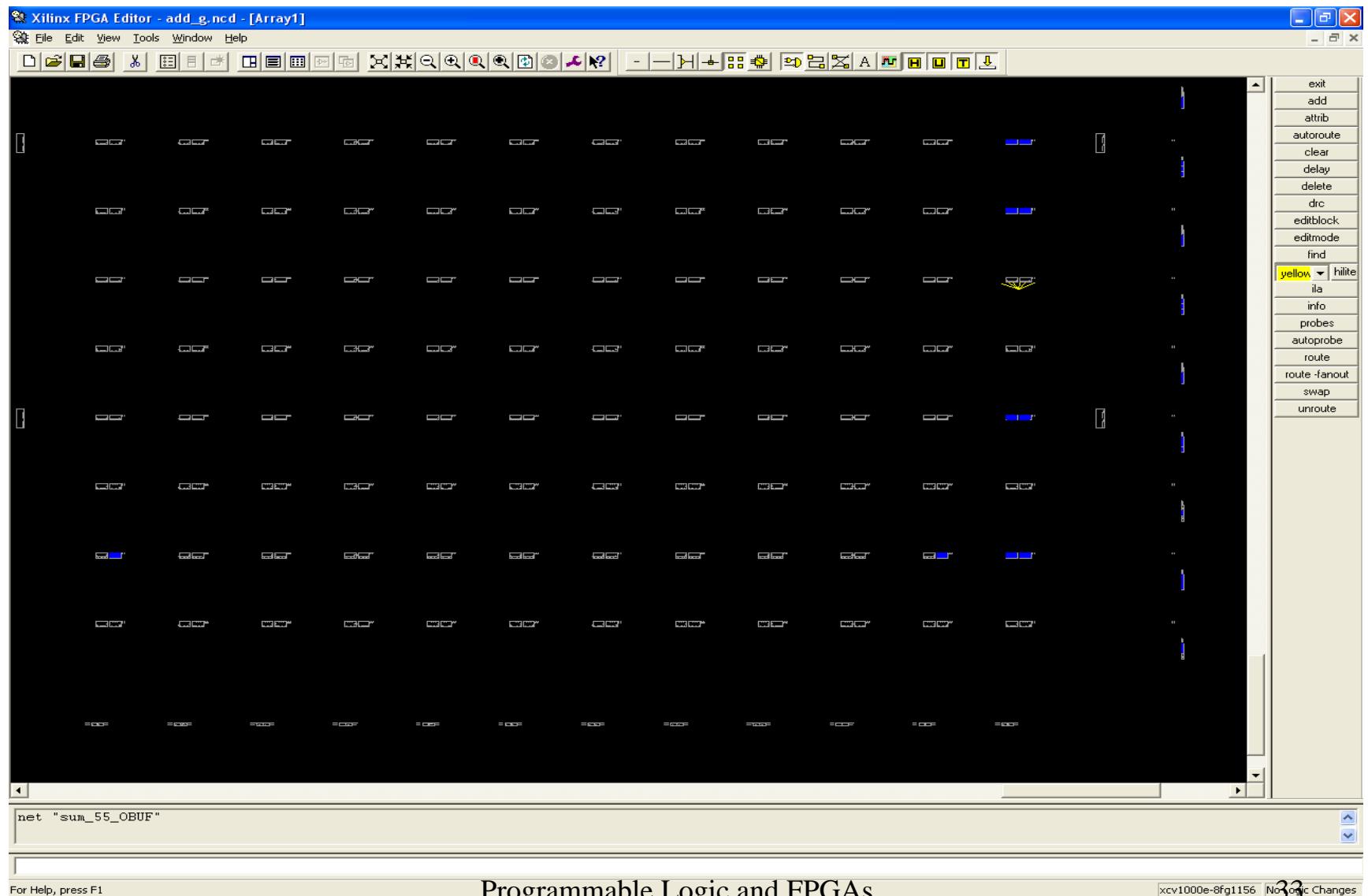
Synthesis



# Mapping the Design

- Mapping : Process of assigning portions of the logic design to the physical chip resources (CLBs).
- Function similar to synthesizing.
- Synthesis is not necessarily specific to a particular FPGA, but mapping usually is.

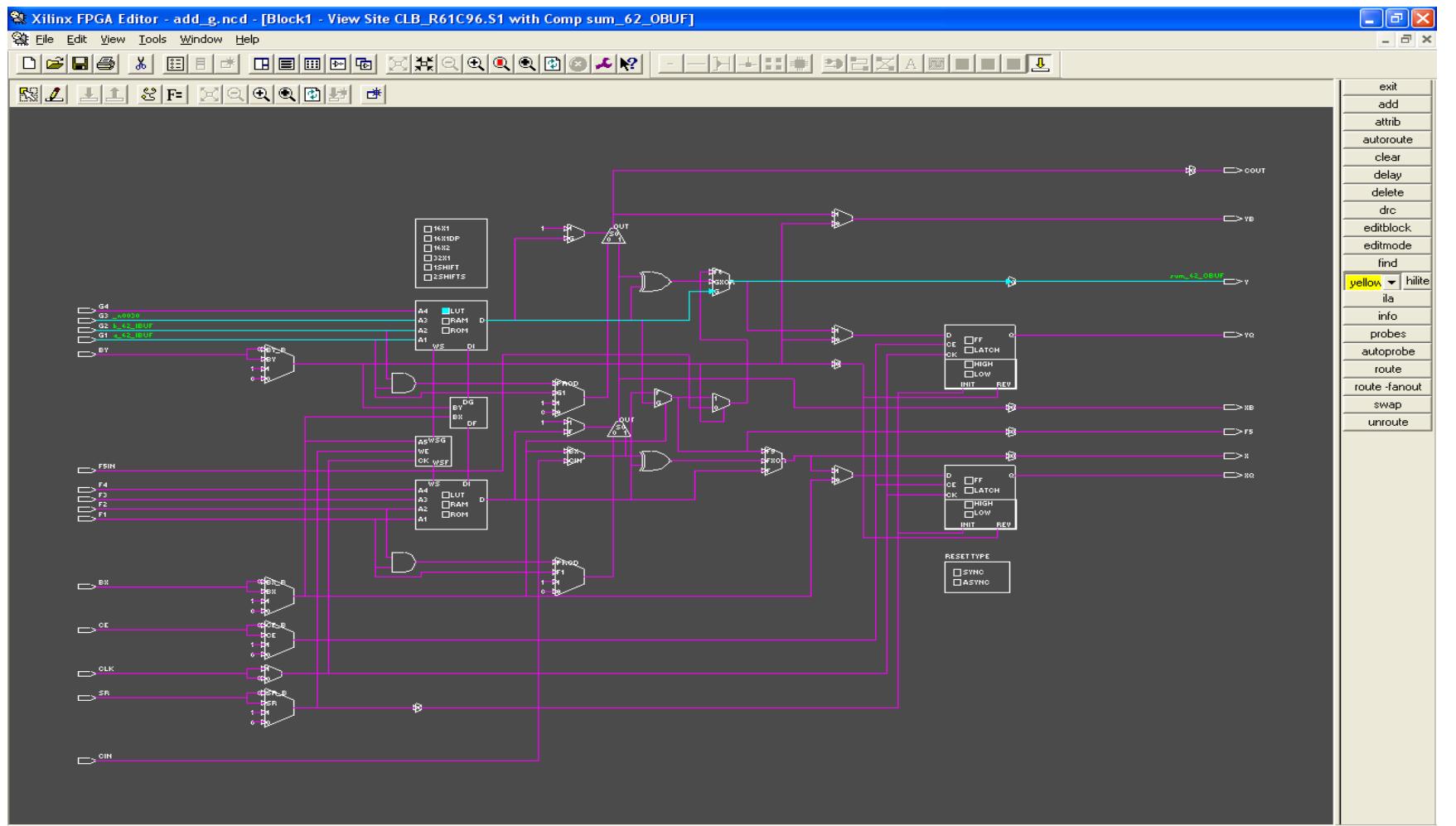
# Mapping the Design (contd.)



# Placing the Design

- Placing : In FPGAs, the process of assigning specific parts of the design to specific locations (CLBs) on the chip.
- Usually done automatically.
- Once the design has been converted into the logic resources of the FPGA, we find a location for these within the FPGA.  
(Generally a 2-dimensional grid structure)

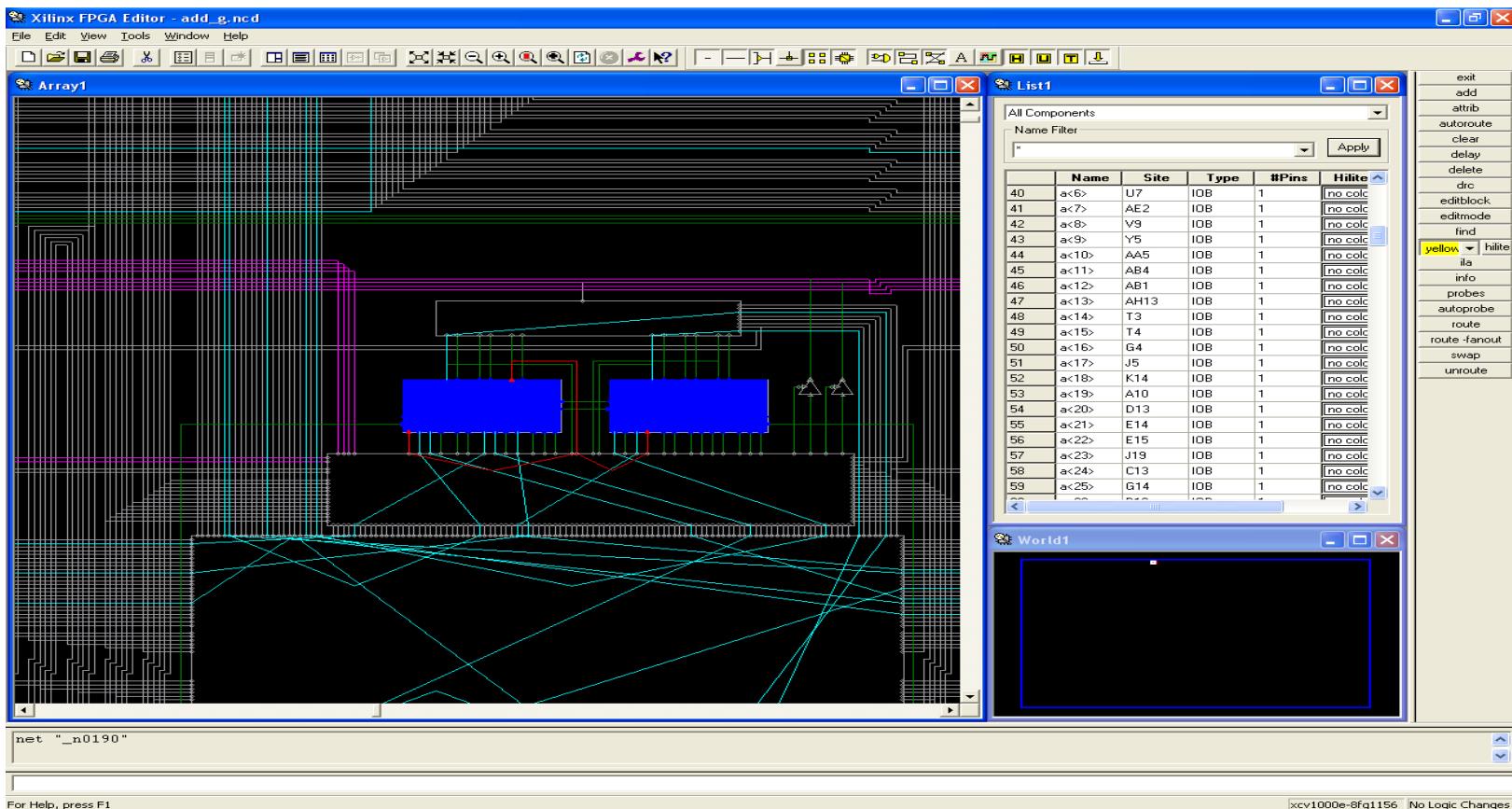
# Placing the Design (contd.)



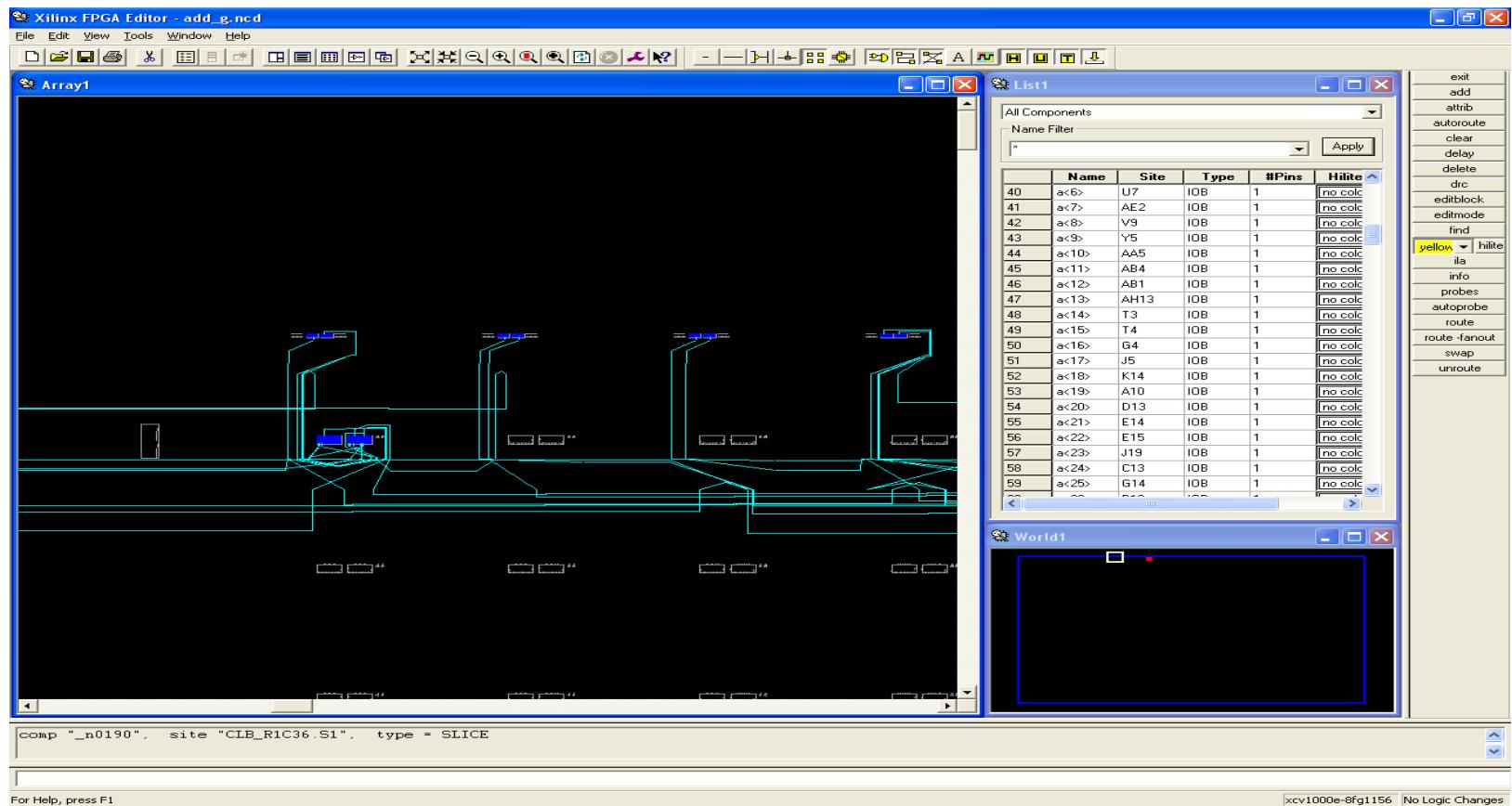
# Routing the Design

- Routing : The process of creating the desired interconnection of logic cells to make them perform the desired function.
  - Routing follows after placement.
  - Once logic resources have been assigned a location within the FPGA, we need to interconnect the logic resources using internal buses inside the FPGA.

# Routing the Design (contd.)



# Routing the Design (contd.)

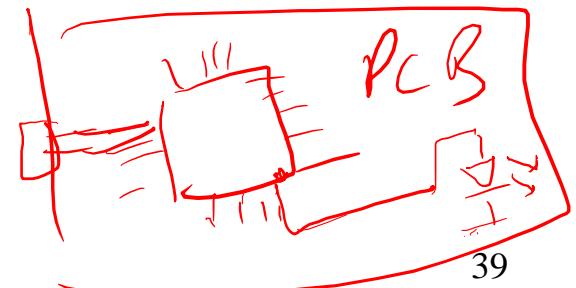
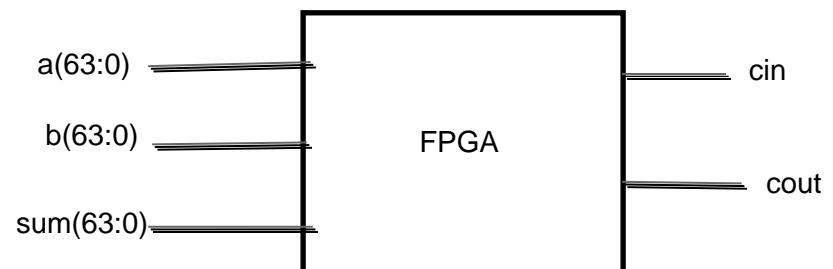


VLF

# Assigning Pins

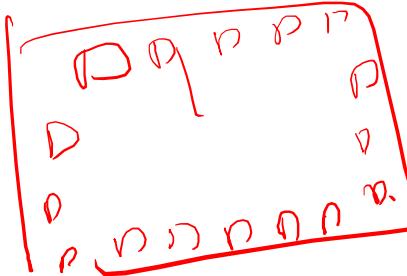
When implementing an entity in FPGA, the input and output ports are mapped to pins of the FPGA

```
entity add_g is
generic (x : natural := 63);
port (a : in std_logic_vector (x downto 0);
      b : in std_logic_vector (x downto 0);
      cin : in std_logic;
      sum : out std_logic_vector (x downto 0);
      cout : out std_logic);
end entity add_g;
```



# Assigning Pins (cont.)

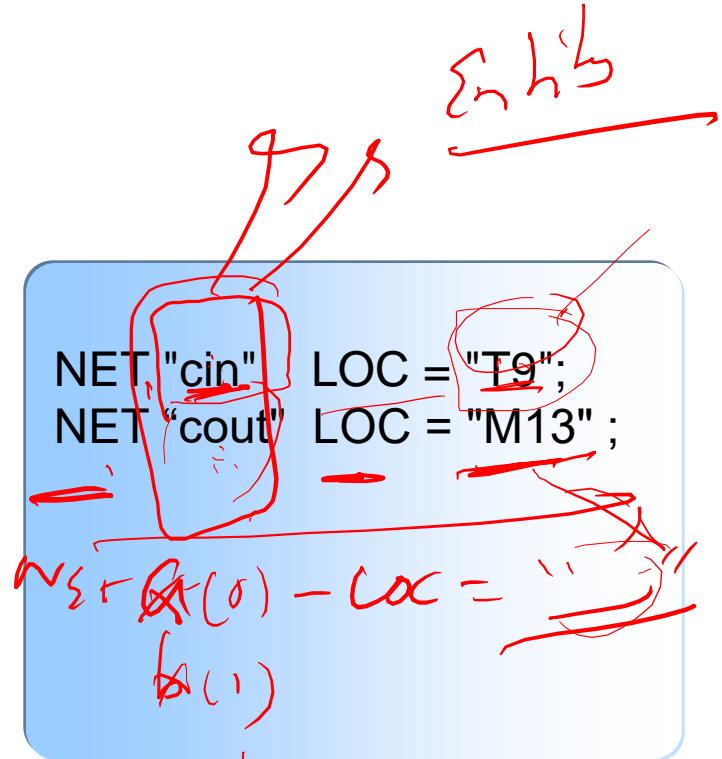
- A file called a UCF (User Constraint File) is used to define which pin will be connected to a particular input or output.
- Within Xilinx Project manager, the “assign package pin” function can be used to easily define input and output pin location.



