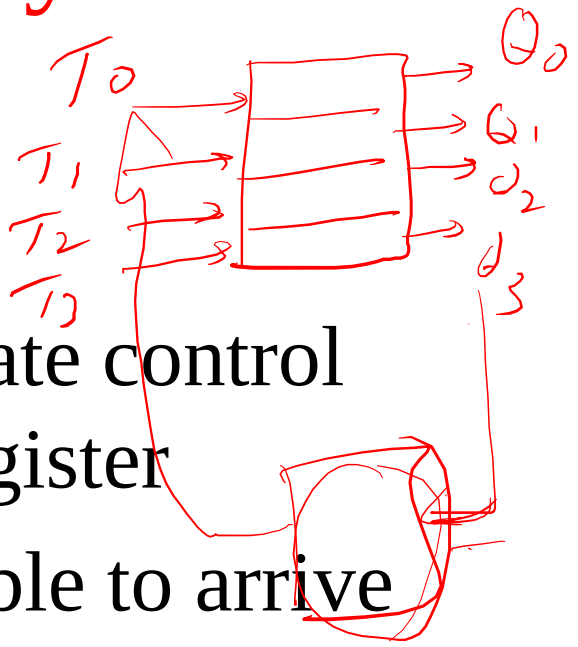
Lecture 23
Multiple State Machine Implementation & Clock Period

M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

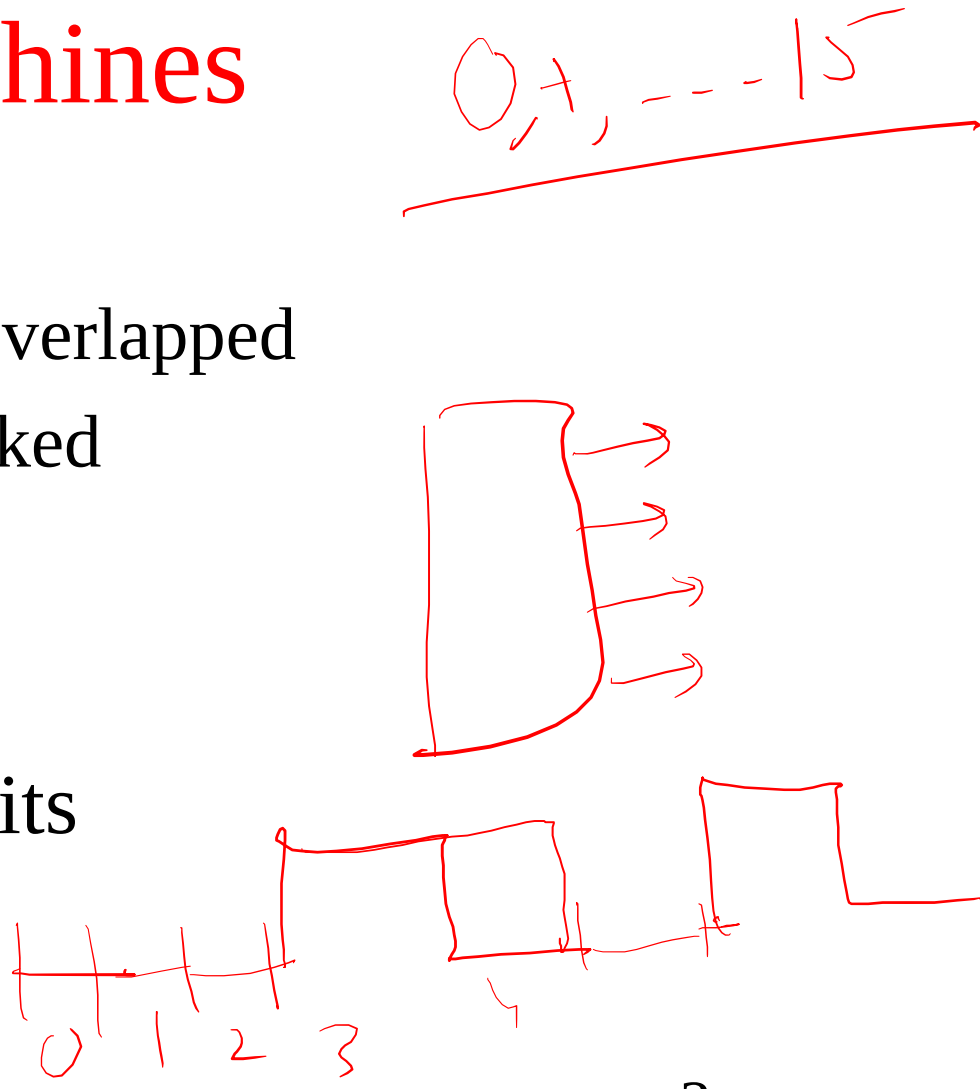
Steps in State Machine Synthesis using Counters