# Example UC File

```
entity add_g is
generic (x : natural := 63);

port (a : in std_logic_vector (x downto 0);
b : in std_logic_vector (x downto 0);
cin : in std_logic;
sum : out std_logic_vector (x downto 0);
cout : out std_logic);
end entity add_g;
```



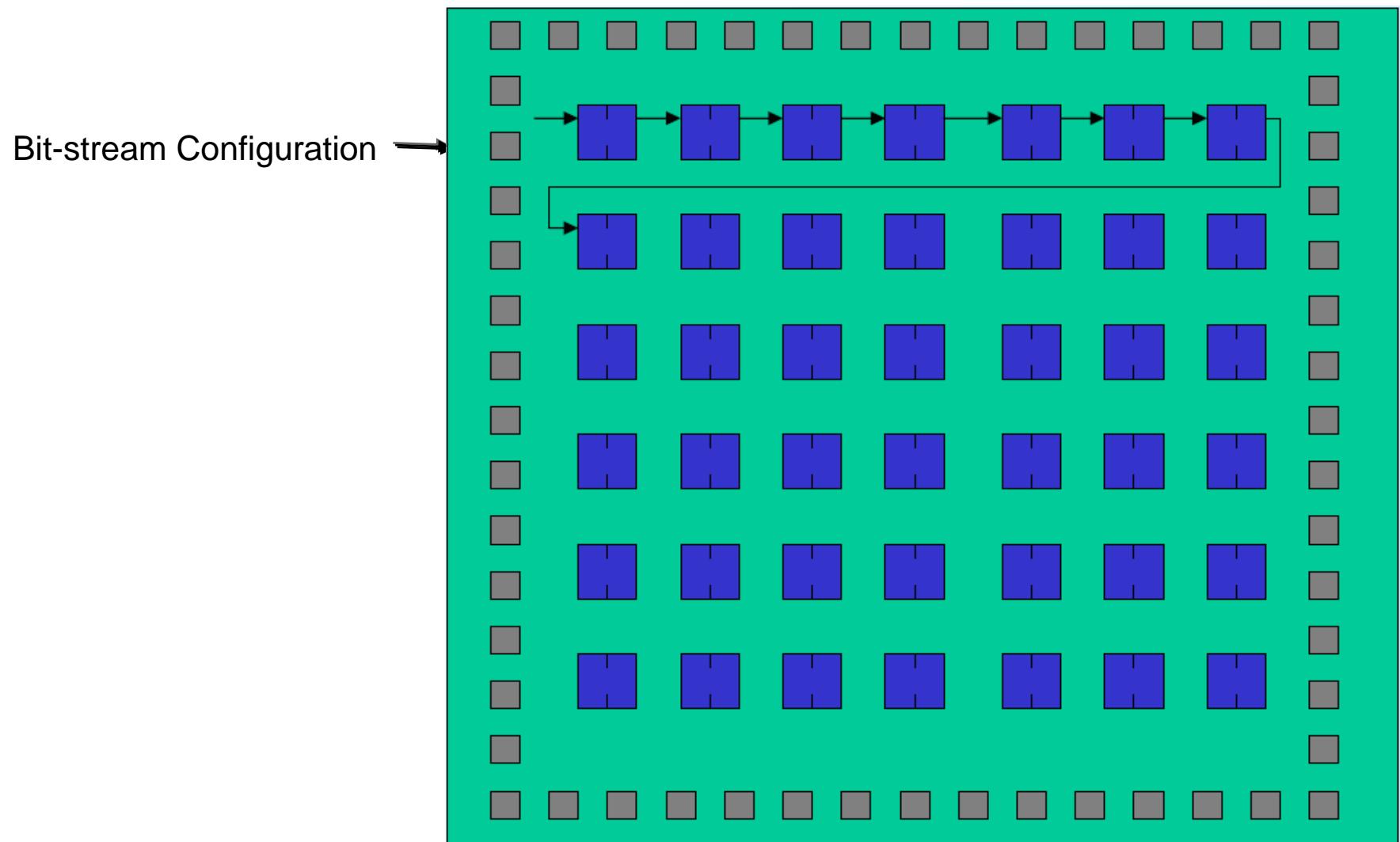
# Convert Design into Bit-stream

100K

- Always done using automated tools.
- The placed and routed design is converted into a bit-stream that is downloaded into the FPGA to configure it.

Slow tools

# Design to Bit-stream (contd.)



# Design Summary

Xilinx - ISE - D:\Gunjan\Xilinx8.2\adder\_64\adder\_64.ise - [Design Summary]

File Edit View Project Source Process Window Help

Sources X

Sources for: Synthesis\adder\_64

- adder\_64
- xcv1000e-8fg1156
- add\_g - beha

Processes X

Processes:

- View Symb
- View RTL
- View Tec
- Check Sy
- Generate
- Implement De
- Translate
- Map
- Place & R
  - Place
  - Clock
  - Asym
  - Pad F
  - Guide
- MPPF
- Gene
- View
- View
- Analy
- Gene
- Gene
- Gene
- Back
- Generate Proj
- Program
- Generate
- Configure
- Update Bitstr

FPGA Design Summary

Project File: adder\_64.ise Current State: Placed and Routed

Module Name: add\_g • Errors: No Errors

Target Device: xcv1000e-8fg1156 • Warnings: 1 Warning

Product Version: ISE, 8.1i • Updated: Thu Feb 12 07:26:10 2009

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	129	24,576	1%	
Logic Distribution				
Number of occupied Slices	96	12,288	1%	
Number of Slices containing only related logic	96	96	100%	
Number of Slices containing unrelated logic	0	96	0%	
Total Number of 4 input LUTs	129	24,576	1%	
Number of bonded IOBs	194	660	29%	
Total equivalent gate count for design	774			
Additional JTAG gate count for IOBs	9,312			

Performance Summary

Final Timing Score:	0	Pinout Data:	Pinout Report
Routing Results:	All Signals Completely Routed	Clock Data:	Clock Report
Timing Constraints:	All Constraints		

Detailed Reports

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Wed Feb 11 23:42:32 2009	0	0	0
Translation Report	Current	Thu Feb 12 07:26:00 2009	0	0	0
Map Report	Current	Thu Feb 12 07:26:04 2009	0	0	2 Infos
Place and Route Report	Current	Thu Feb 12 07:26:07 2009	0	1 Warning	1 Info
Static Timing Report	Current	Thu Feb 12 07:26:10 2009	0	0	2 Infos
Bitgen Report					

Started : "Launching Technology Schematic Viewer for add\_g.ngc".

Console Errors Warnings Find in Files

Programmable Logic and FPGAs 44

Snapshot from Xilinx ISE

# Synthesis Report

The screenshot shows the Xilinx ISE Design Summary interface. The main window displays the 'FPGA Design Summary' table of contents and the 'Synthesis Options Summary' report. A red curly brace on the right side groups the 'Design Overview' and 'Errors and Warnings' sections.

**TABLE OF CONTENTS**

- 1) Synthesis Options Summary
- 2) HDL Compilation
- 3) HDL Analysis
- 4) HDL Synthesis
  - 4.1) HDL Synthesis Report
- 5) Advanced HDL Synthesis
  - 5.1) Advanced HDL Synthesis Report
- 6) Low Level Synthesis
- 7) Final Report
  - 7.1) Device utilization summary
  - 7.2) TIMING REPORT

**-----**

**Synthesis Options Summary**

**Source Parameters**

Input File Name	:	"add_g.prj"
Input Format	:	mixed
Ignore Synthesis Constraint File	:	NO

**Target Parameters**

Output File Name	:	"add_g"
Output Format	:	NGC
Target Device	:	xcv1000e-8-fg1156

**Source Options**

Top Module Name	:	add_g
Automatic FSM Extraction	:	YES
FSM Encoding Algorithm	:	Auto
FSM Style	:	lut
RAM Extraction	:	Yes
RAM Style	:	Auto
ROM Extraction	:	Yes
Mux Style	:	Auto
Decoder Extraction	:	YES
Priority Encoder Extraction	:	YES
Shift Register Extraction	:	YES
Logical Shifter Extraction	:	YES
XOR Collapsing	:	YES
ROM Style	:	Auto
Mux Extraction	:	YES
Resource Sharing	:	YES
Multiplier Style	:	lut
Automatic Register Balancing	:	No

**Design Summary**

Started : "Launching Design Summary".

Console Errors Warnings Find in Files

# Synthesis Report – Device Utilization Summary

The screenshot shows the Xilinx ISE Design Suite interface with the title bar "Xilinx - ISE - D:\Gunjan\Xilinx8.2\adder\_64\adder\_64.ise - [Design Summary]". The left sidebar contains "Sources" and "Processes" sections. The "Processes" section is expanded, showing a tree view with "Synthesizer" selected, and its sub-options like "View Syn", "View RTL", "View Tec", etc. A red arrow points from the "Synthesis Report" option under "Detailed Reports" in the "Processes" sidebar towards the synthesis report text in the main window.

**FPGA Design Summary**

Mapping all equations...  
Building and optimizing final netlist ...  
Found area constraint ratio of 100 (+ 5) on block add\_g, actual ratio is 0.

=====

\* Final Report \*

=====

**Final Results**

RTL Top Level Output File Name	:	add_g.ngr
Top Level Output File Name	:	add_g
Output Format	:	NGC
Optimization Goal	:	Speed
Keep Hierarchy	:	NO

**Design Statistics**

# IOs	:	194
-------	---	-----

**Cell Usage :**

# BELs	:	129
# LUT2	:	1
# LUT3	:	127
# LUT4	:	1
# IO Buffers	:	194
# IBUF	:	129
# OBUF	:	65

**Device utilization summary:**

Selected Device : v1000efg116-8

Number of Slices:	74	out of	12288	0%
Number of 4 input LUTs:	129	out of	24576	0%
Number of bonded IOBs:	194	out of	664	29%

**TIMING REPORT**

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.  
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT  
GENERATED AFTER PLACE-and-ROUTE.

**Clock Information:**

Started : "Launching Technology Schematic Viewer for add\_g.ngc".

Transcript

Console Errors Warnings Find in Files

# Synthesis Report – Timing Report

Xilinx - ISE - D:\Gunjan\Xilinx8.2\adder\_64\adder\_64.ise - [Design Summary]

File Edit View Project Source Process Window Help

Sources Sources for: Synthesis/1

- adder\_64
- xcv1000e-0fg1156
- + add\_g - beha

Processes Processes:

- User Constraint
- Synthesis - X
- View Synl View RTL
- View Tec
- Check Sy
- Generate
- Implement De
- Translate
- Map
- Place & R
- Place
- Clock
- Async
- OPadF
- Guide
- MPPF
- Gene
- View
- View
- Analy
- Gene
- Gene
- Gene
- Back
- Generate Prog
- Programm
- Generate
- Configure
- Update Bitstre

FPGA Design Summary

### TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.  
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT  
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:  
-----  
No clock signals found in this design

Timing Summary:  
-----  
Speed Grade: -8

Minimum period: No path found  
Minimum input arrival time before clock: No path found  
Maximum output required time after clock: No path found  
Maximum combinational path delay: 90.059ns

Timing Detail:  
-----  
All values displayed in nanoseconds (ns)

Timing constraint: Default path analysis  
Total number of paths / destination ports: 4353 / 65

Delay: 90.059ns (Levels of logic = 66)  
Source: cin (PAD)  
Destination: cout (PAD)

Data Path: cin to cout

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
IBUF:I->O	3	0.744	1.056	cin_IBUF (cin_IBUF)
LUT3:IO->O	1	0.398	0.736	_n0000_SW0 (N188)
LUT3:I2->O	2	0.398	0.920	_n0000_( n0000)
LUT3:I2->O	2	0.398	0.920	_n02251 (_n0225)
LUT3:I2->O	2	0.398	0.920	_n00011 (_n0001)
LUT3:I2->O	2	0.398	0.920	_n02371 (_n0237)
LUT3:I2->O	2	0.398	0.920	_n00021 (_n0002)
LUT3:I2->O	2	0.398	0.920	_n02291 (_n0229)
LUT3:I2->O	2	0.398	0.920	_n00031 (_n0003)
LUT3:I2->O	2	0.398	0.920	_n02211 (_n0221)
LUT3:I2->O	2	0.398	0.920	_n00041 (_n0004)
LUT3:I2->O	2	0.398	0.920	_n000111 (_n00011)

Started : "Launching Technology Schematic Viewer for add\_g.ngc".

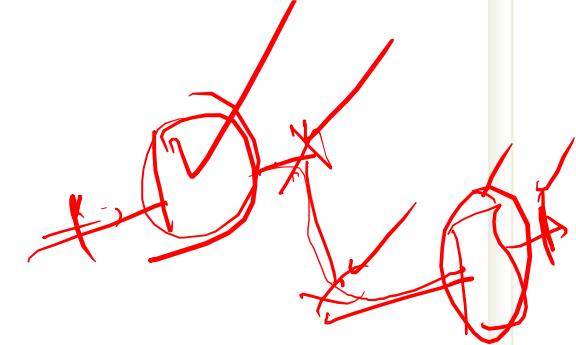
Transcript

Console Errors Warnings Find in Files

Programmable Logic and FPGAs

Snapshot from Xilinx ISE

47

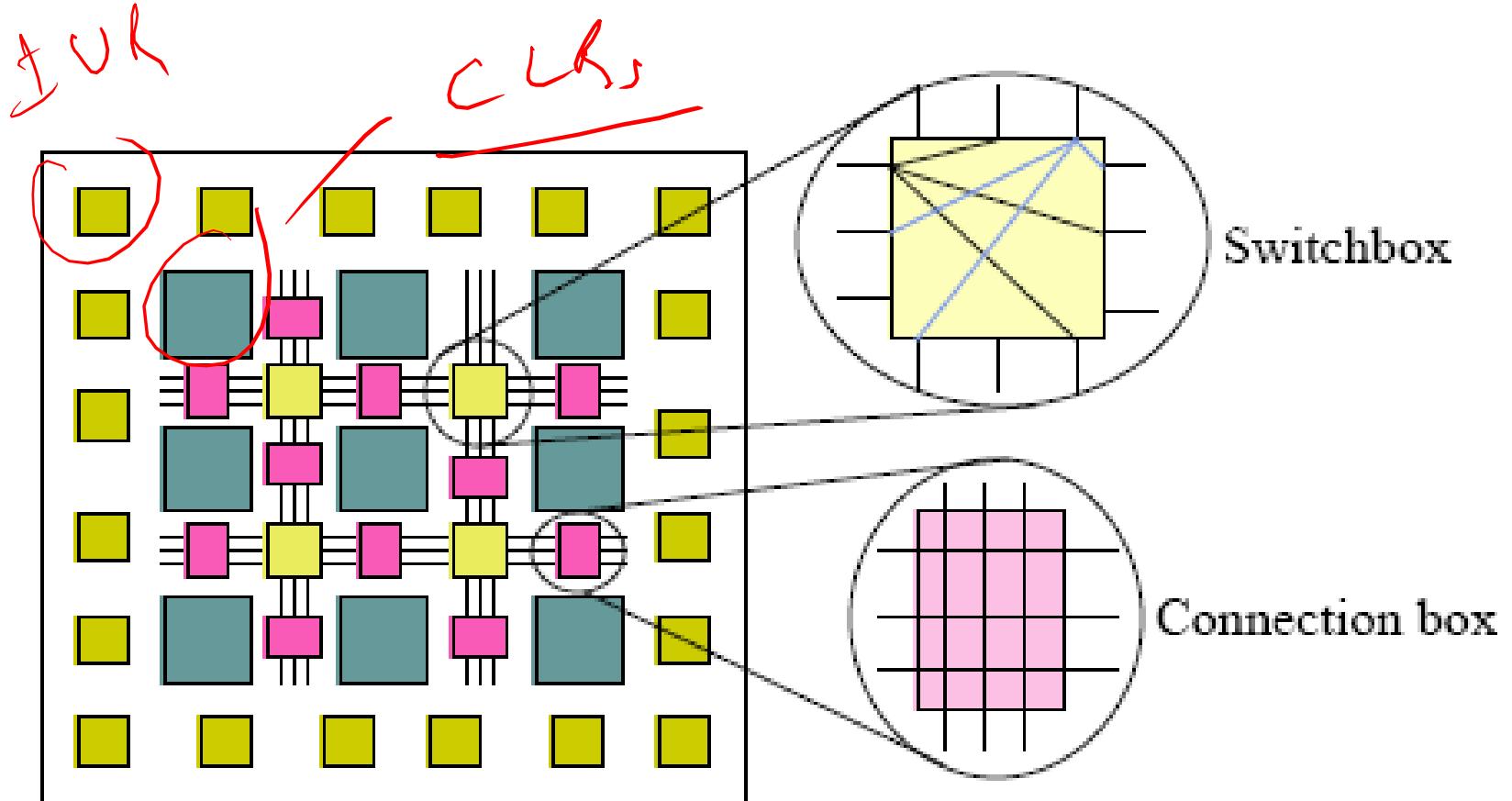




# Lecture 34: FPGA Architecture

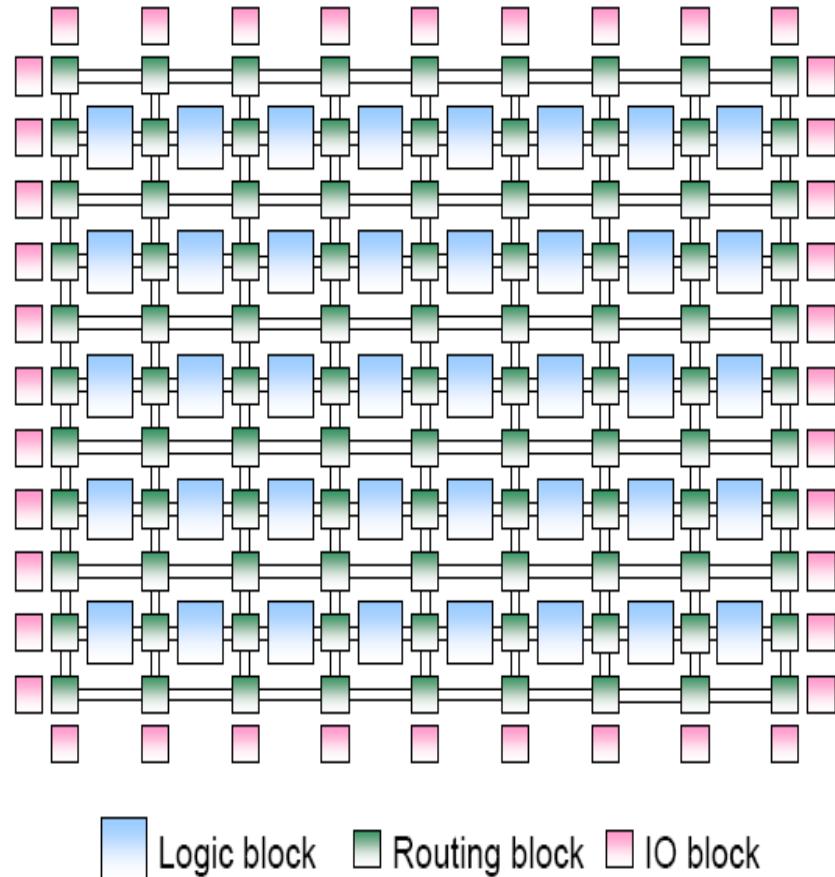
M. Balakrishnan

# Generic 2D FPGA

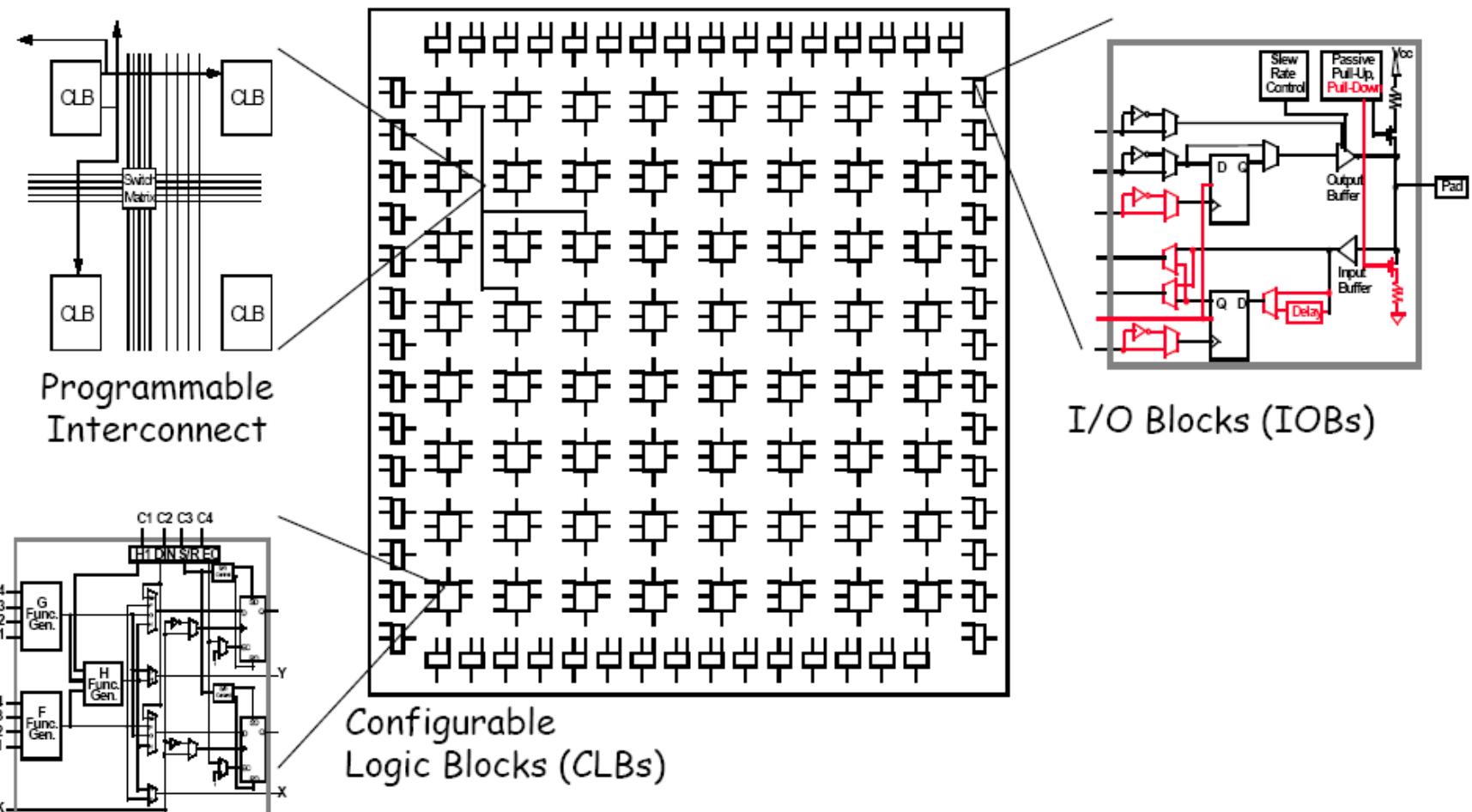


# FPGA Architecture

- An FPGA is an array of programmable logic elements that can be connected to inputs or outputs.