- 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

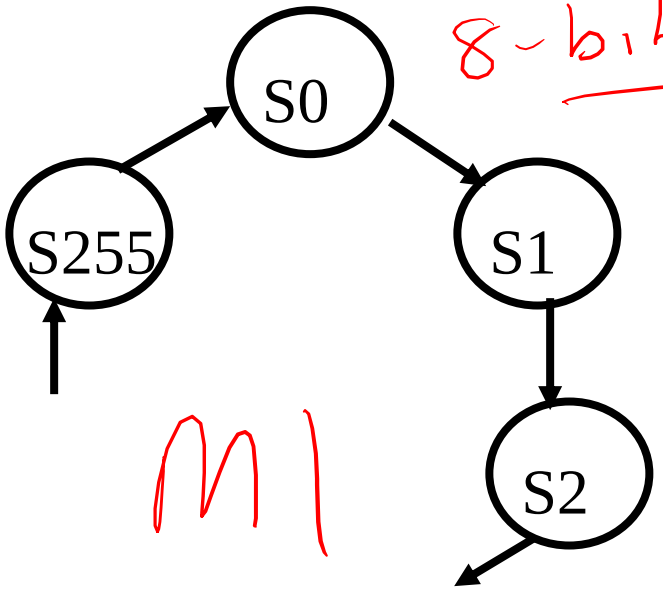
- Pattern matching
  - Overlapped or non-overlapped
  - Blocked or non-blocked
- Sequential decoding
- Controllers
- Memory based circuits