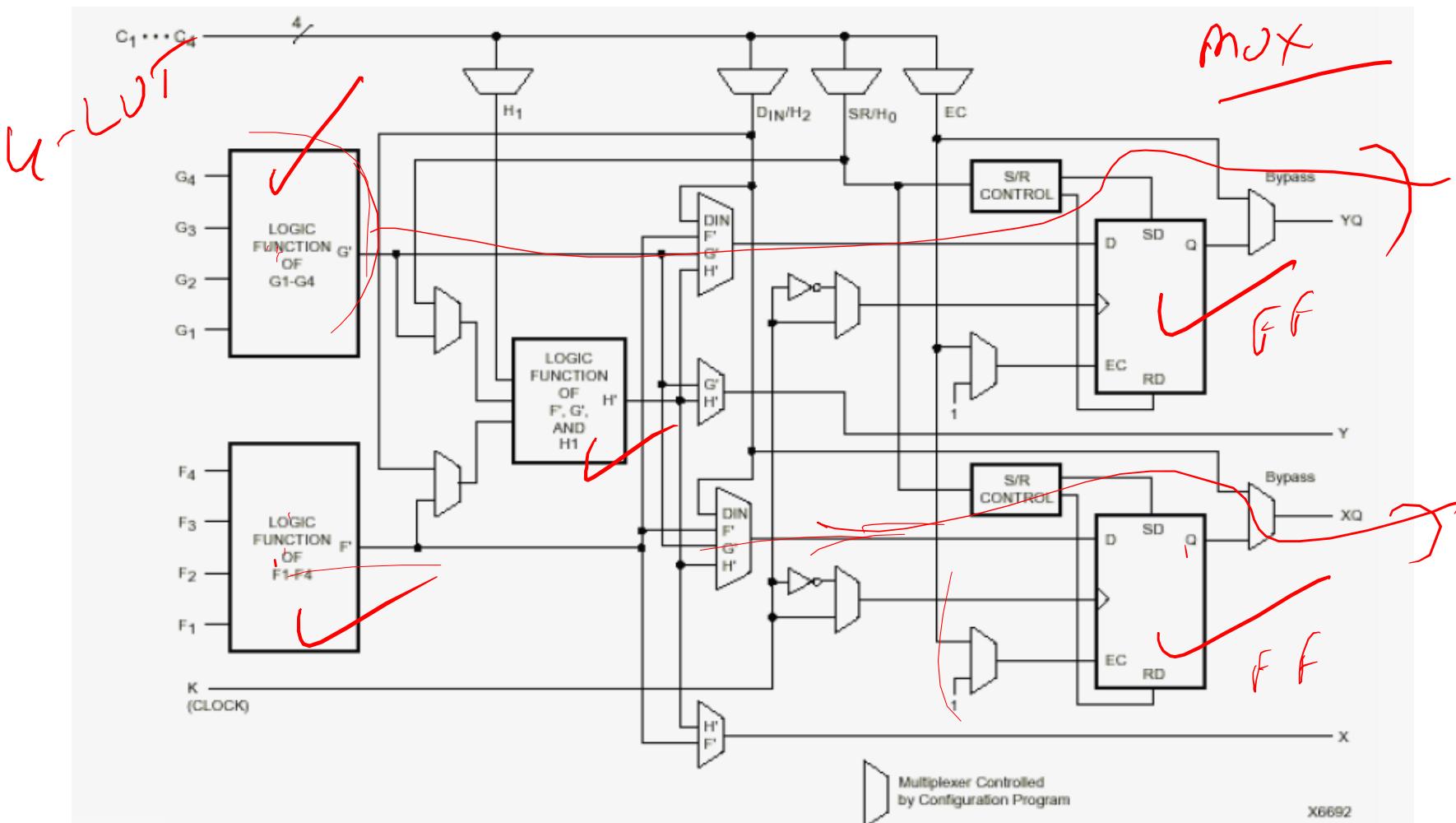


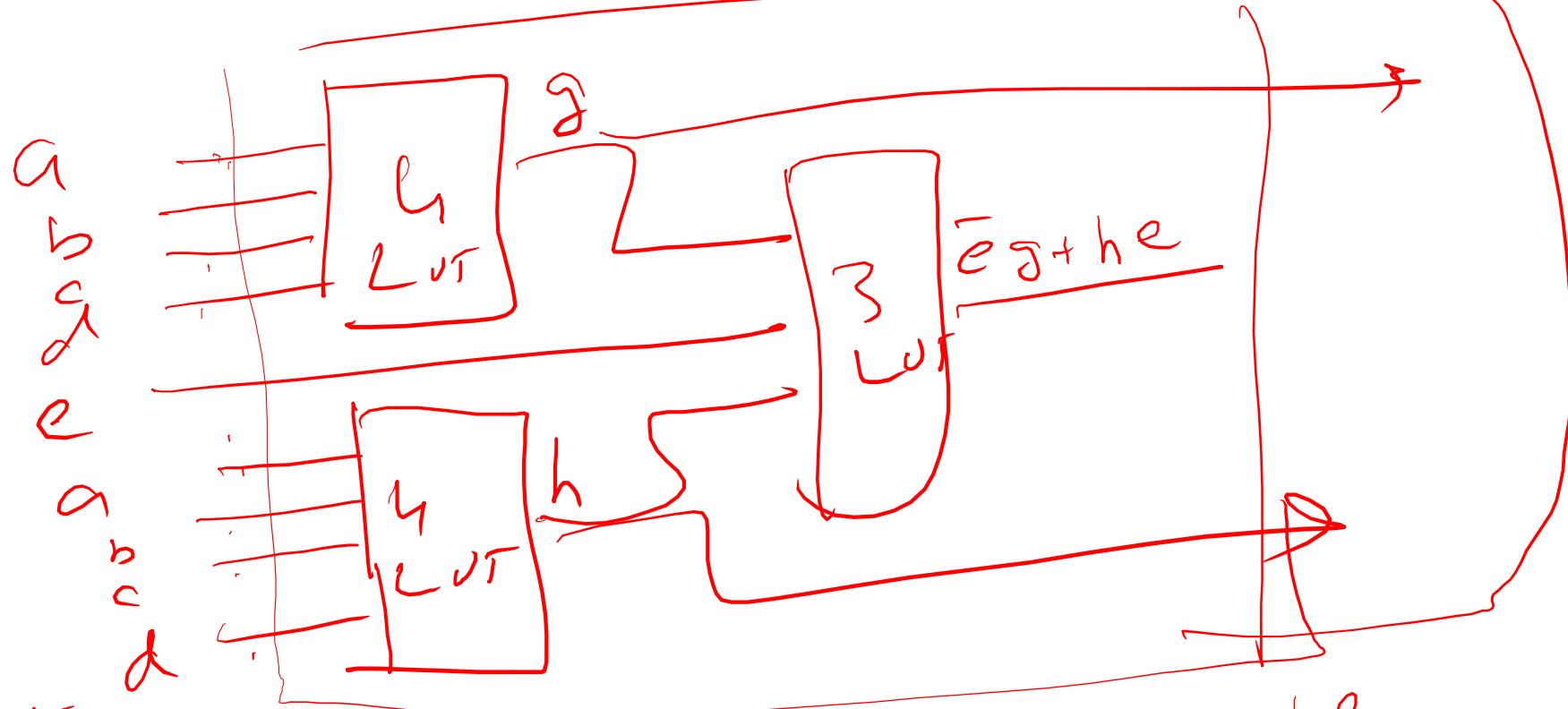
# SRAM Based FPGA - XILINX



# Xilinx 4000 CLB

Configurable





9 inputs

$$f(a, b, c, d)$$

9 variables

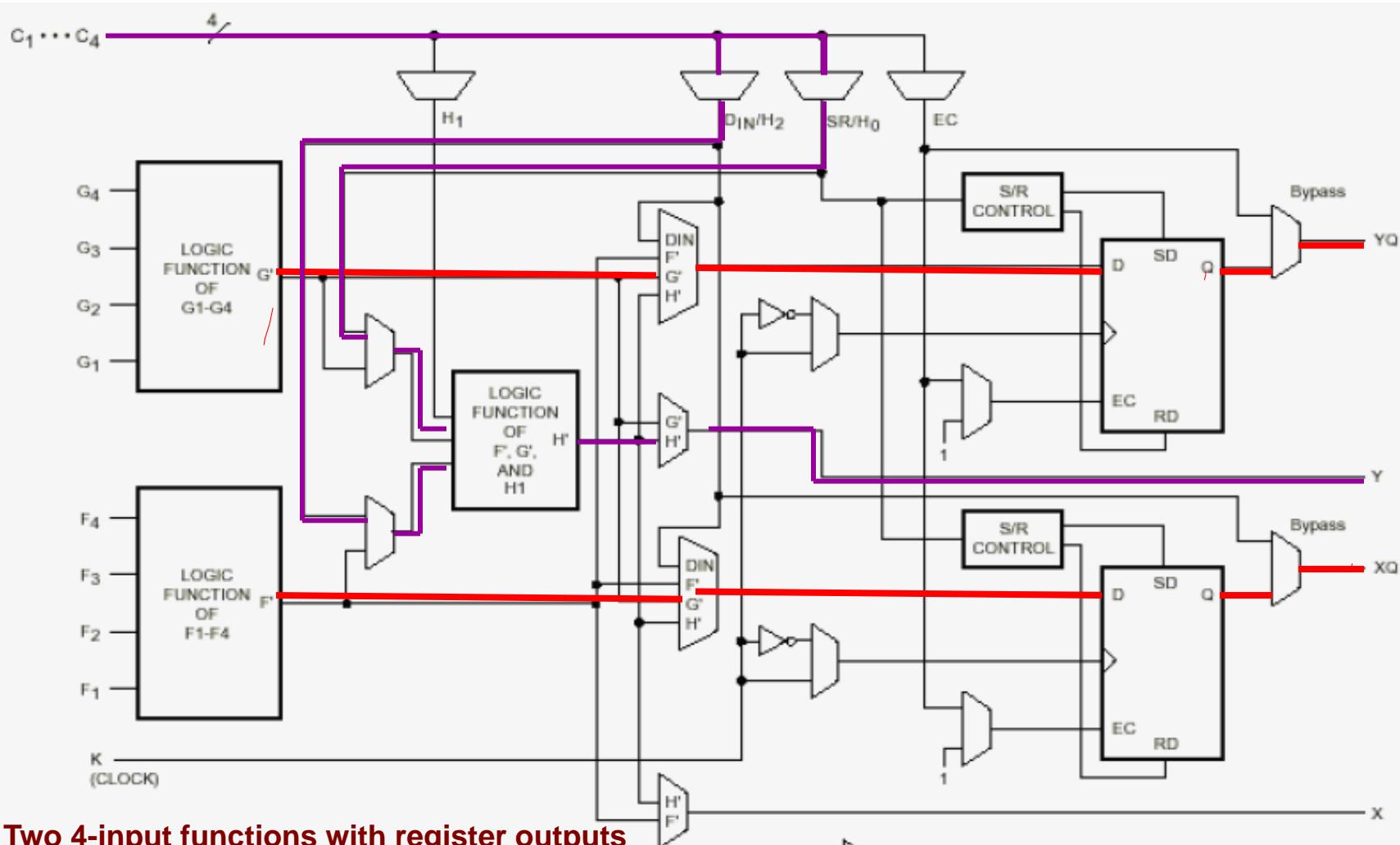
Answer NO

Programmable Logic and FPGAs

5 variables

$$f(a, b, c, d, e) = g(a, b, c, d) \bar{e} + h(a, b, c, d) e$$

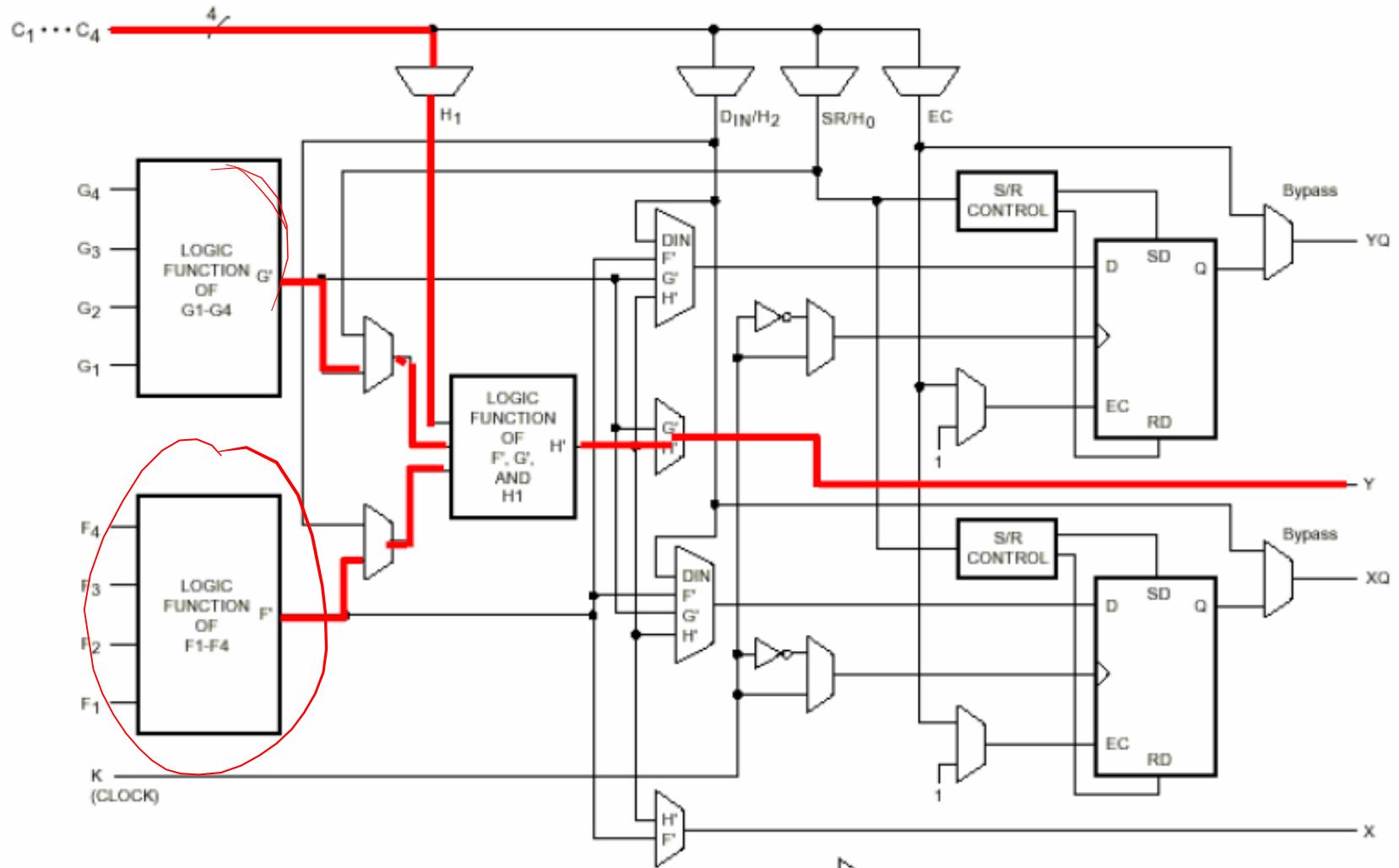
# Implementing Combinational Logic



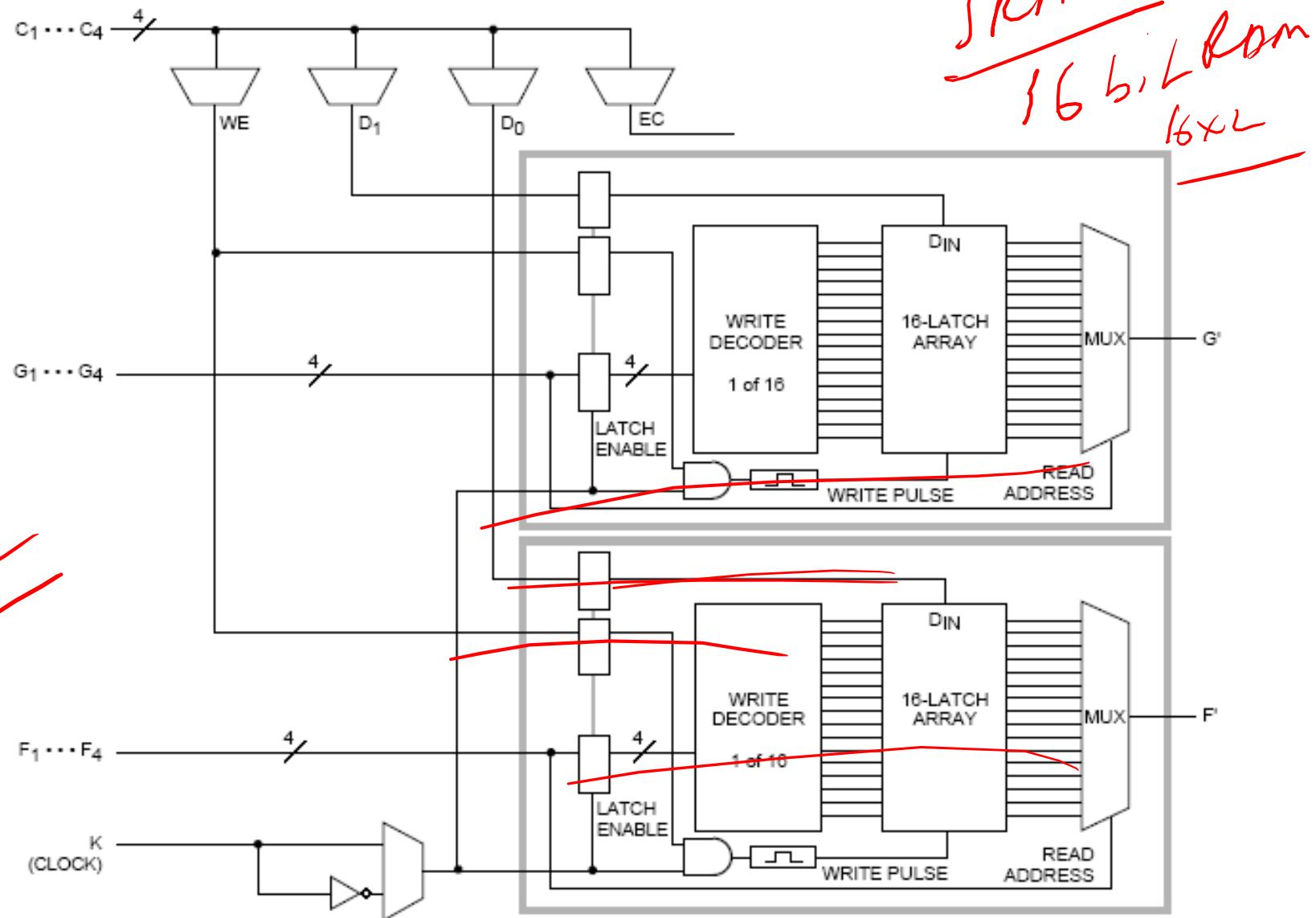
Two 4-input functions with register outputs  
and one 2-input function