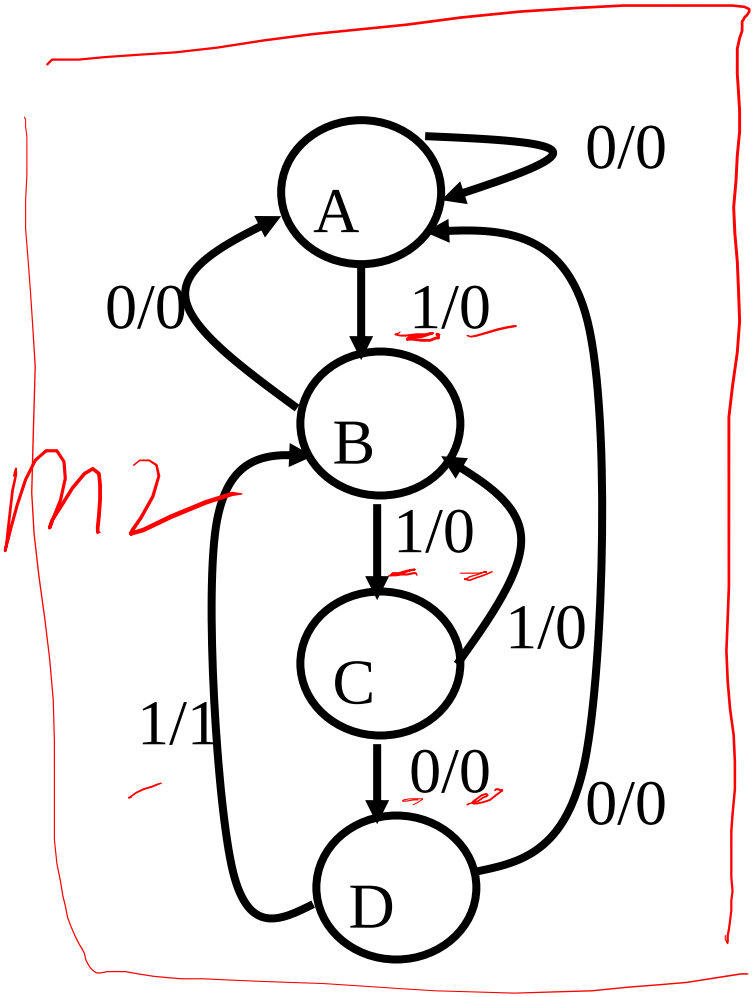


Interacting State Machines : Example

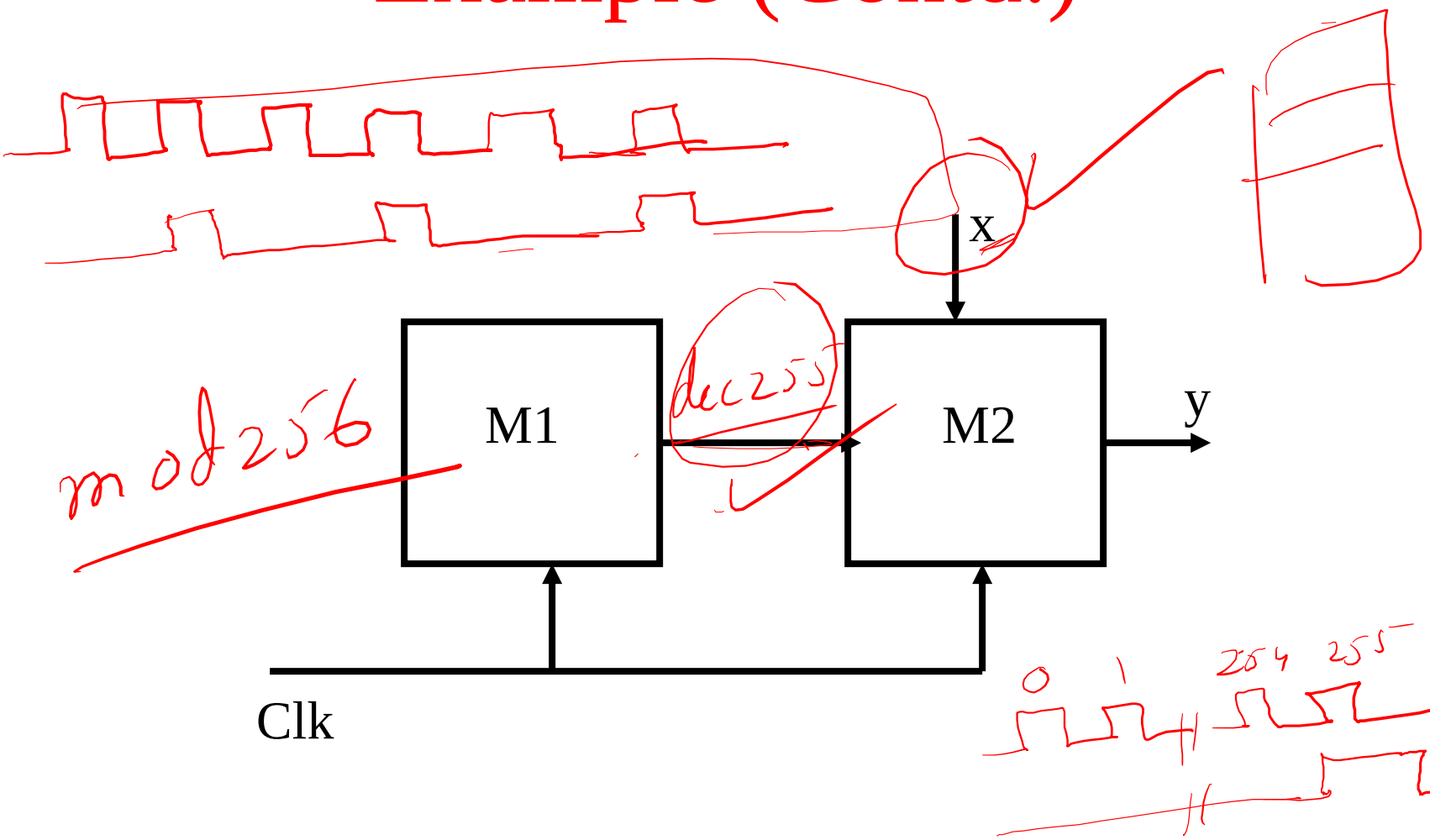
- Search for a pattern P = ‘1101’ within blocks of 256 bits. The pattern should not cross block boundaries.
- Design two state machines M1 and M2
  - M1 is a modulo 256 counter
  - M2 is the pattern recognizer
- The 256th transition of M1 should initialize M2

Example (Contd.)