Multiplexer Controlled  
by Configuration Program

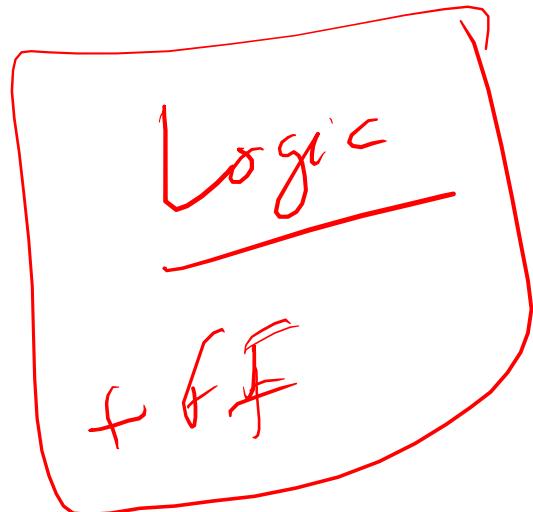
# 5 input Function



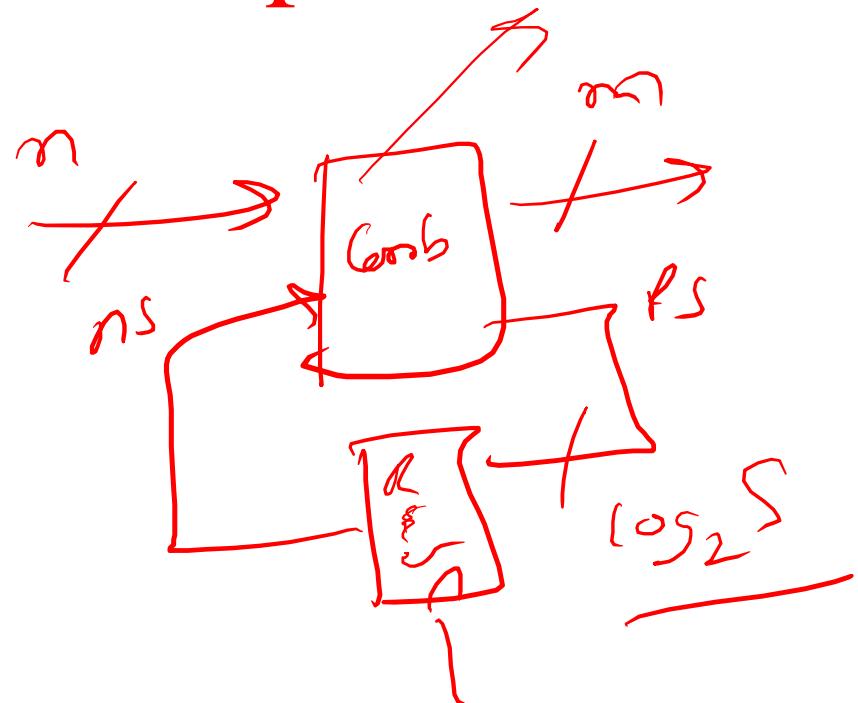
# Single Port RAM



# Function Decomposition

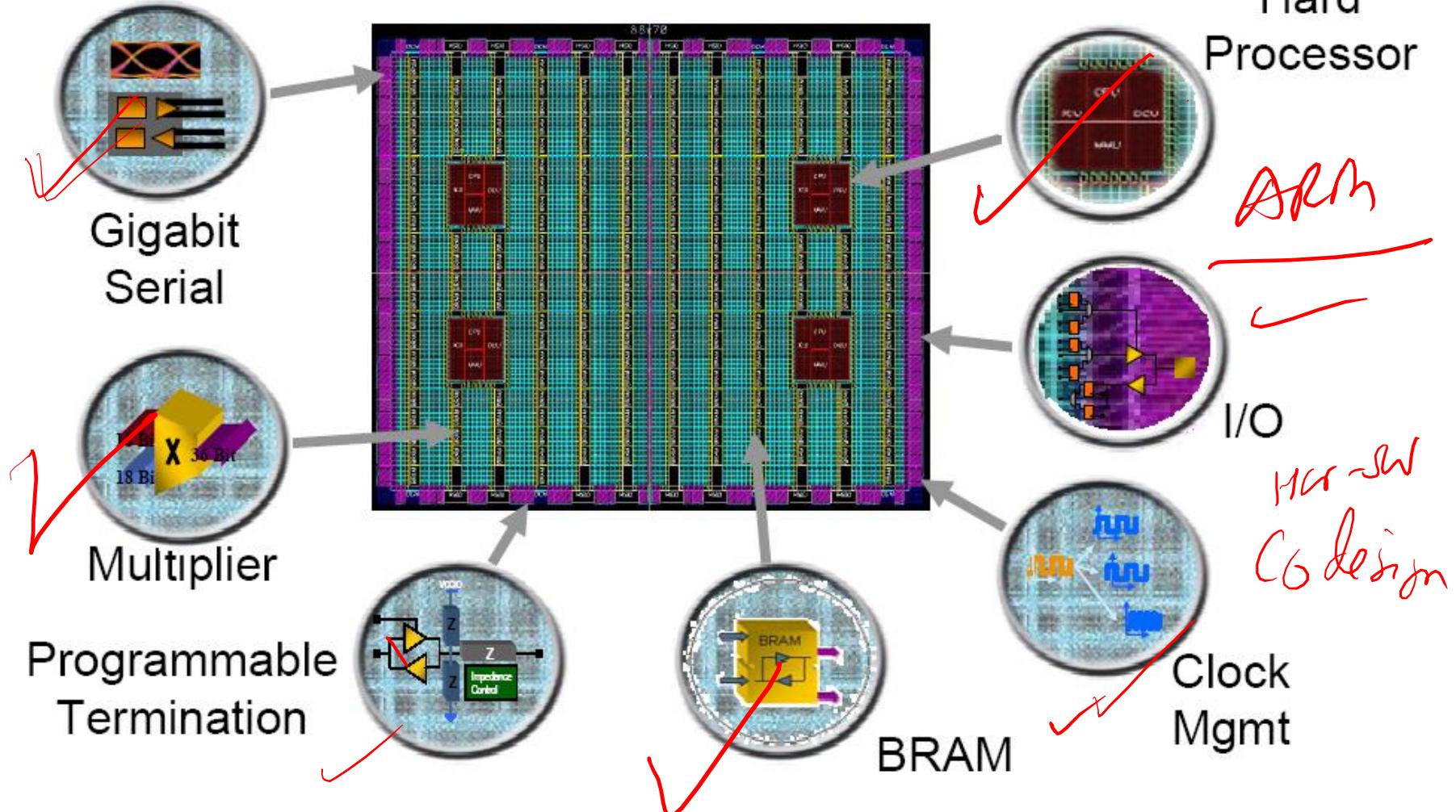


8 shk bits  
YCLBs



X C4000

# Platform Computing



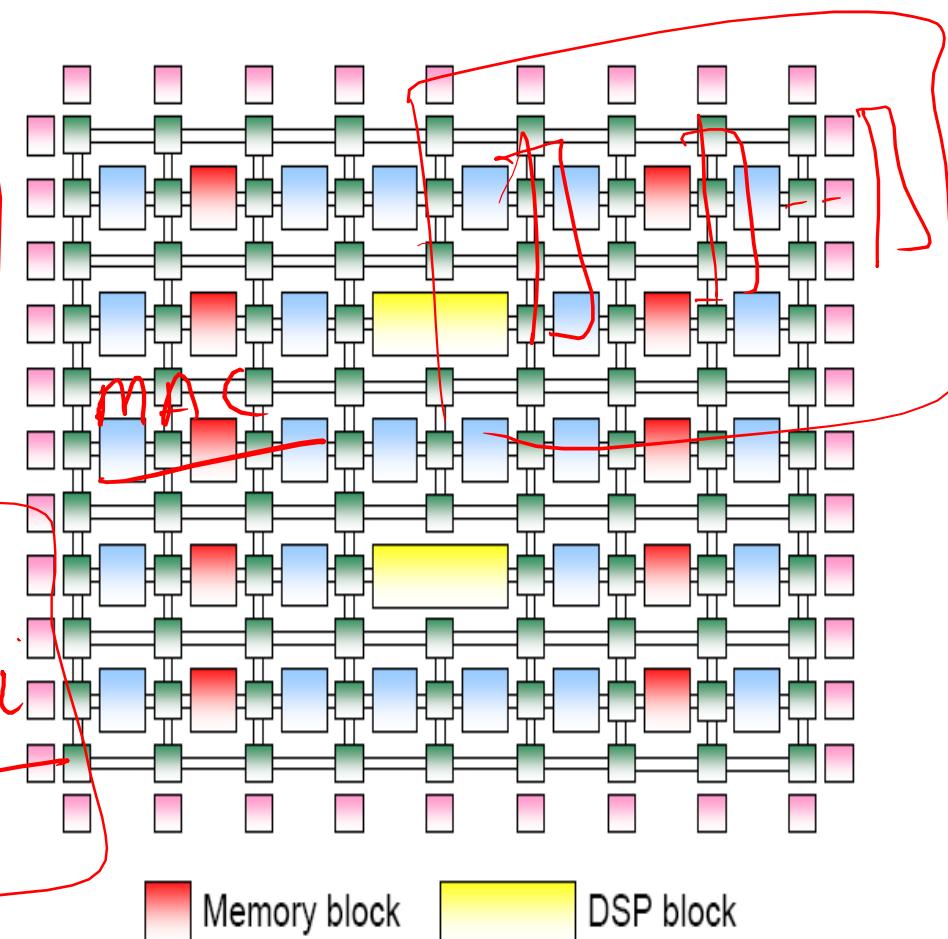
Courtesy of David B. Parlour, ISSCC 2004 Tutorial,  
“The Reality and Promise of Reconfigurable Computing in Digital Signal Processing”

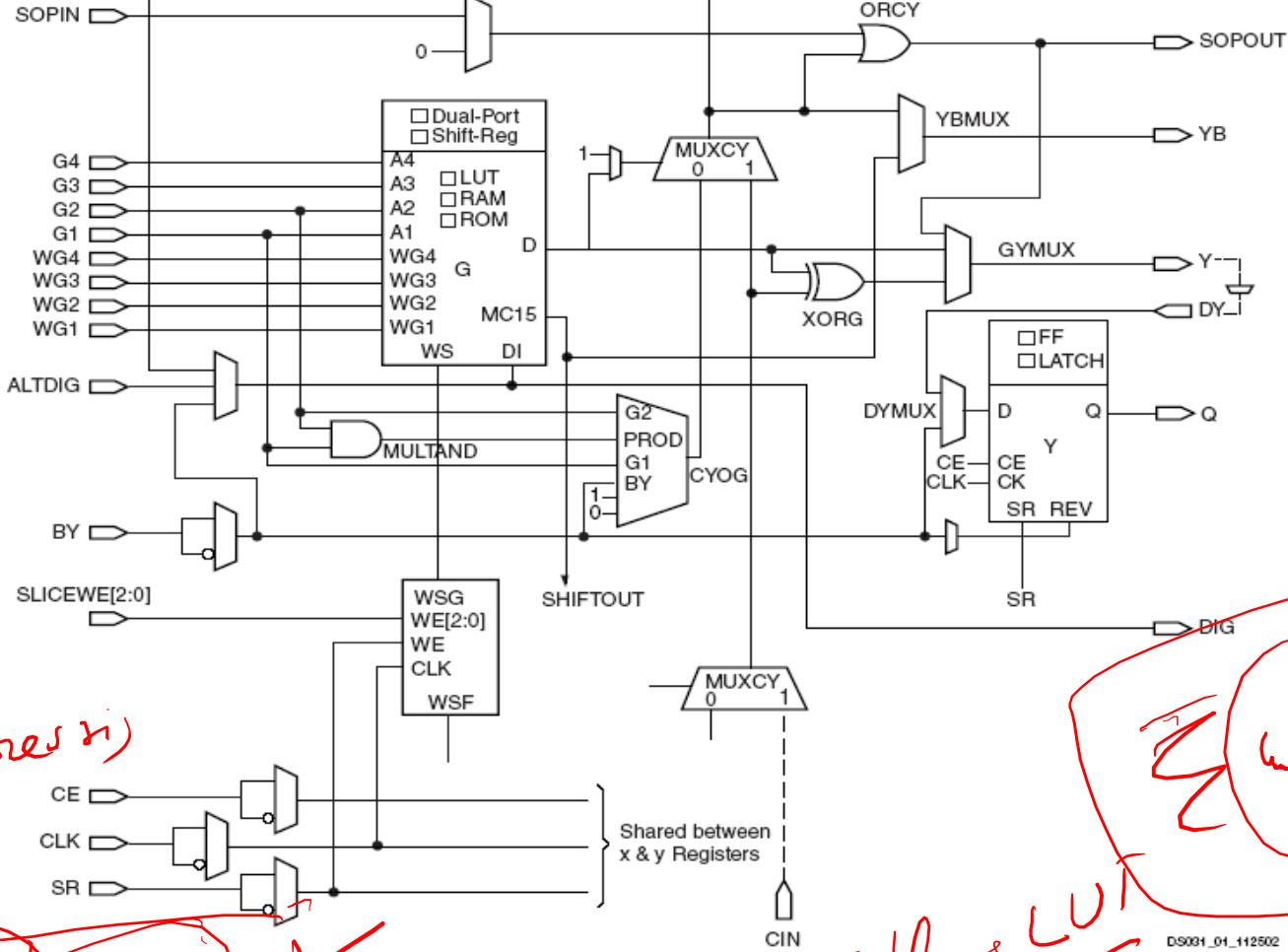
# Modern FPGA Fabric

Programmable fabric

- + Memory blocks
- DSP blocks

$$\sum_{i=1}^n \omega_i x_i$$

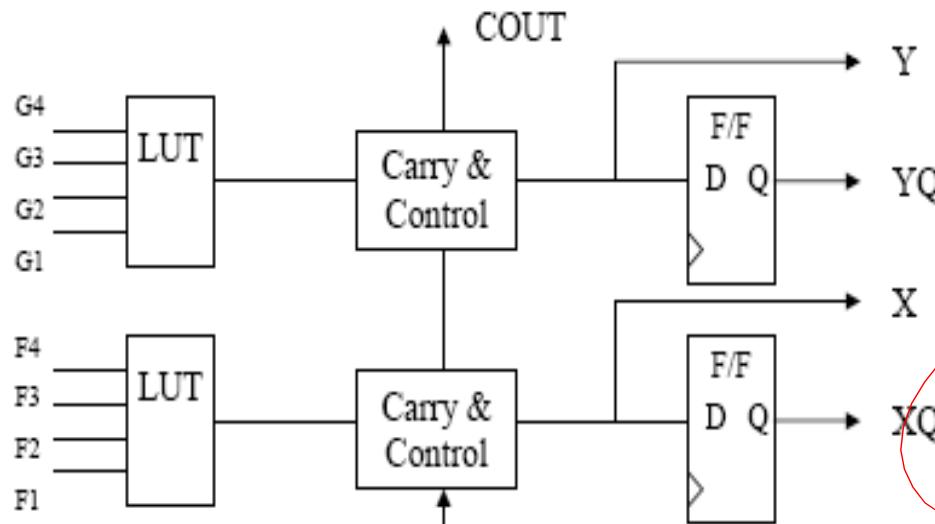




Truth Table  $\rightarrow$  LUT  
Configuration bit file

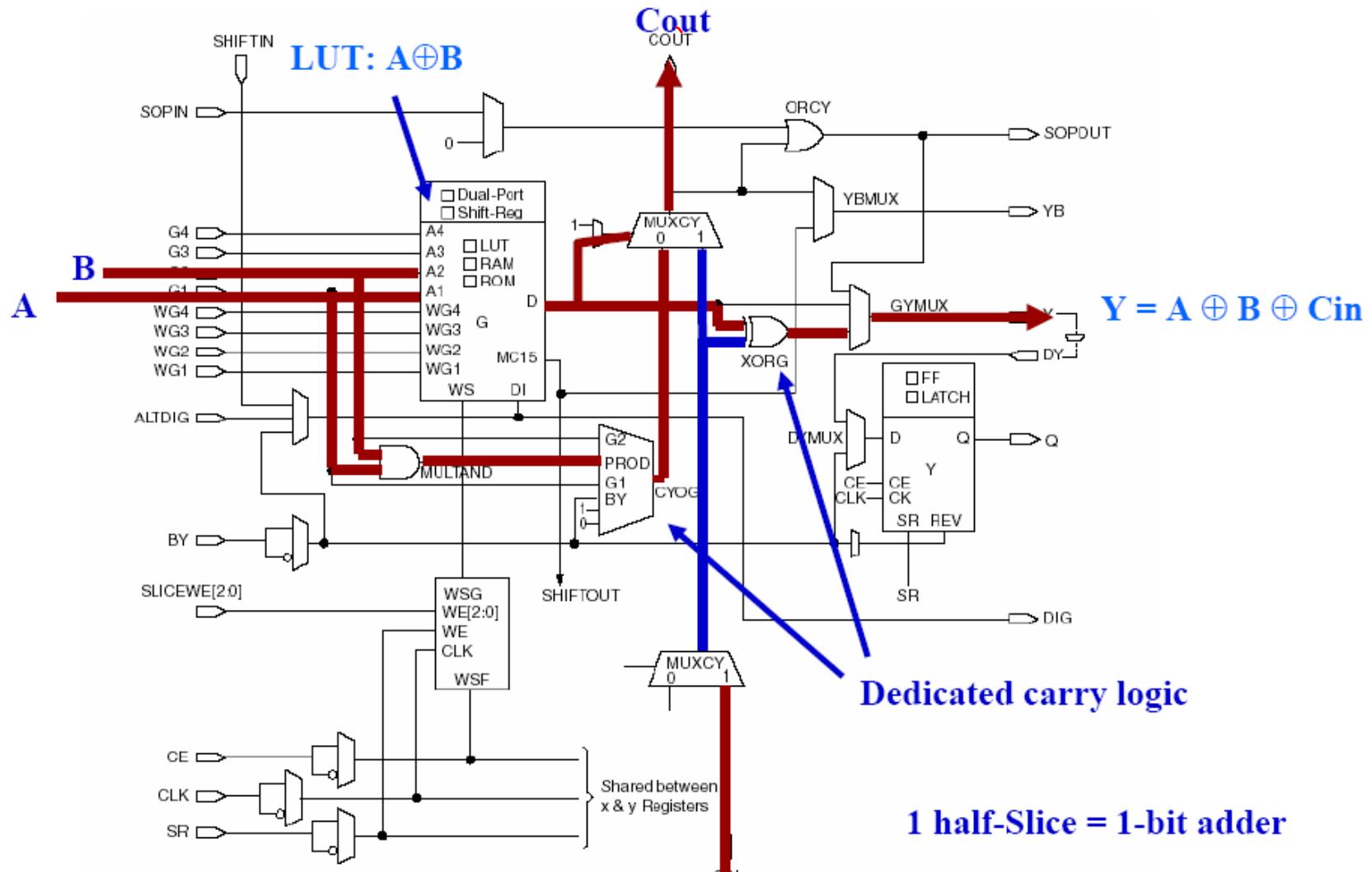
# FPGA Performance Issues

- Flexible logic – unused logic and additional delays
- Interconnects through switches – additional delays