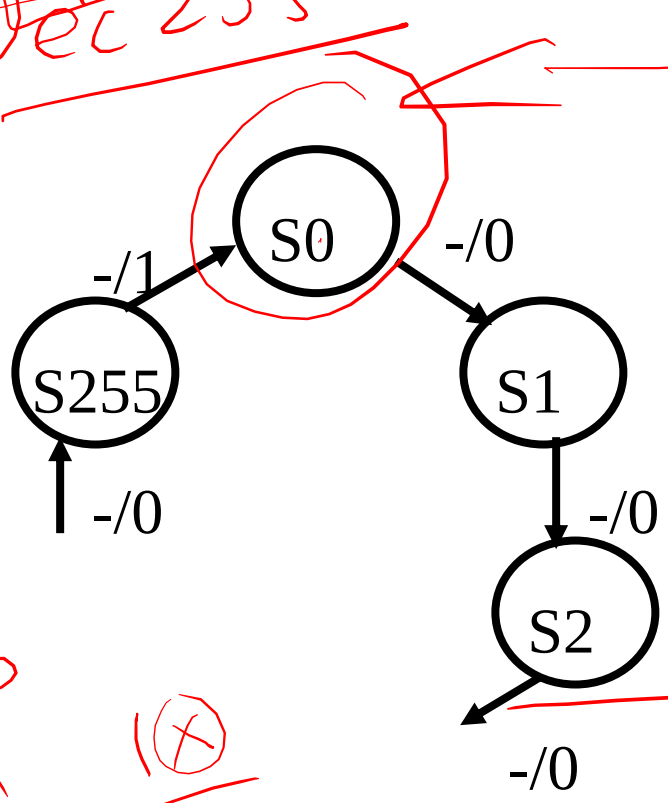


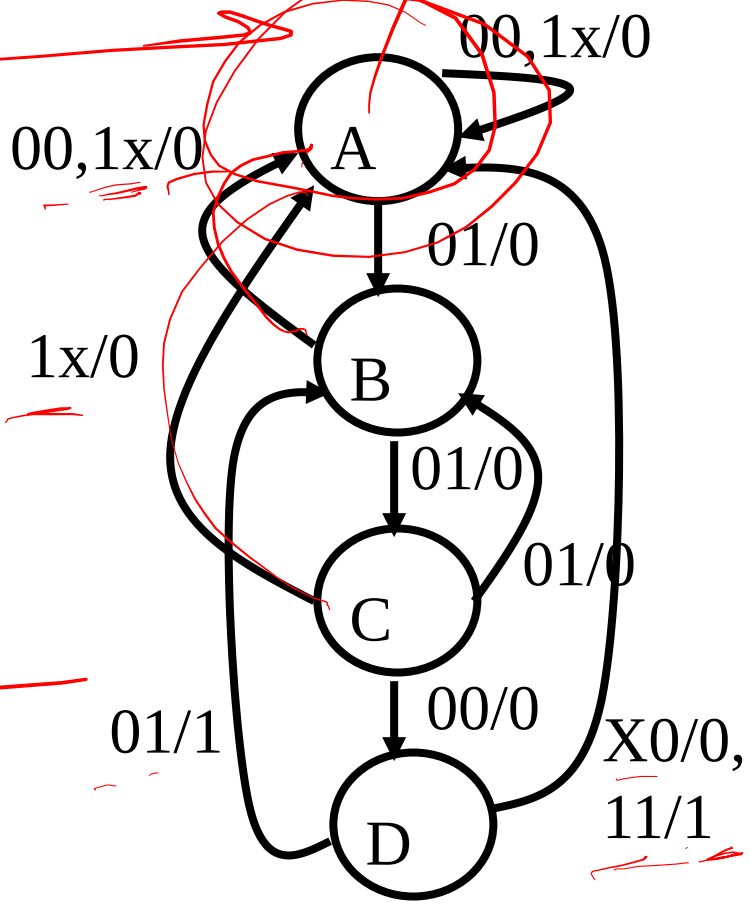


Example (Contd.)



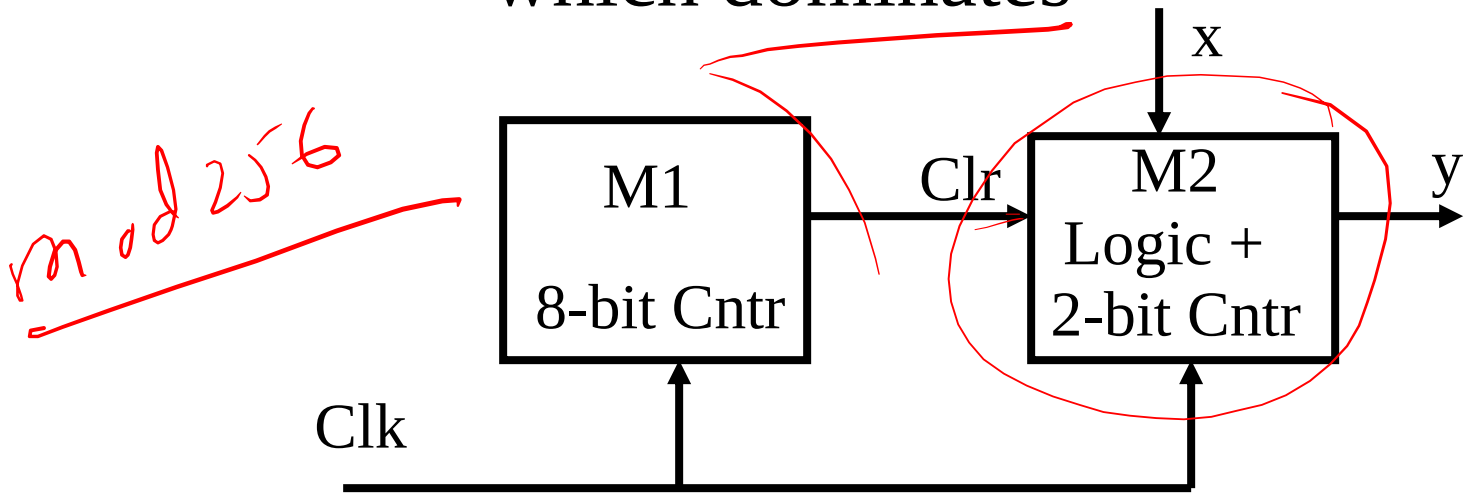
Example (Contd.)