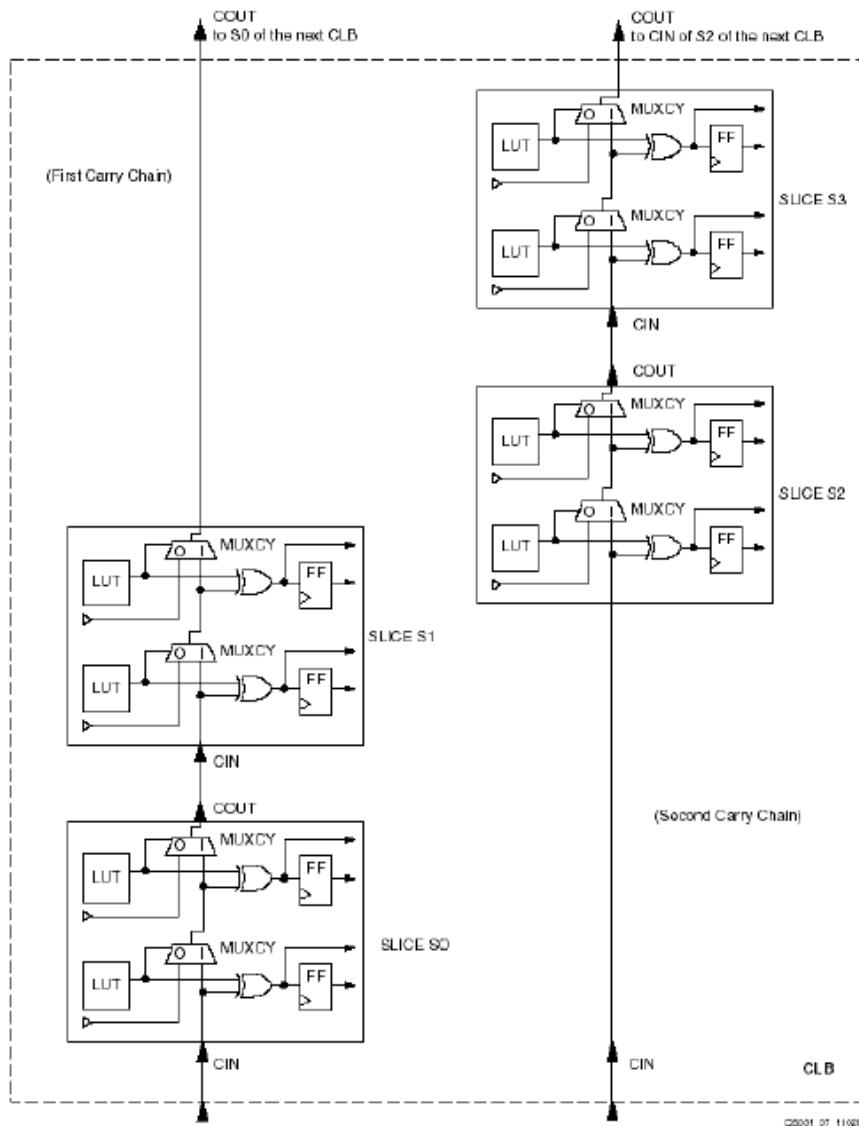


Source: xilinx.com

# Adder Implementation



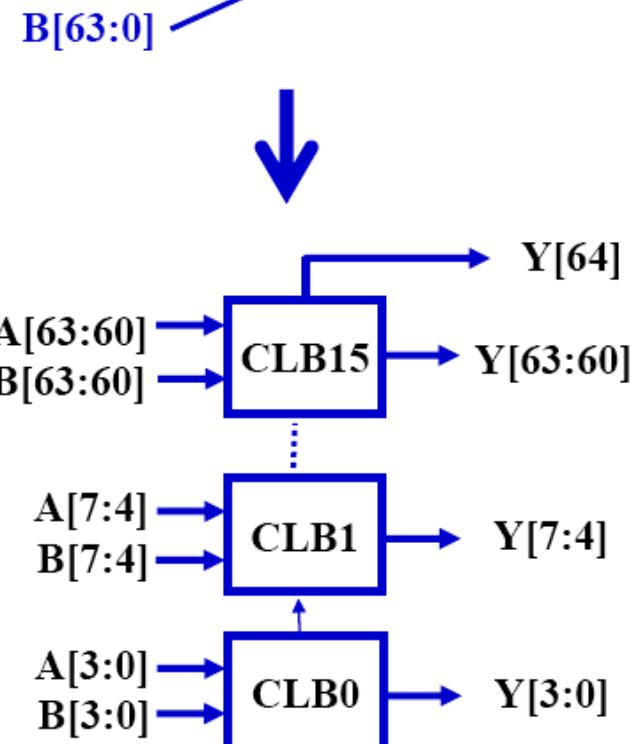
# Carry Chain Implementation



1 CLB = 4 Slices = 2, 4-bit adders

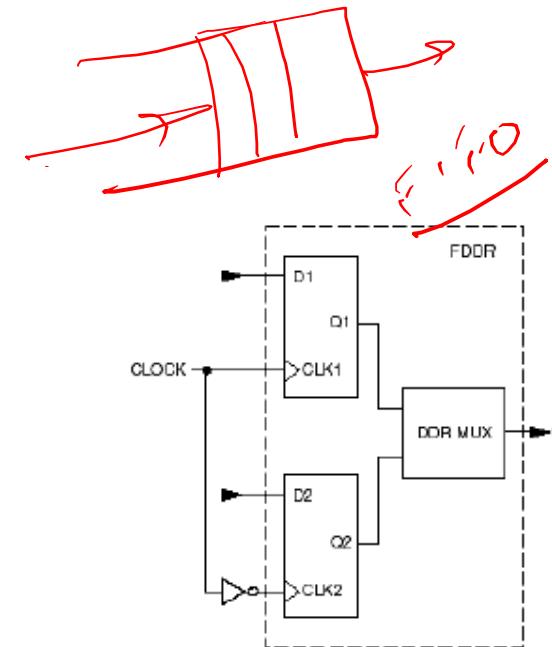
64-bit Adder: 16 CLBs

$$A[63:0] + B[63:0] \rightarrow Y[63:0]$$

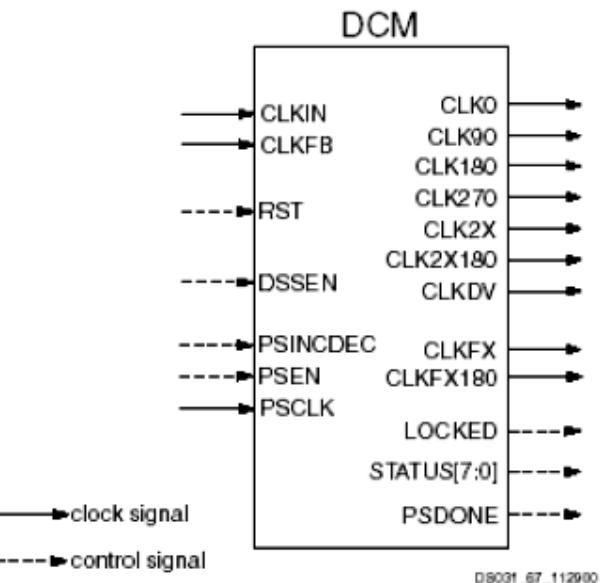


CLBs must be in same column

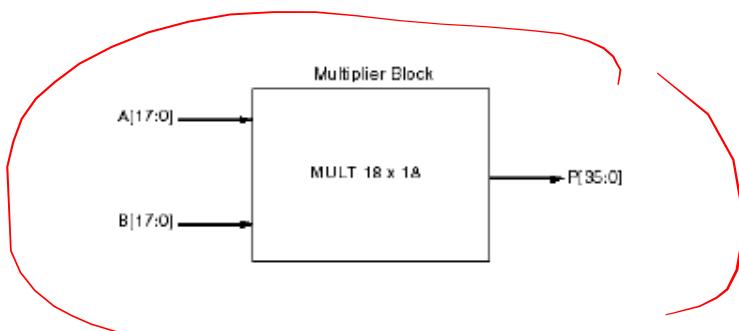
# Other Features



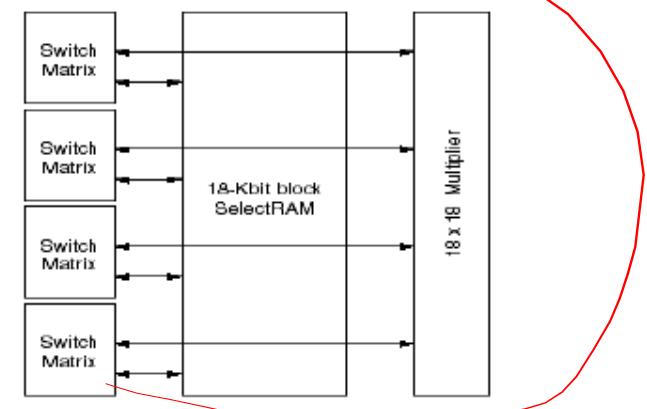
Double Data Rate registers



Digital Clock Manager

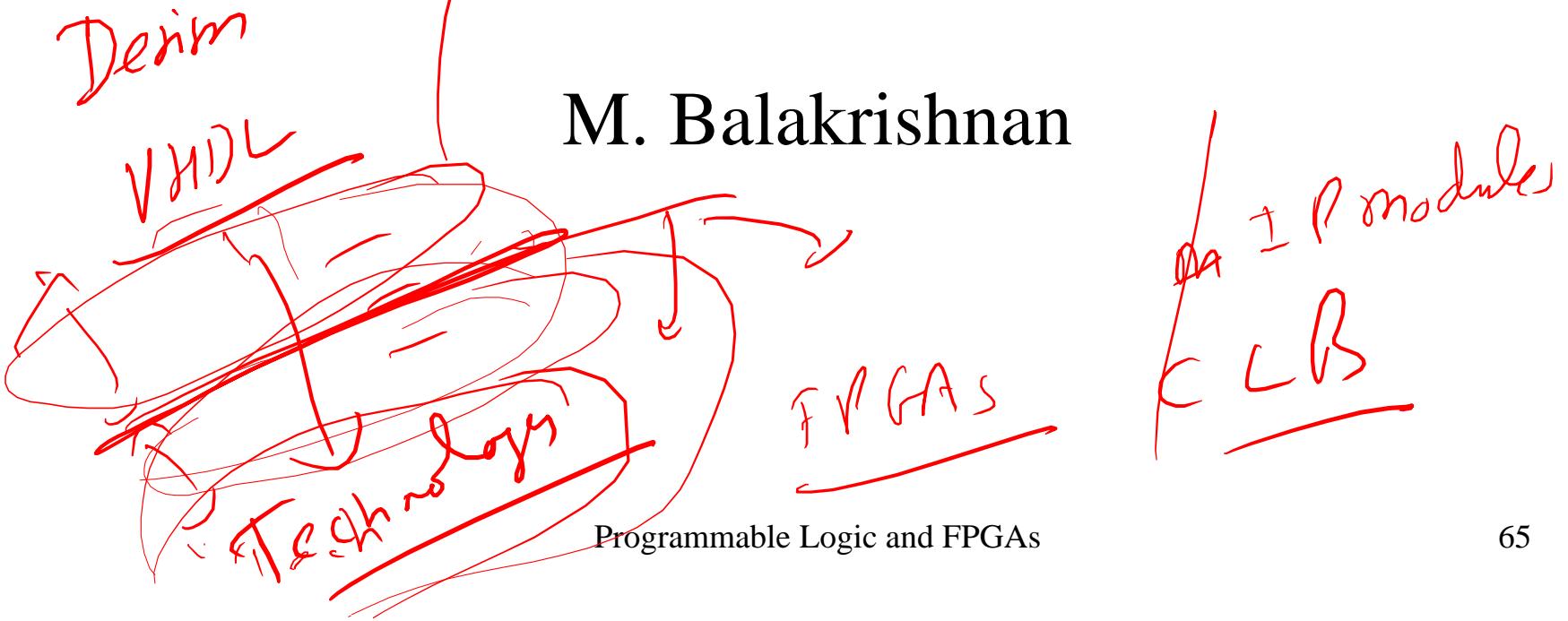


Embedded Multiplier



Block SelectRAM

# Lecture 35: FPGA Technology Mapping



# Definition



Technology mapping is also referred to as *library binding*.

Given a Boolean network and a characterized cell library, generate a mapping of the network components onto cell library components with the objective of cost optimization or delay optimization.

AAC

# Input & Library

1.5 ns

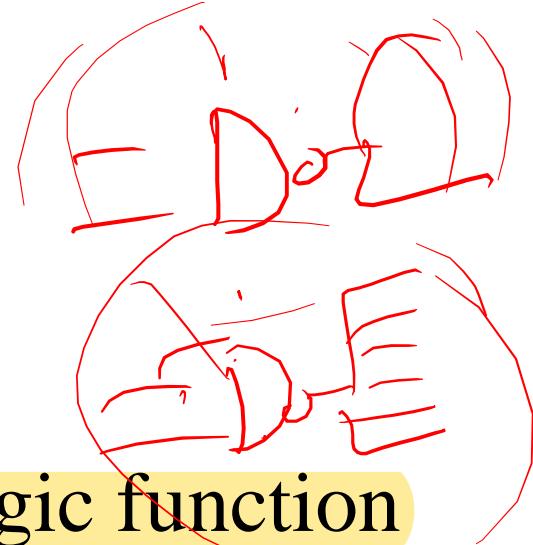
2 ns

≤ 0.001ns

- Input: Boolean network - Technology independent optimized network; typically a multi-level network
- Library:
  - Characterization in terms of area, delay and power
  - Enumerated or implicit library cells

A11C

# Typical Library



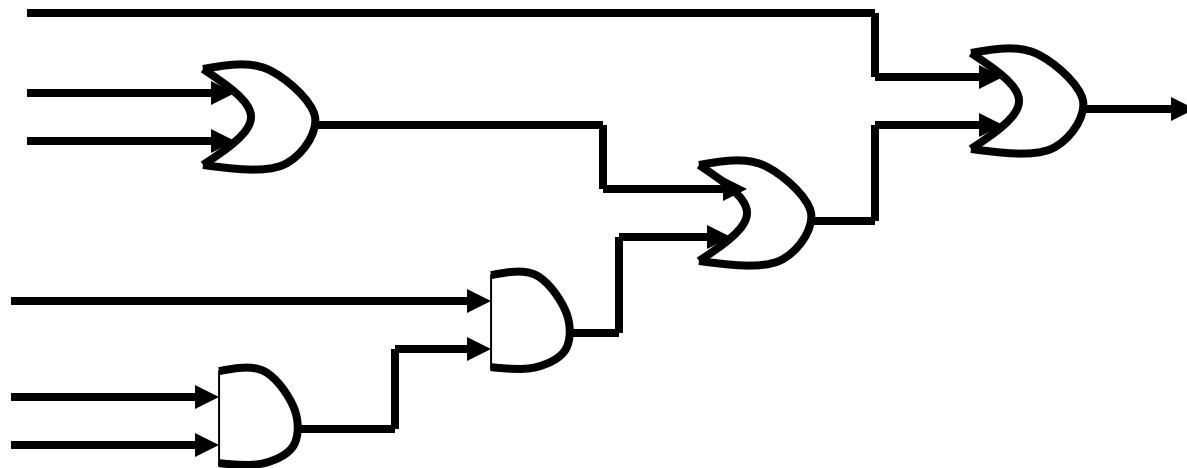
A typical simple library cell :-

- a single output **combinational logic function**
- cost in terms of **area**
- delay in terms of **propagation delays** for each input/output pair and as a function of load and/or fanout. Sometimes only the worst case values are stored.
- **power** in terms of average current

# Network Covering

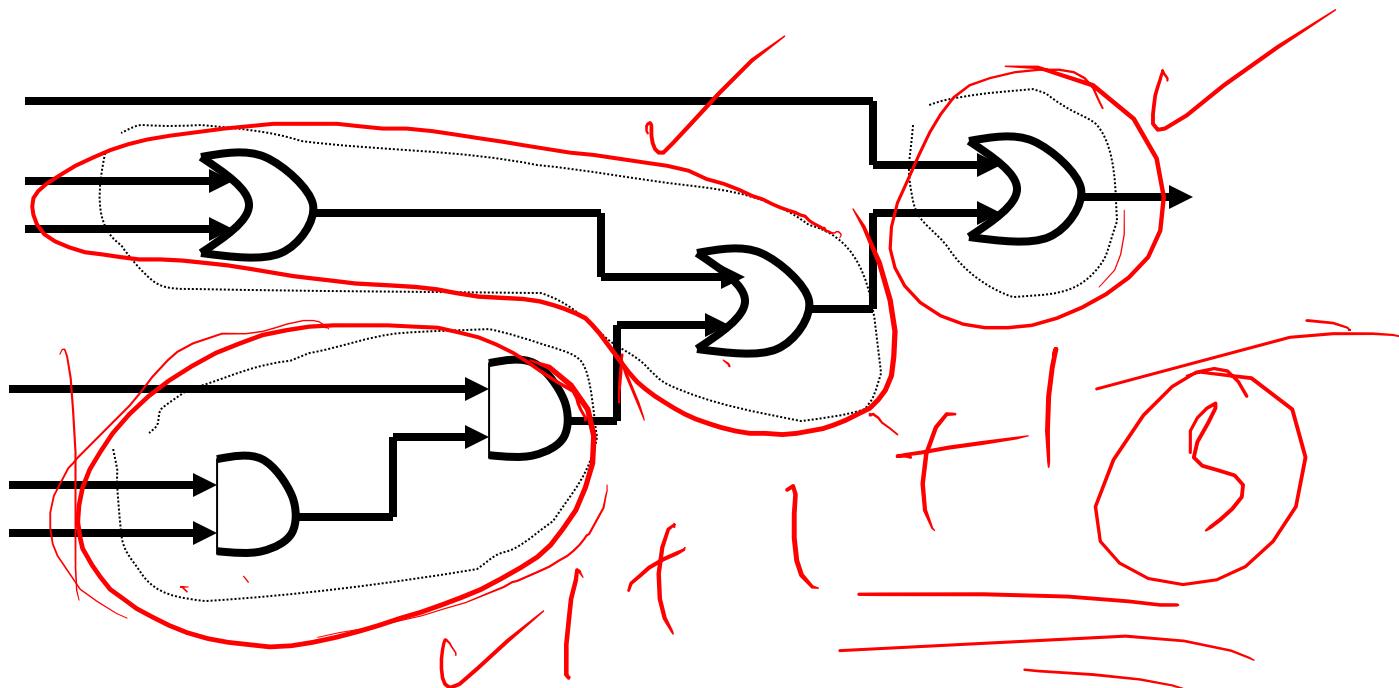
Network covering implies replacement of the sub-networks of the original network with cell library instances. Covering entails recognizing the equivalence of library cell to the identified sub-network and selecting adequate number of them to cover the whole network.

# Example 1



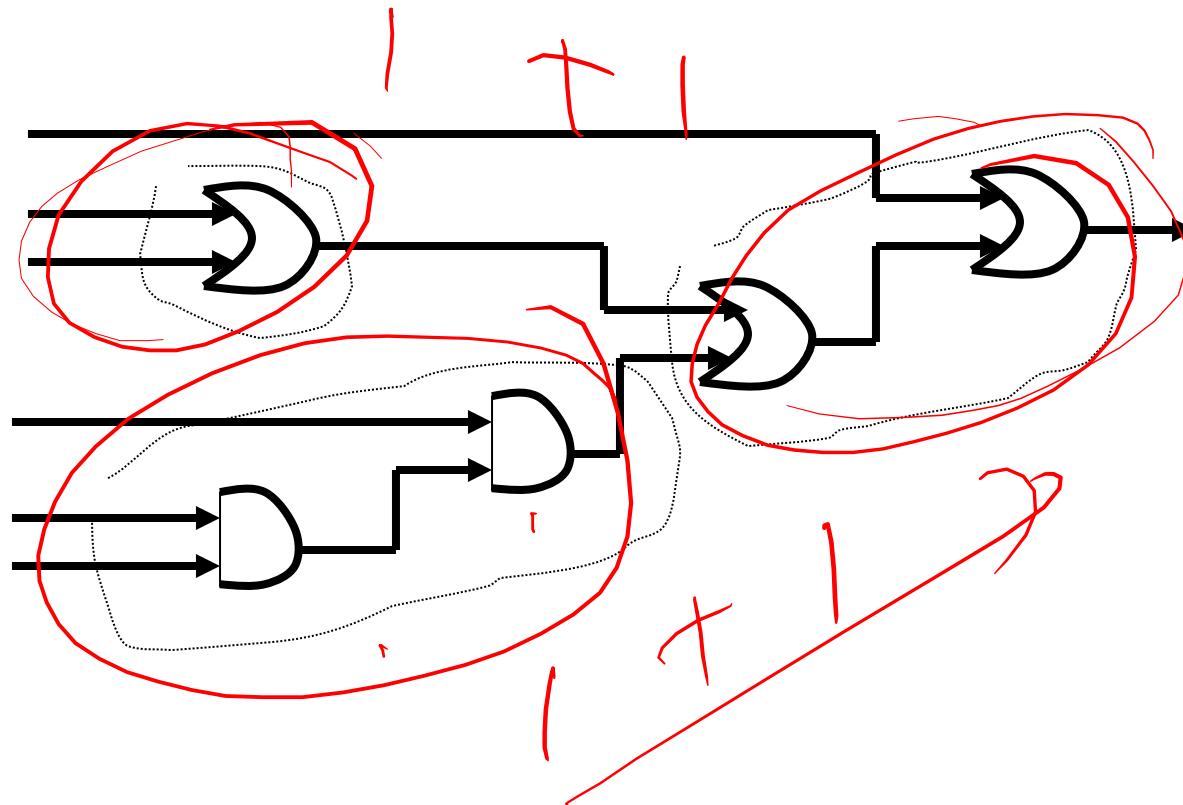
Cell library consists of two and three input gates

# Example: First Mapping



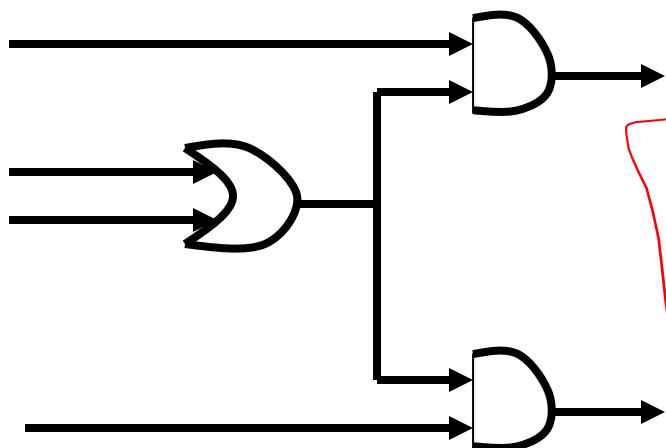
Cell library consists of two and three input gates