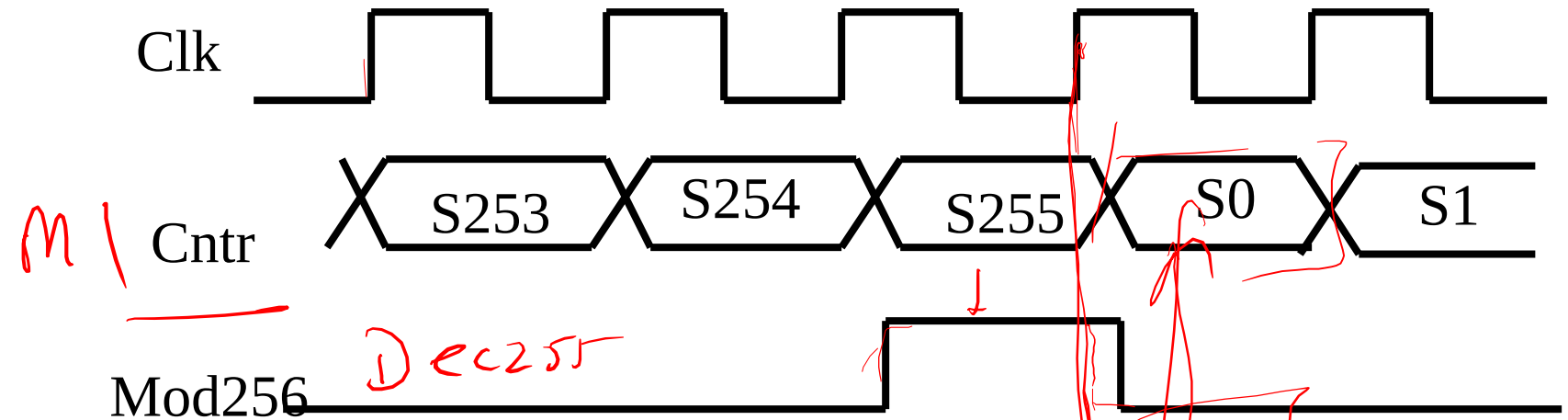


Design Summary: Example

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



Register & Latch Waveforms



Multiple State Machines: Another Example

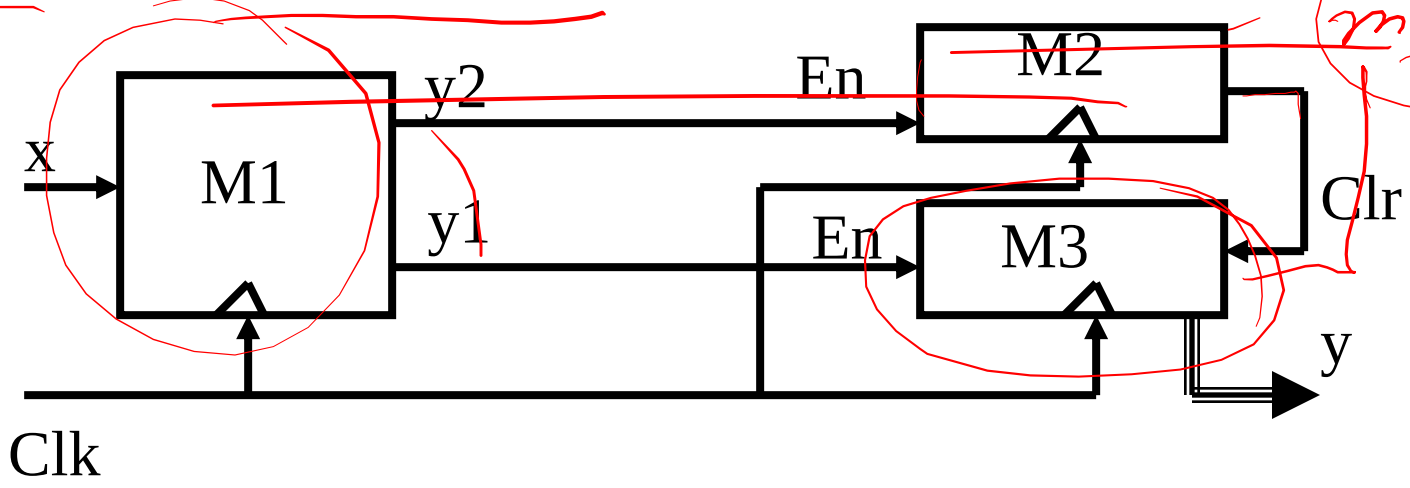
- In a bit stream, count the number of "@" (ASCII Code) characters in blocks of 256 8-bit characters
- Three state machines: M1, M2 and M3
  - M1: Pattern recognizer for "@" character
  - M2: 8-bit counter for counting 256 characters
  - M3: 8-bit Counter for counting no. of "@"

Second Example (Contd.)