# Example: Second Mapping



Cell library consists of two and three input gates

## Example 2

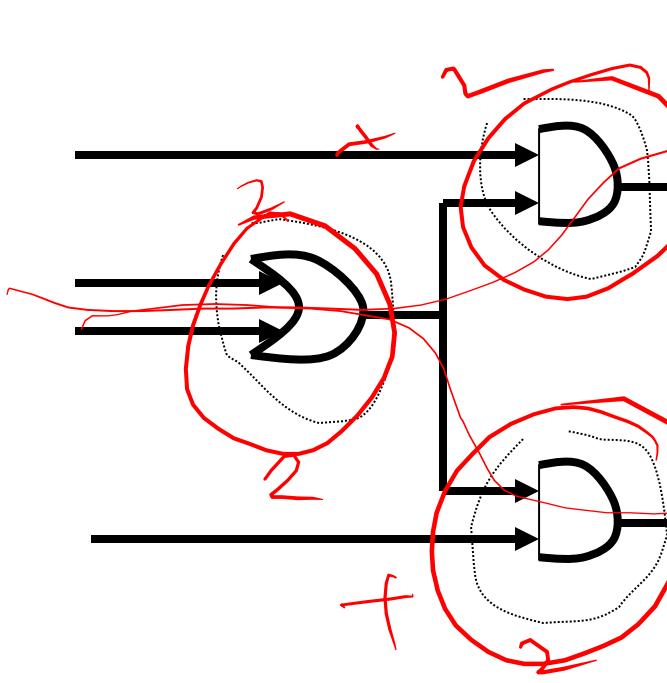


Cell library consists of

Component Area Delay

AND2	3	2
OR2	3	2
OA21	5	3

# Example: First Mapping



Cell library consists of

Component Area Delay

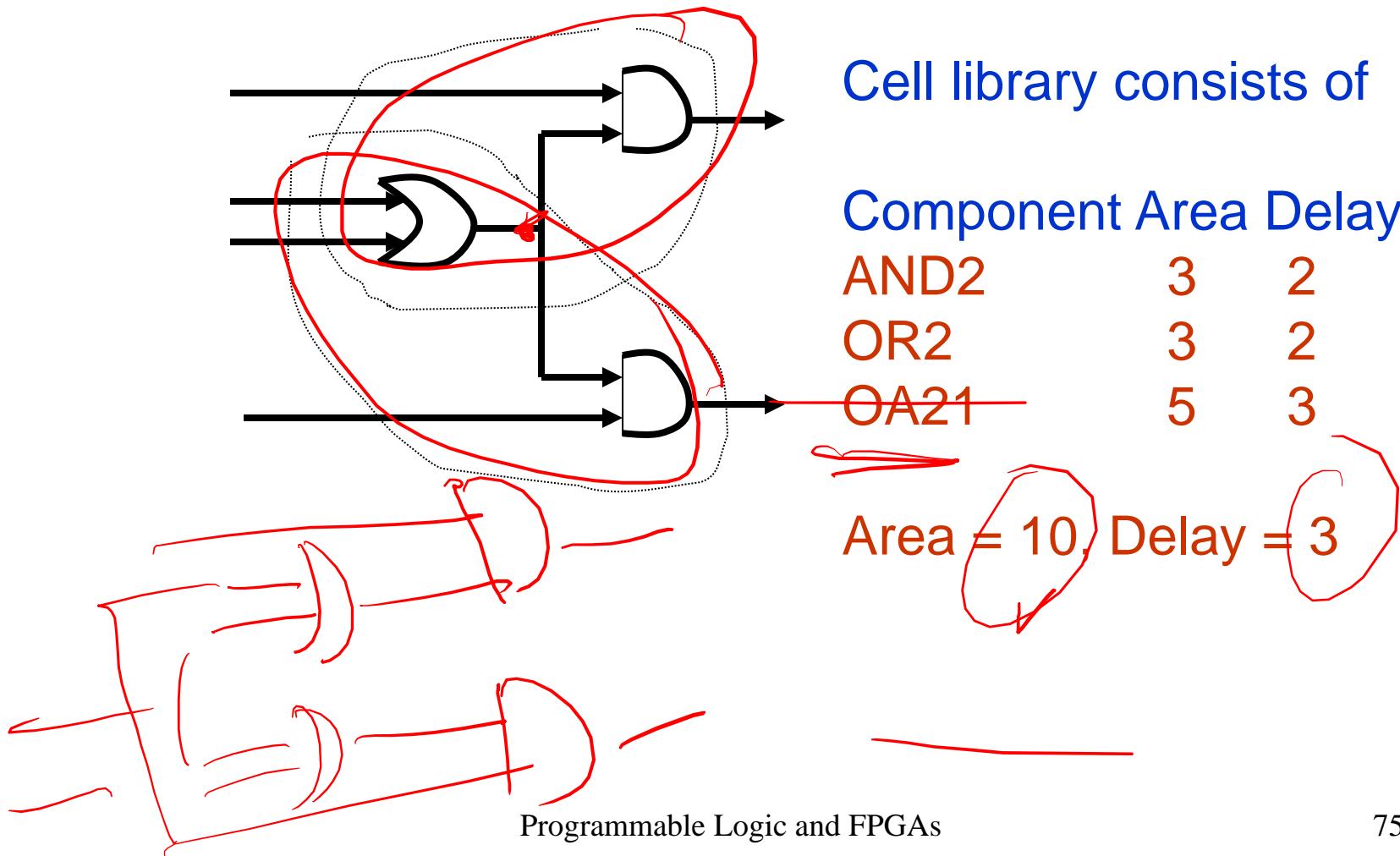
AND2            3      2

OR2            3      2

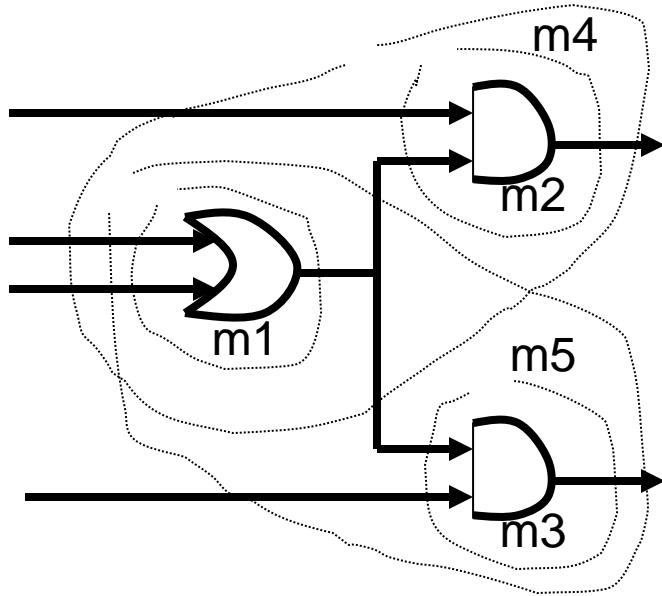
OA21           5      3

Area = 9, Delay = 4

# Example: Second Mapping



# Example



Cell library consists of

Component Area Delay

AND2            3      2

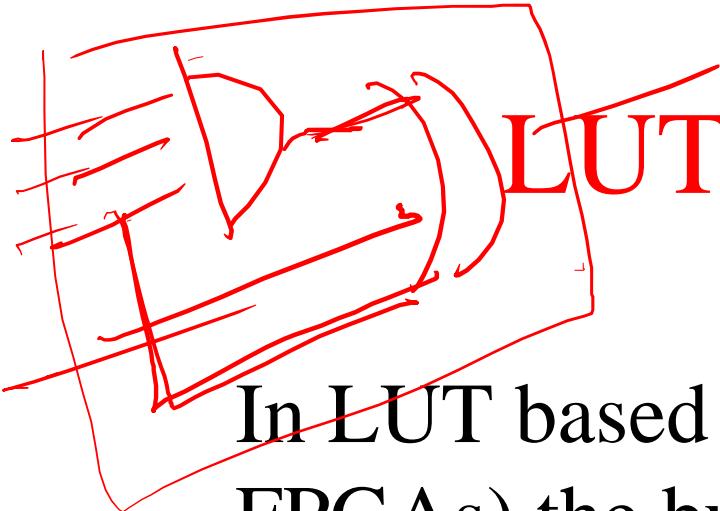
OR2            3      2

OA21           5      3

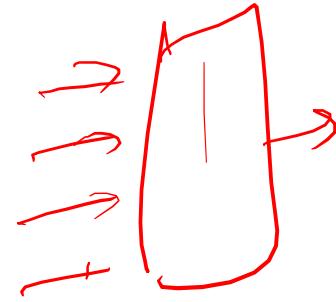
$$(m_1 + m_4 + m_5)(m_2 + m_4)(m_3 + m_5)(m_2' + m_1)(m_3' + m_1) = 1$$

??

# FPGA Structures & Mapping

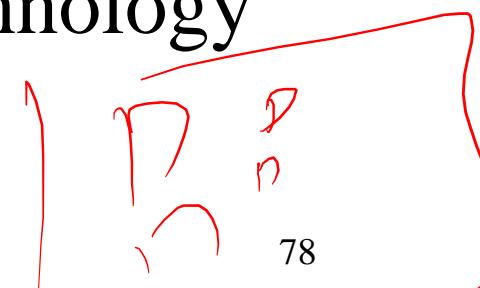


## LUT Based FPGAs

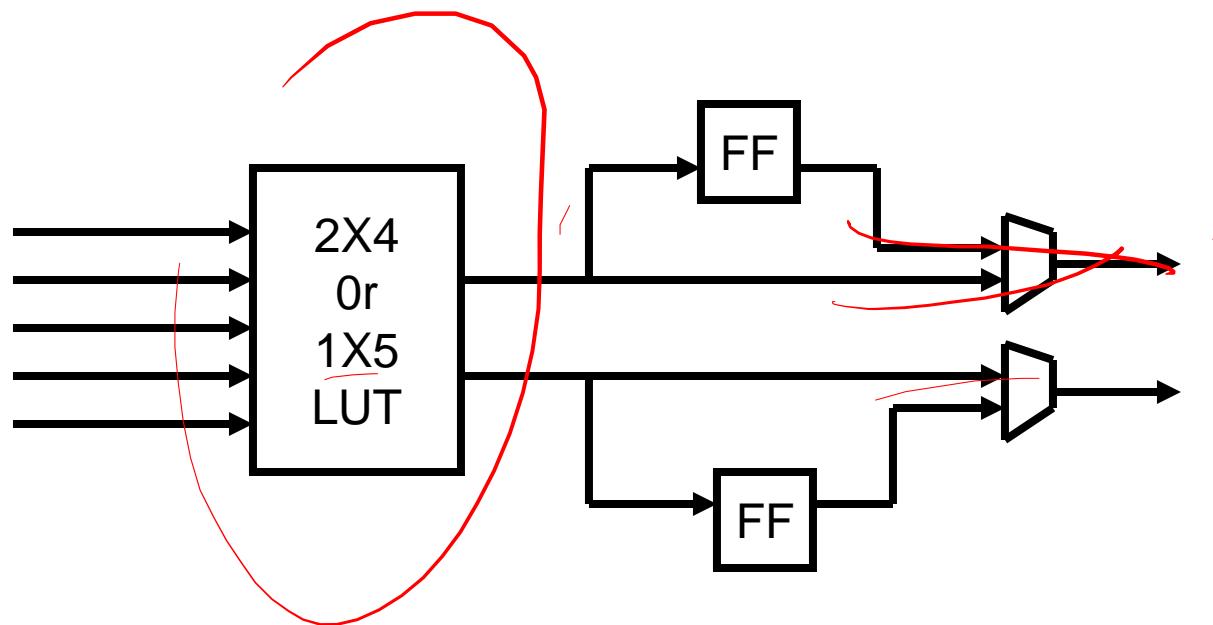


In LUT based FPGAs (example XILINX FPGAs) the building blocks are LUTs and Flip-Flops. A n-input LUT can implement all functions of n-variables.

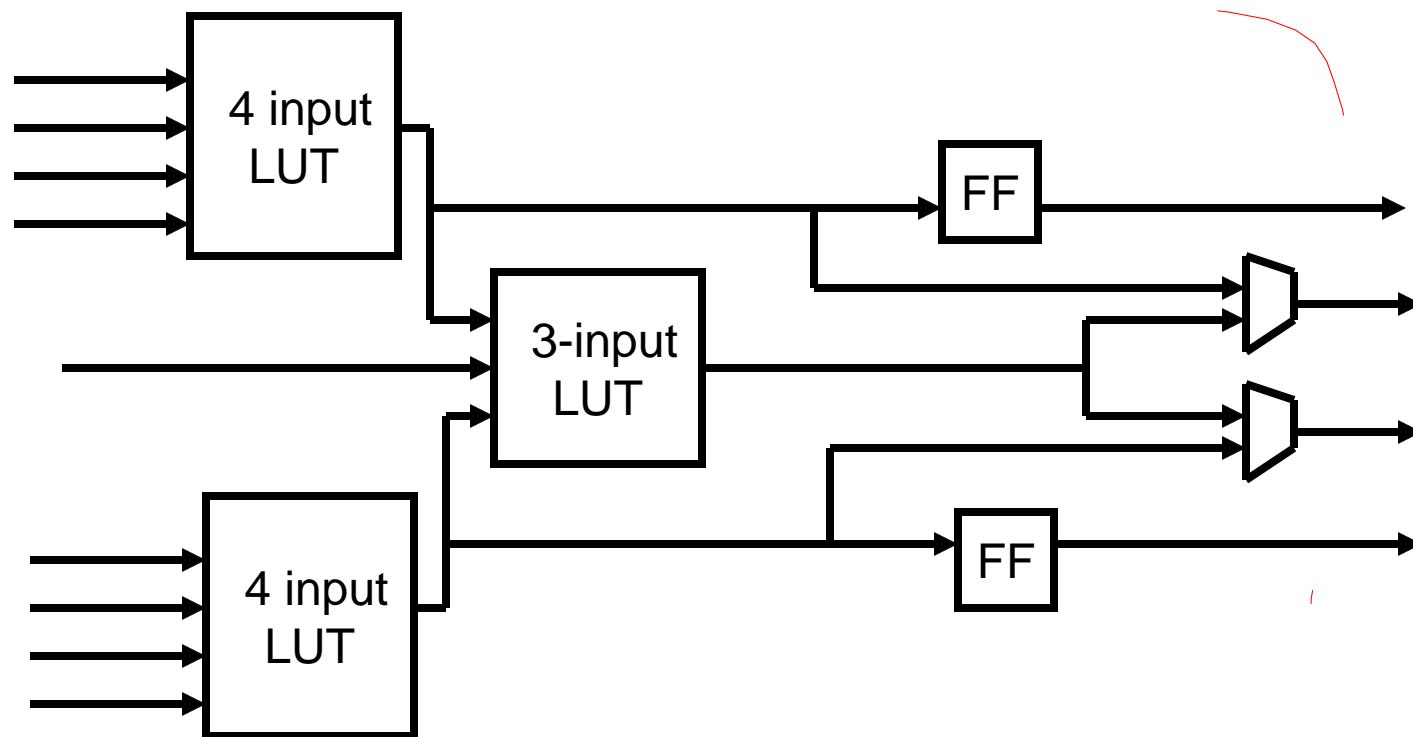
The FPGA itself is composed of CLB's with each CLB containing multiple LUT's and flip-flops which makes the technology mapping problem more complex.



# XC3000 CLB



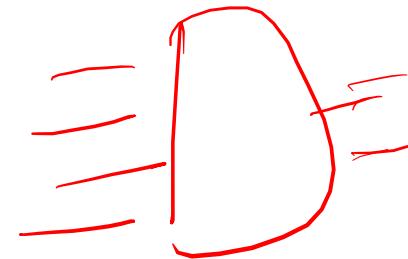
# XC4000 CLB



# Mapping Objectives

- Cost optimal mapping
  - Minimizing the number of LUTs
  - Minimizing the number of CLBs
- Delay optimal mapping
  - Minimizing the number of LUT levels
  - Minimizing the delays (including routing delays)

# Cost Optimal Mapping

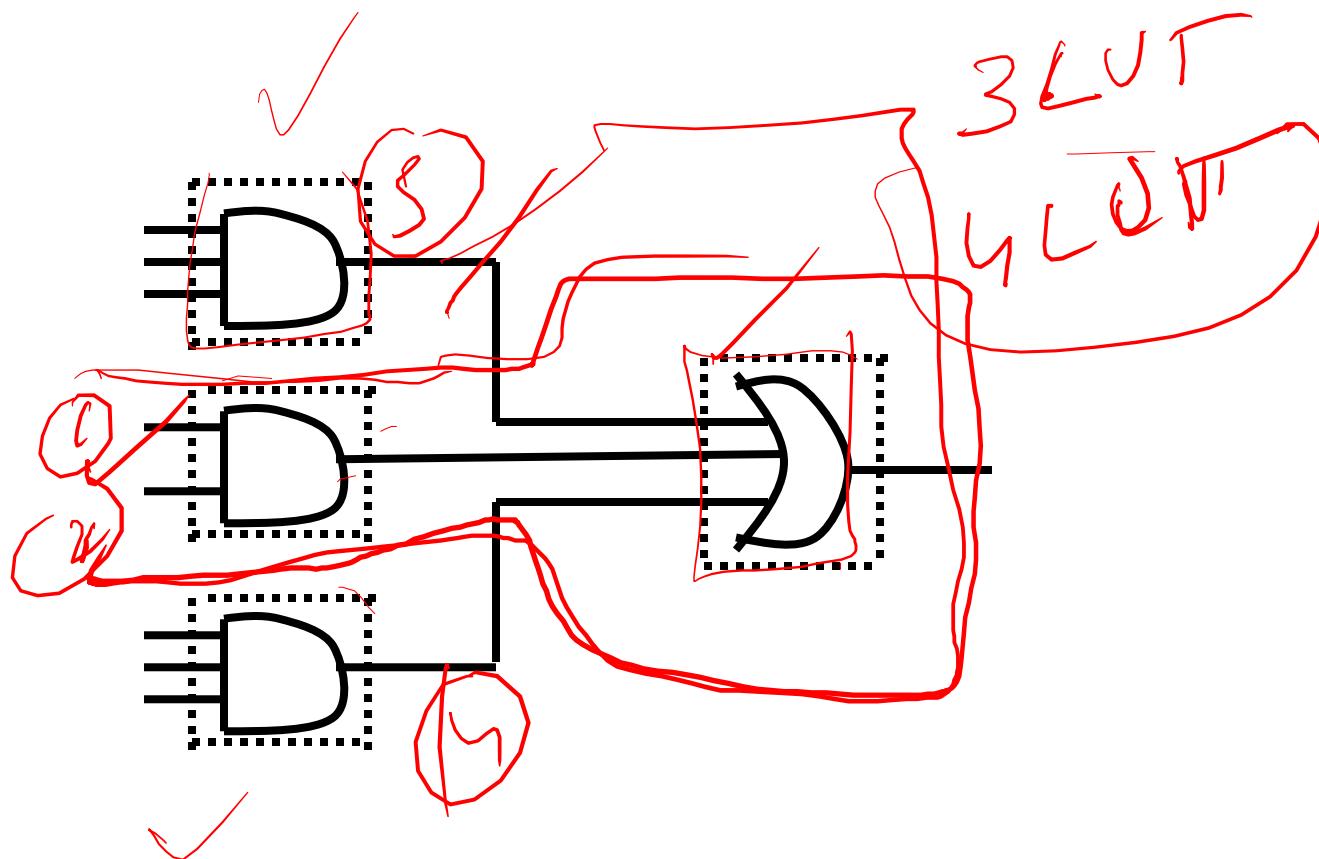


The problem of  $k$ -input LUT maps can be mapped to the problem of bin packing. We have to minimize the number of bins each with a capacity of  $k$ .

Assume the starting point is a gate-level netlist with each gate containing less than equal to  $k$  inputs.

Each gate can be packed into one bin.

# Example: Simple Mapping

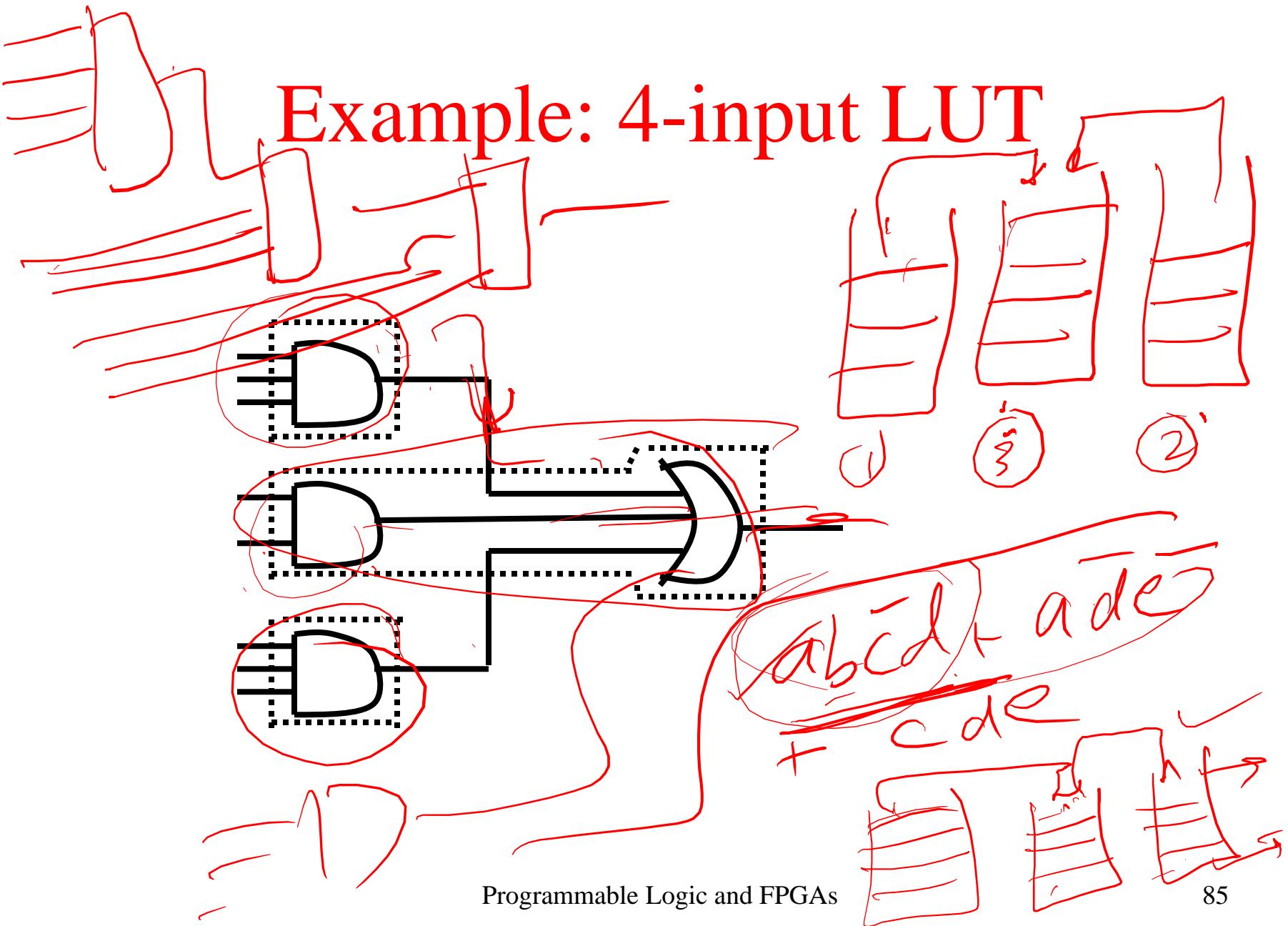


*A.B.C*      *CDE*

## Sum of Products: Bin Packing

- Select the product term with the most number of variables and fit it into any table where it fits and if it doesn't fit anywhere add a new table
- The table with the fewest number of unused inputs is declared as final
- Associate this output with the first table that can accept it

# Example: 4-input LUT



2LUT



WOND

NOR

ENOR

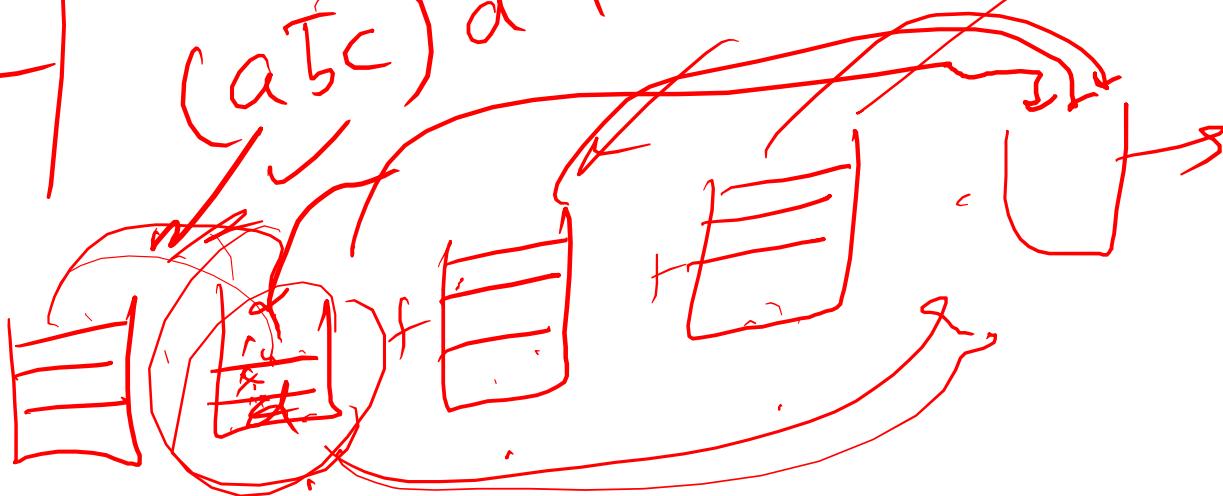
ENAND

$$a\bar{b}cd + \bar{c}\bar{d}e + bde$$

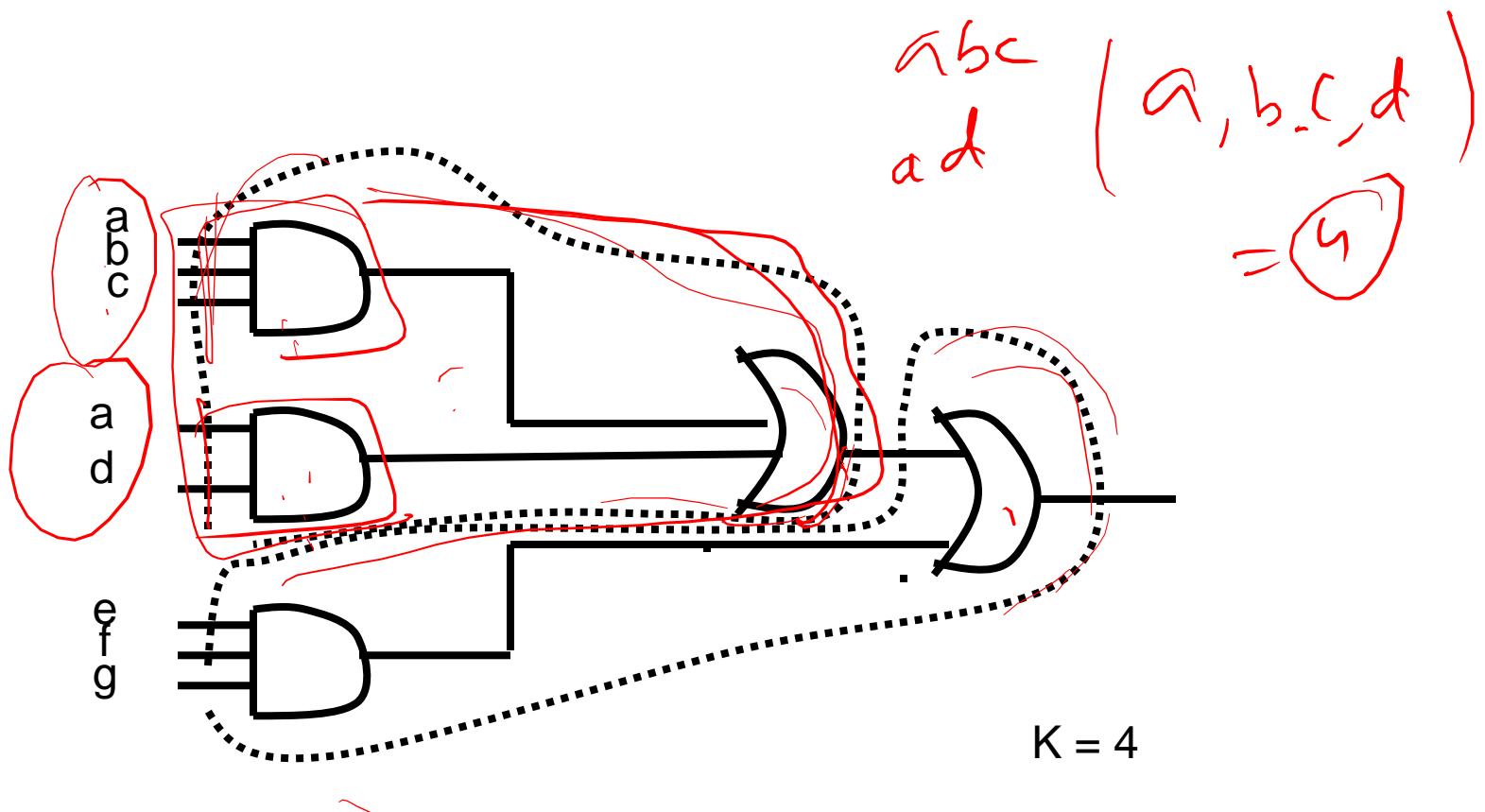
$$a\bar{b}c\bar{d} + c\bar{d}e + b\bar{d}e$$

$$(ab\bar{c})d + c\bar{d}e + b\bar{d}e$$

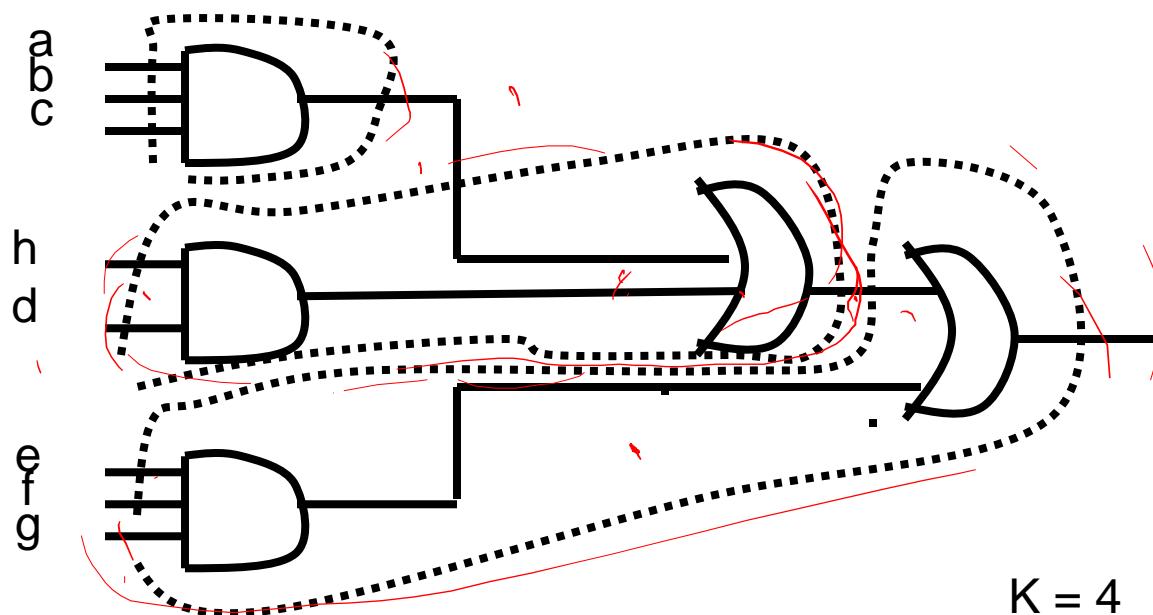
3LUT +



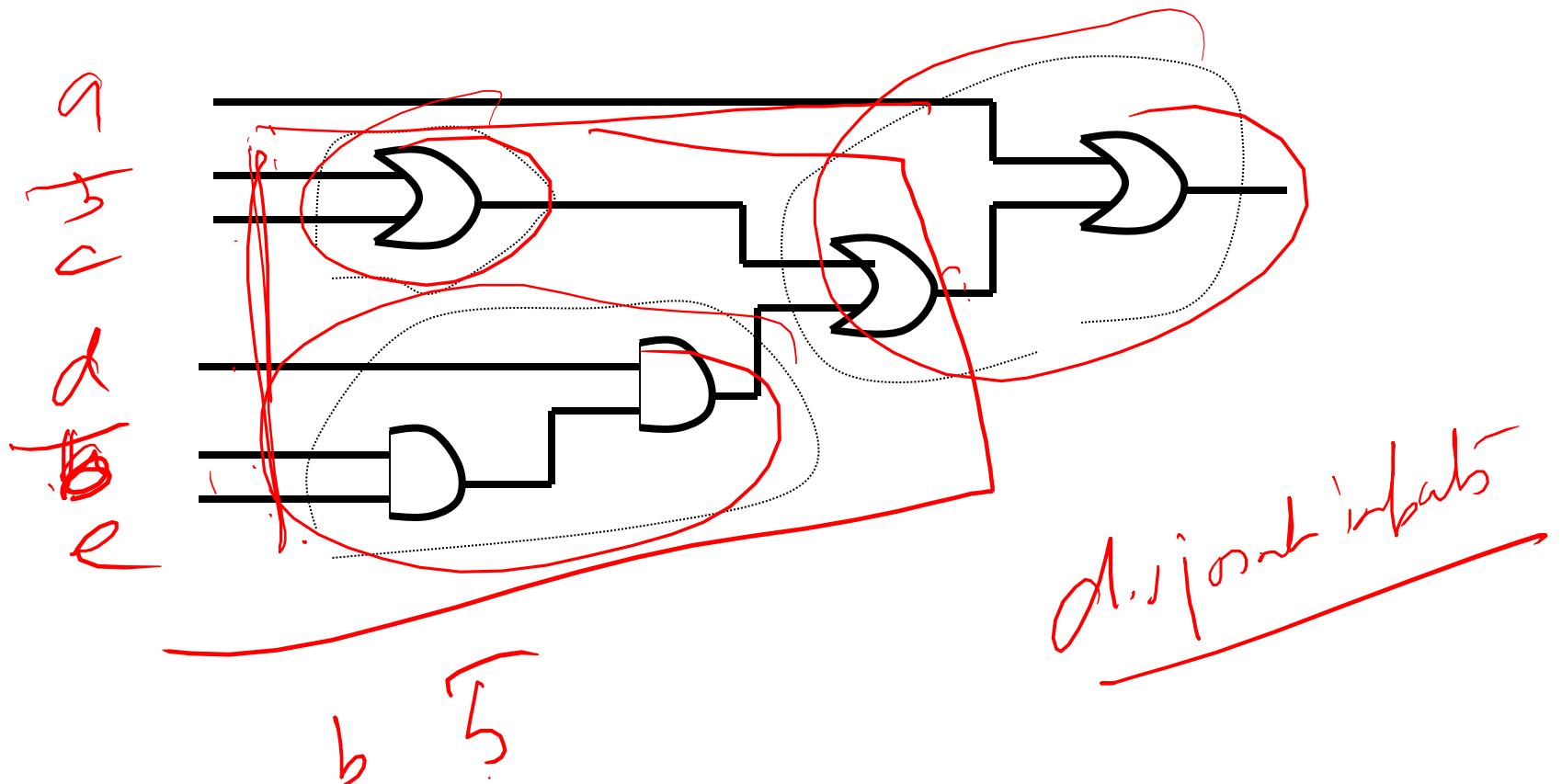
# Example: Overlapping Inputs



# Example: Decomposition



# Example: 3 input LUT



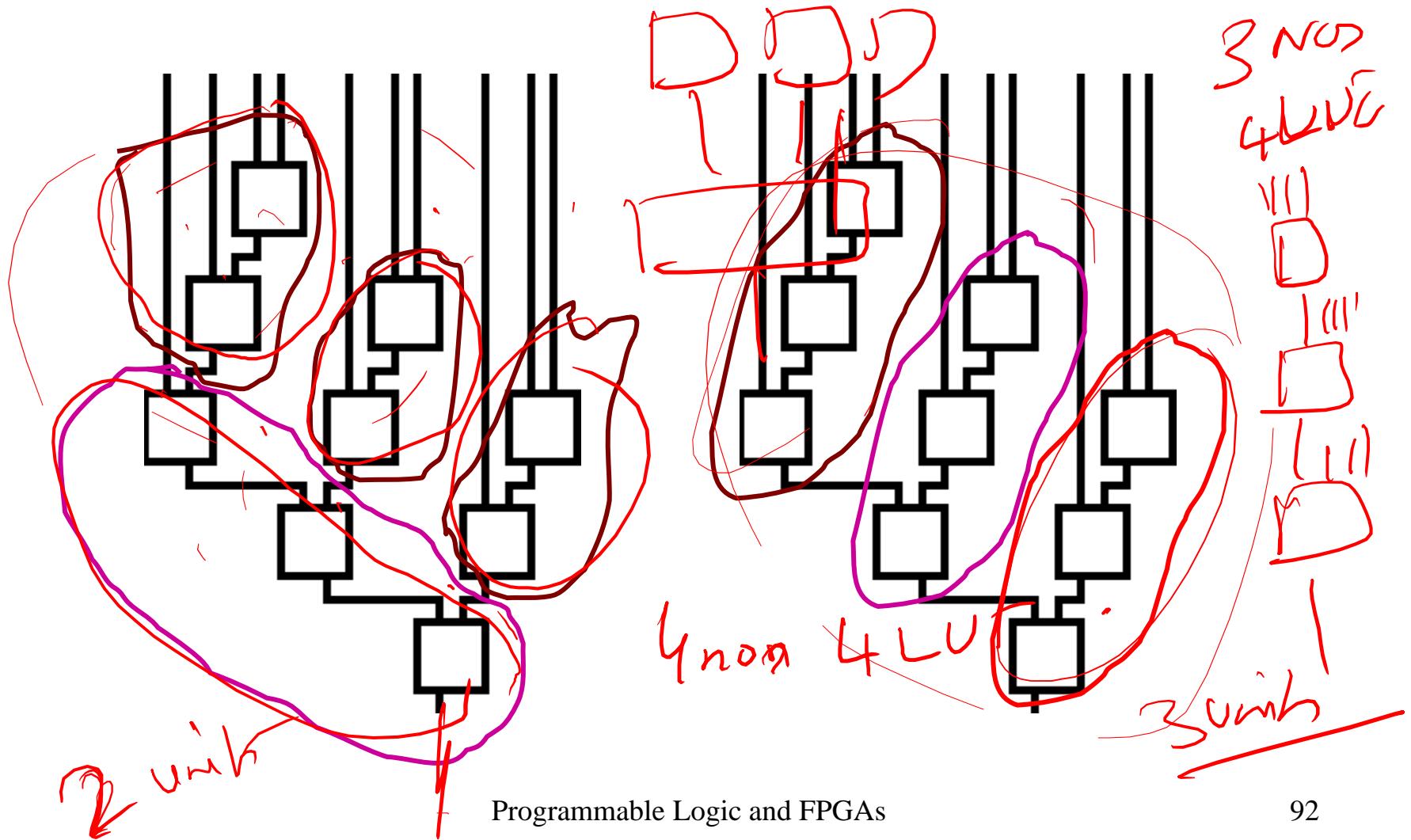
# FPGA Technology Mapping: Issues

# LUT Mapping

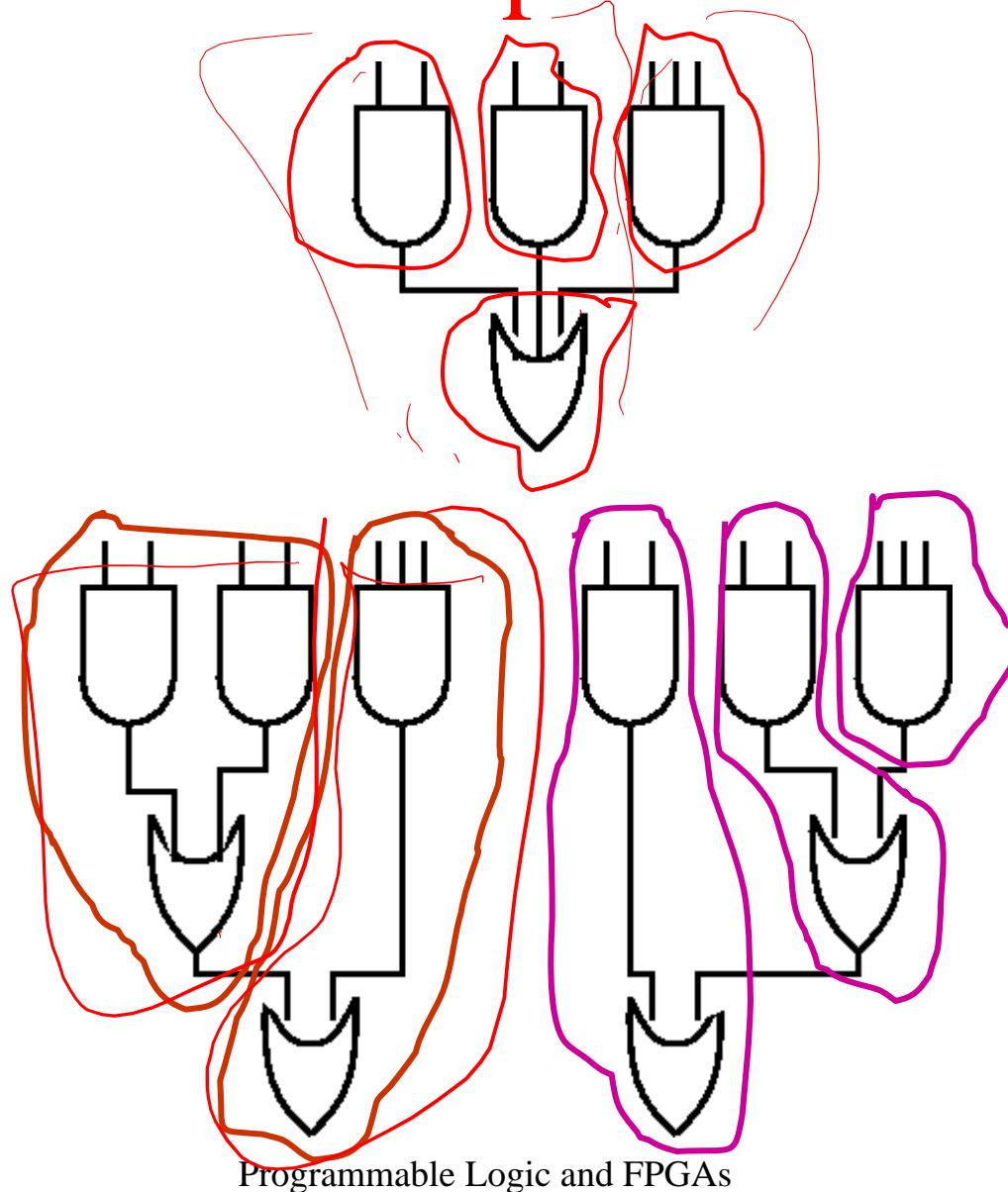
Starting from a technology independent optimized circuit, produce a minimal LUT cover for the circuit. The complexities are due to the following reasons.