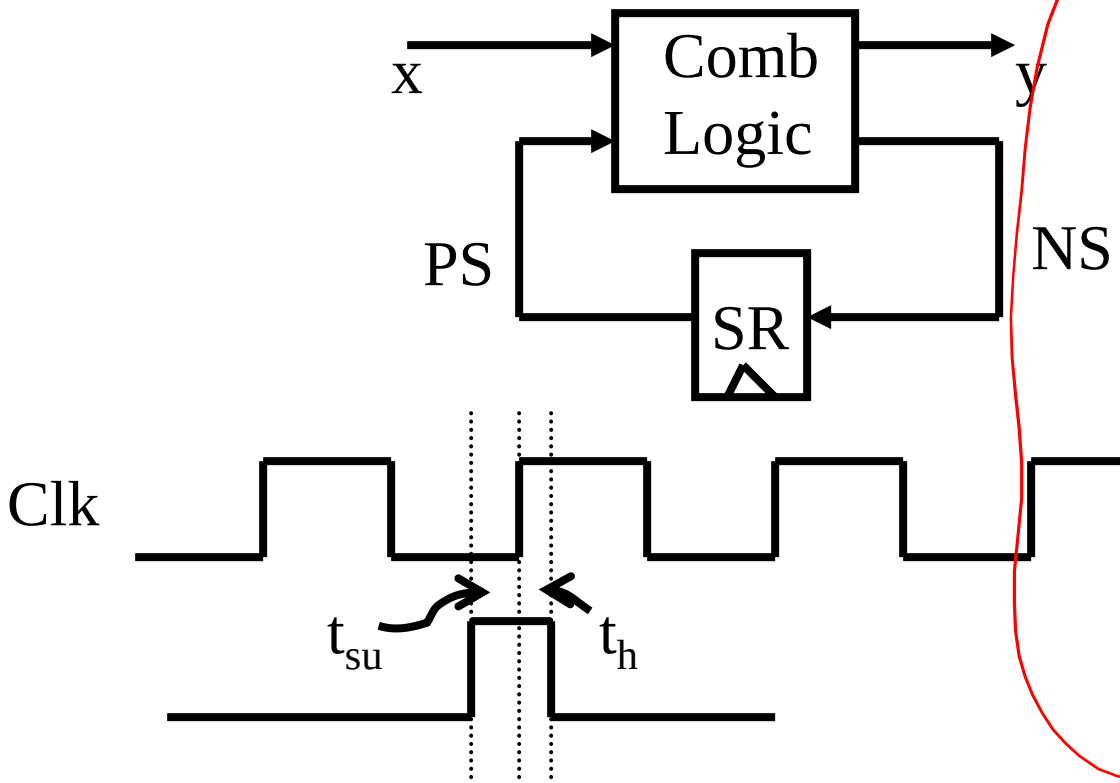
Specification of M1

y1 = 1 if <x(t-7)..x(t)> = "@" and t mod 8 = 7

y2 = 1 if t mod 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

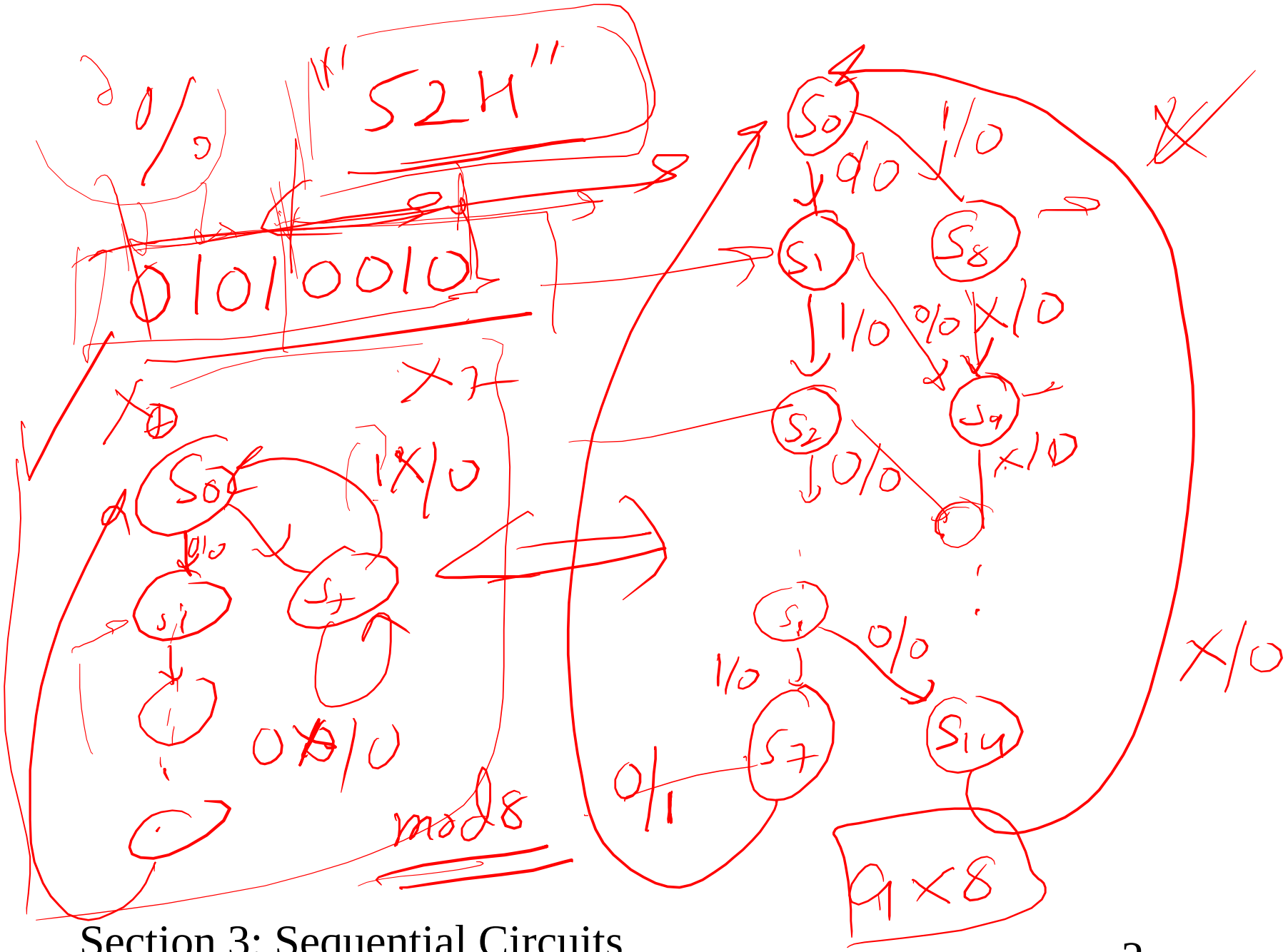
t_{su}: Setup time of the SR