- Fanout nodes
- Reconvergence
- Node decomposition and packing

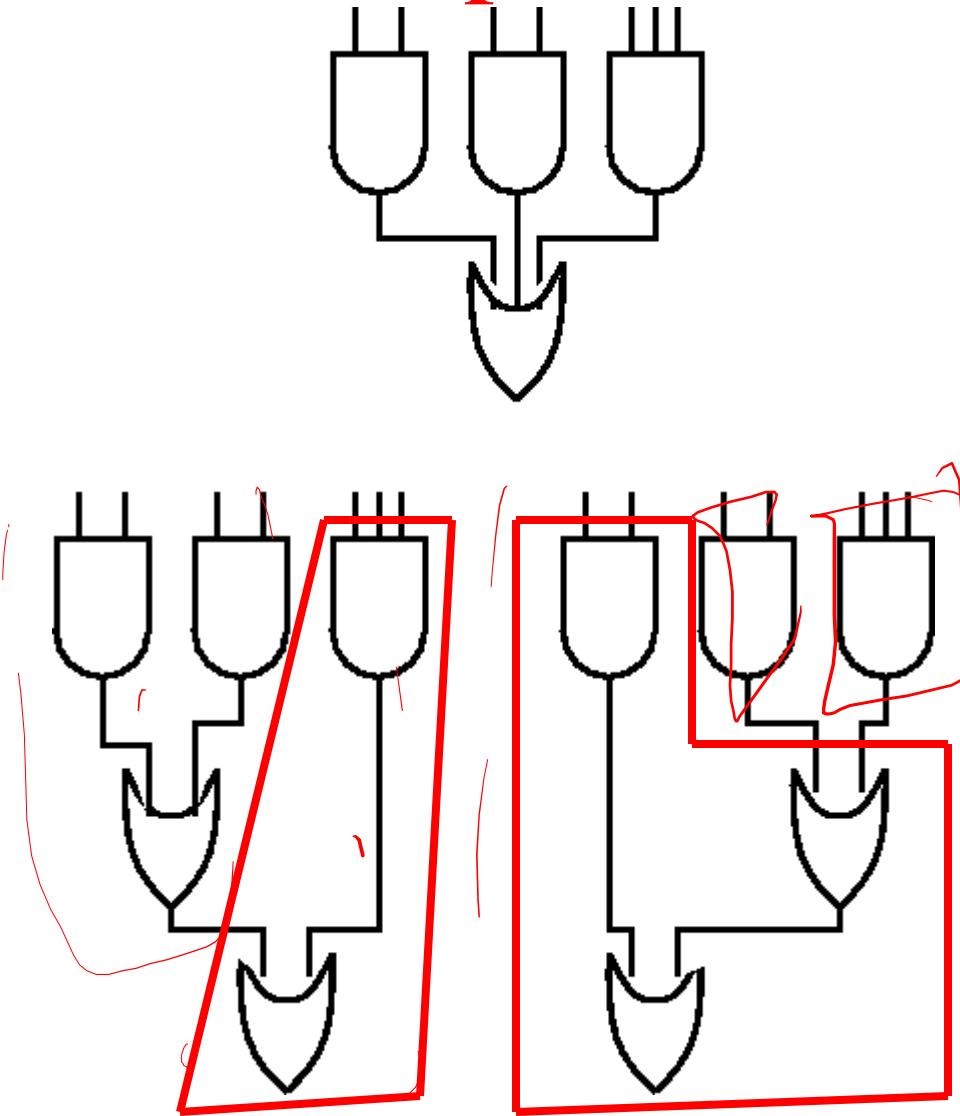
# Area vs. Delay



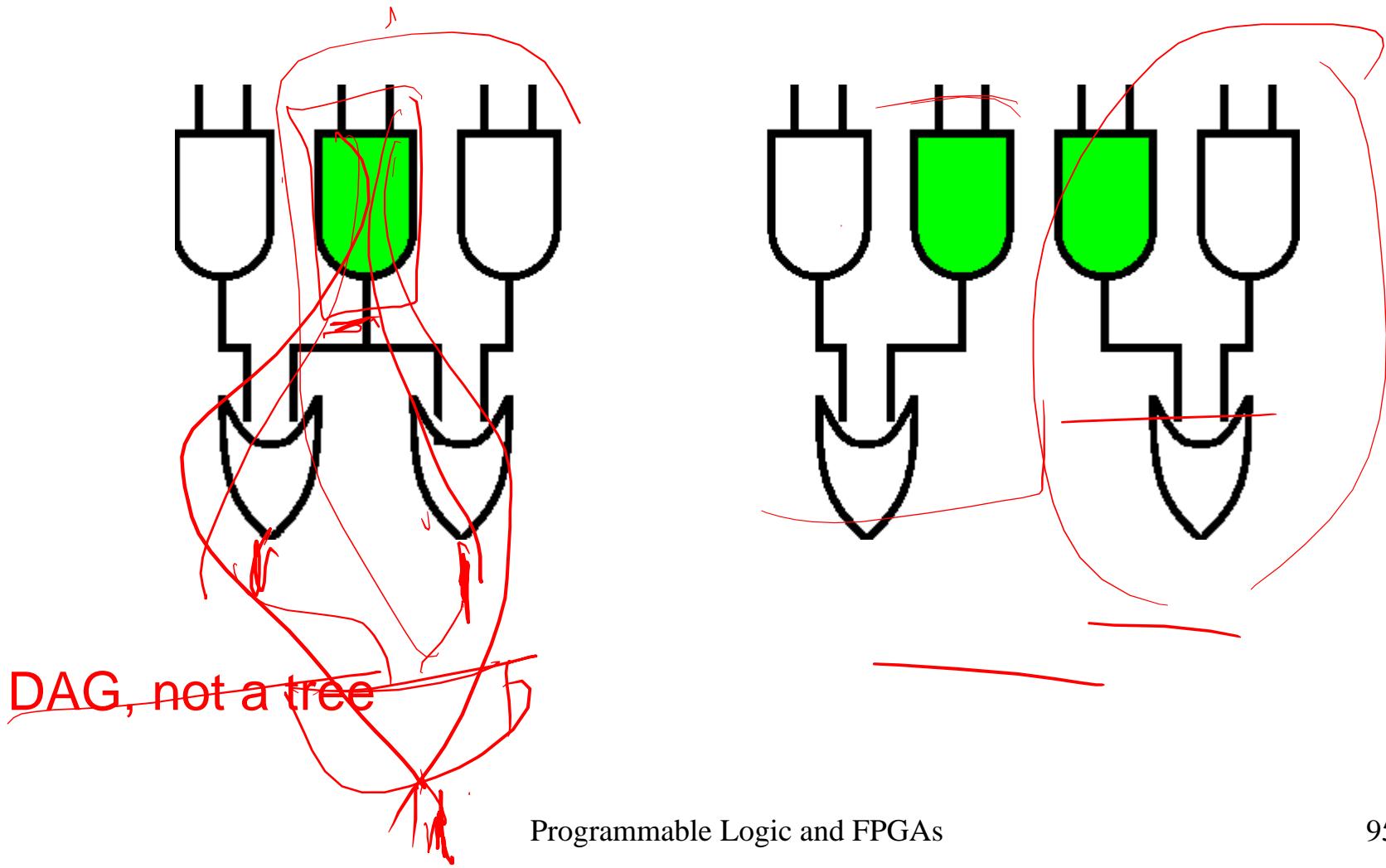
# Decomposition



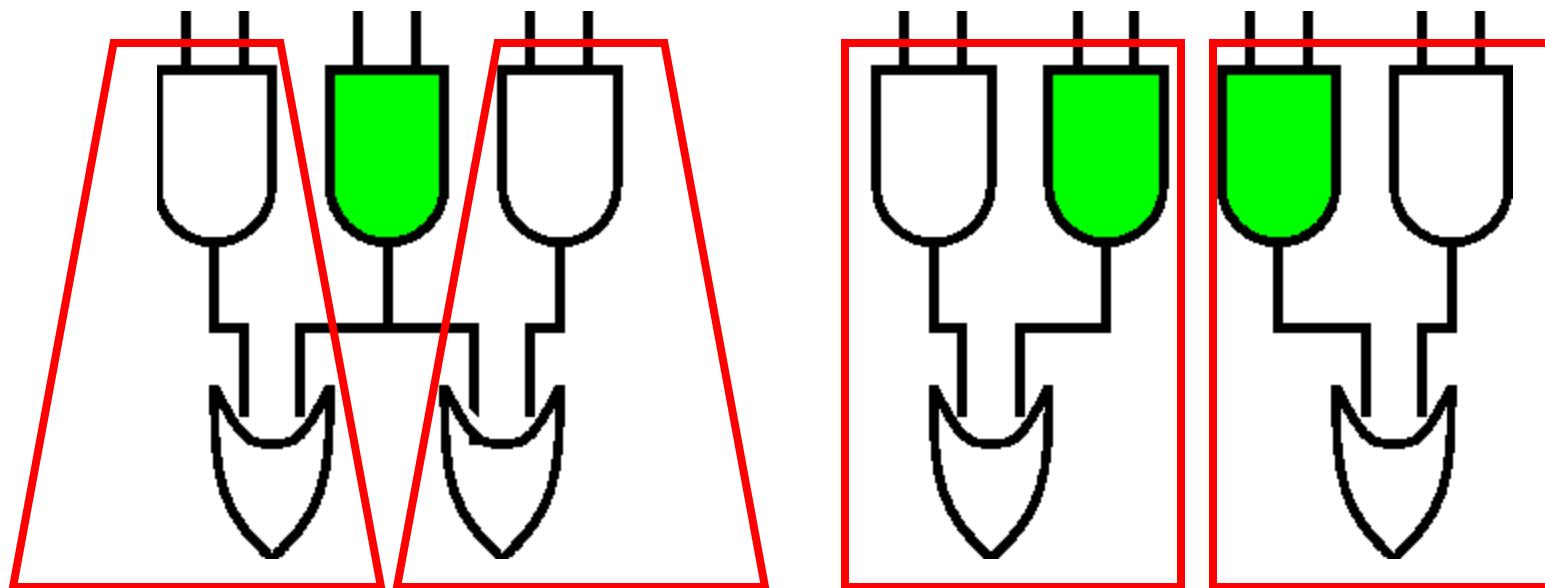
# Decomposition



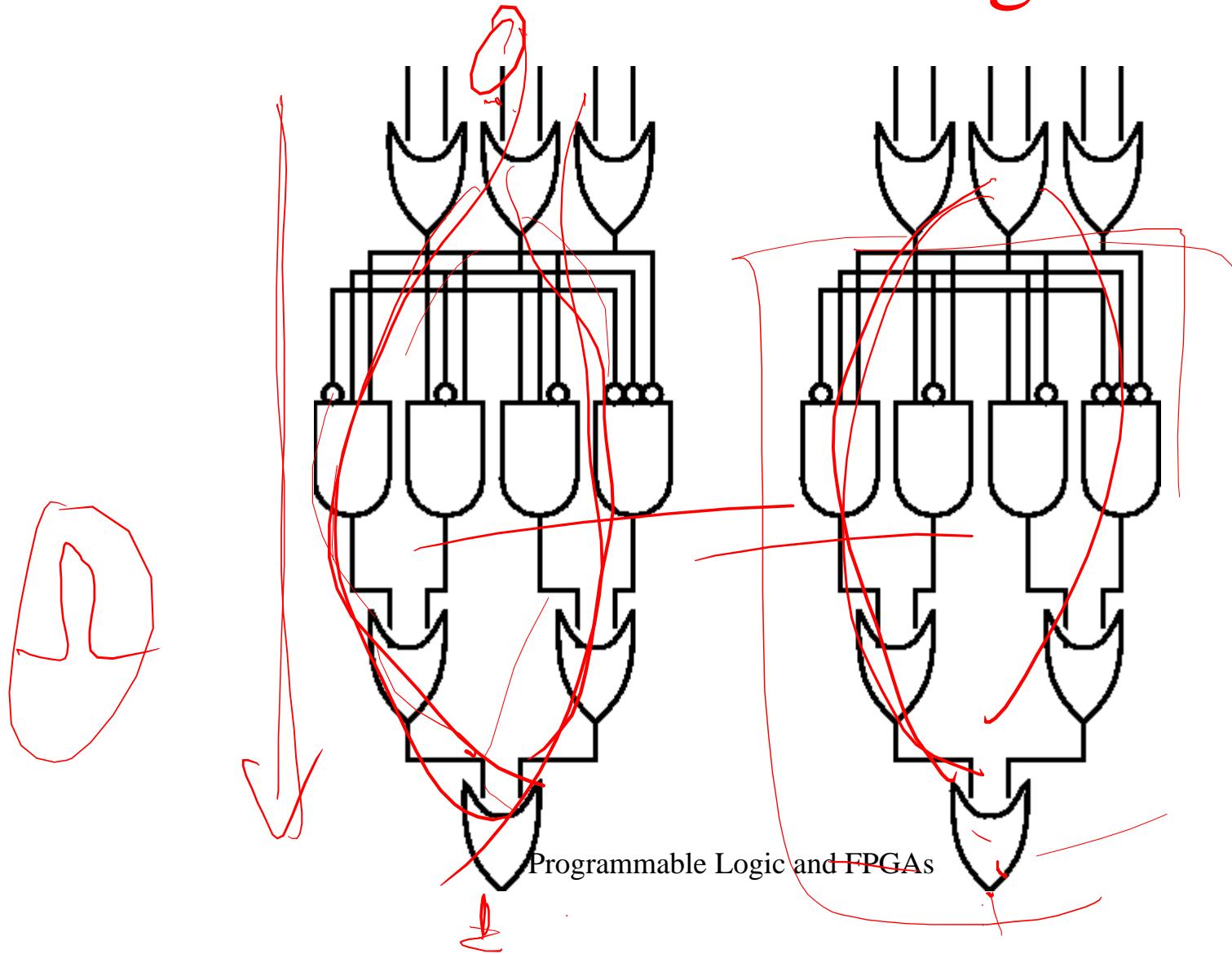
# Fanout: Replication



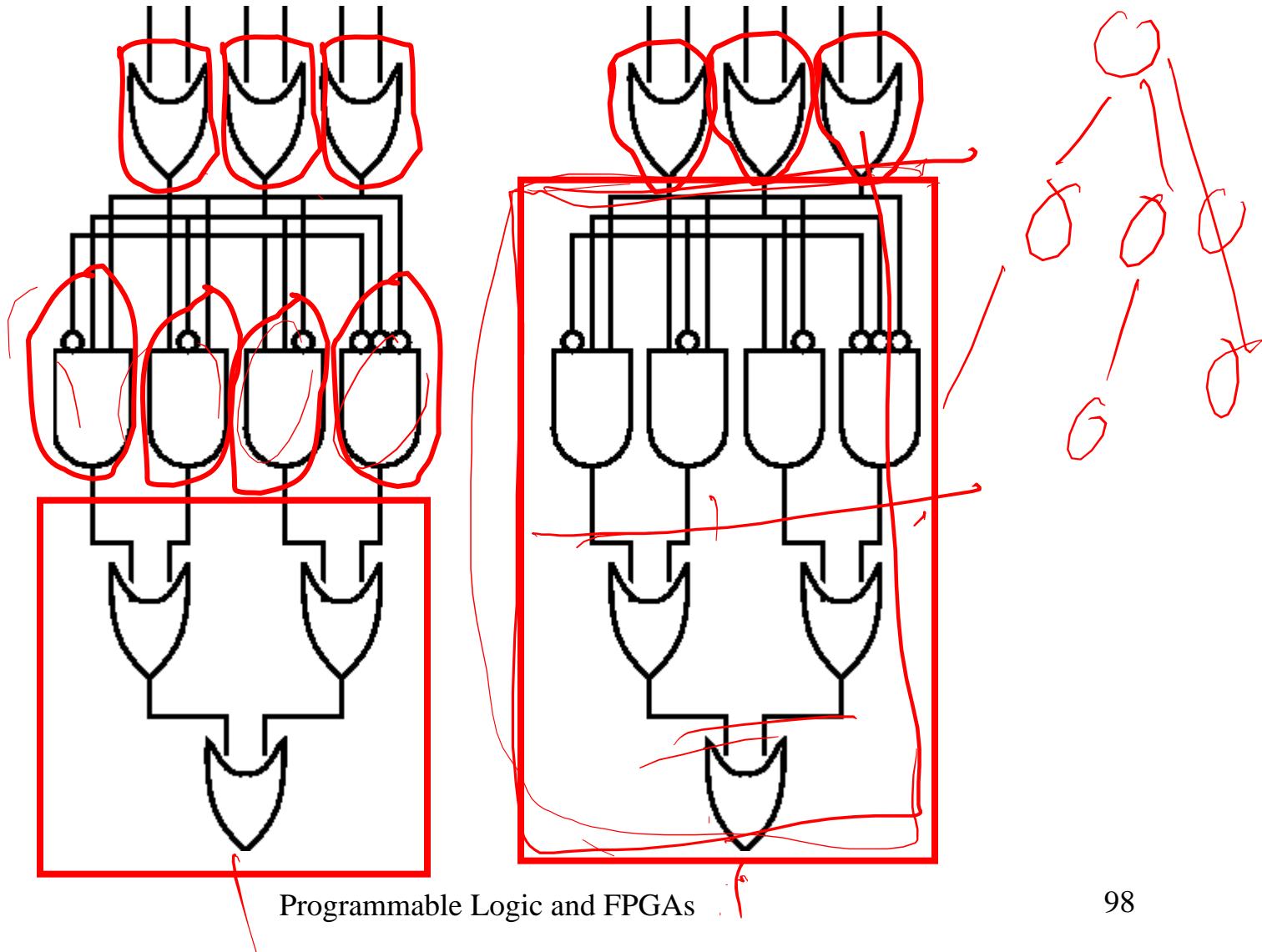
# Fanout: Replication



# Fanout: Reconvergence



# Fanout: Reconvergence



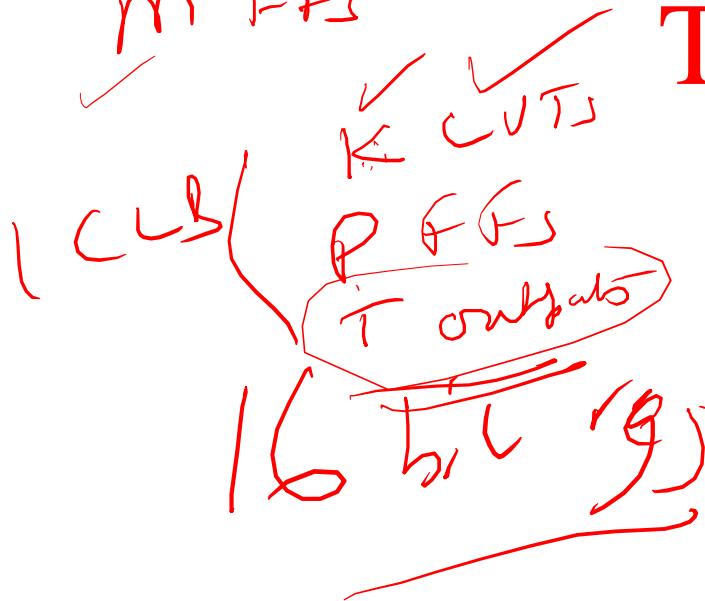
# CLB Mapping

Though direct mapping of technology independent circuit onto CLBs would involve function decomposition.

Alternatively, one can start from a circuit mapped onto LUTs and then pack them onto CLBs.

N LUTs

m FFs



Thank You



\$0

2 FFs

T < K  
16 LUT

T = K +  
T < K + P

Programmable Logic and FPGAs

Max LUT, FF

100