t_h: Hold time of the SR

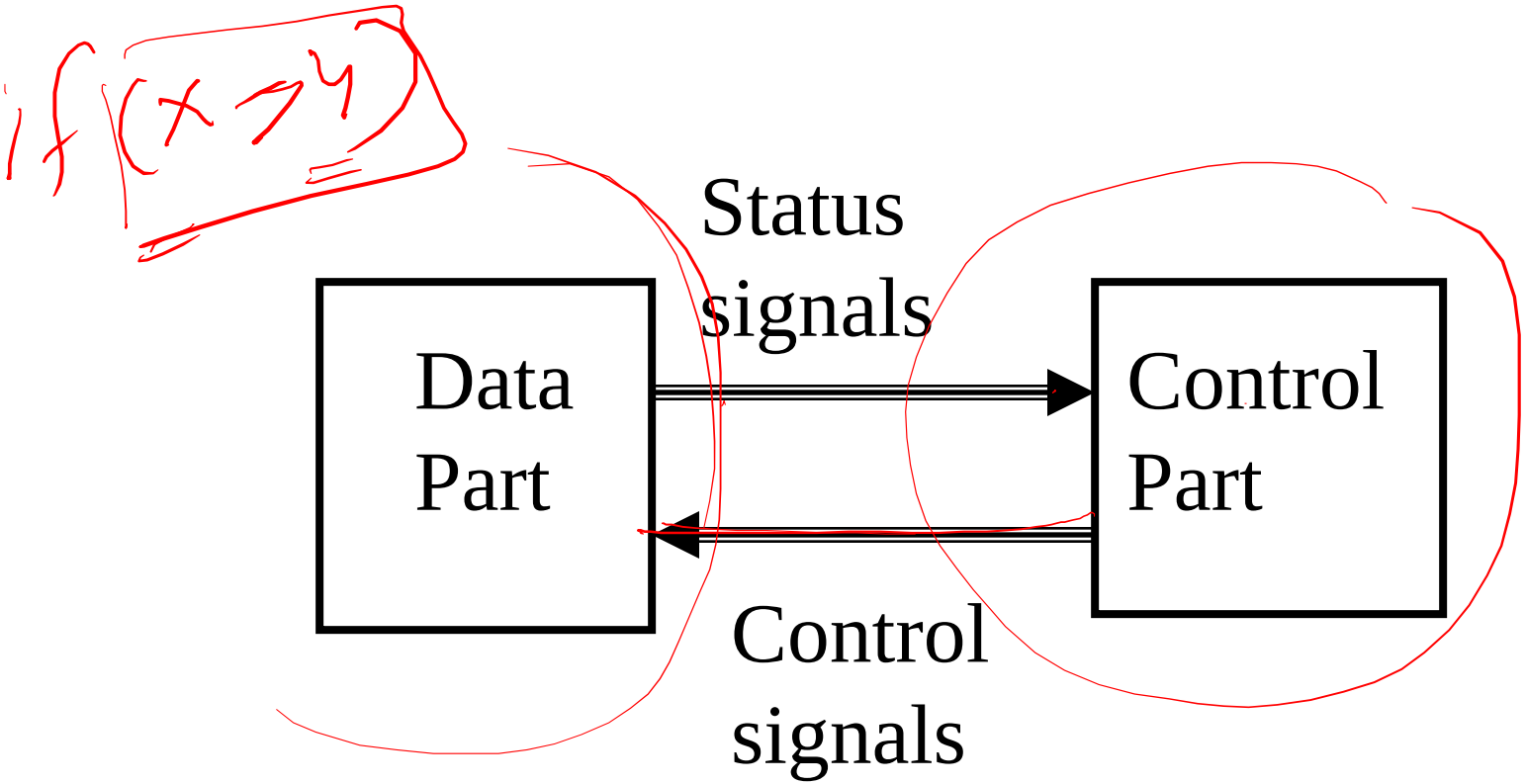
$$t_{clk} \geq \max\{ t_d + t_o, t_d + t_{ns} + t_{su} \}$$

Lecture 24
System Design Case Studies

M. Balakrishnan
Dept. of Comp. Sci. & Engg.
I.I.T. Delhi

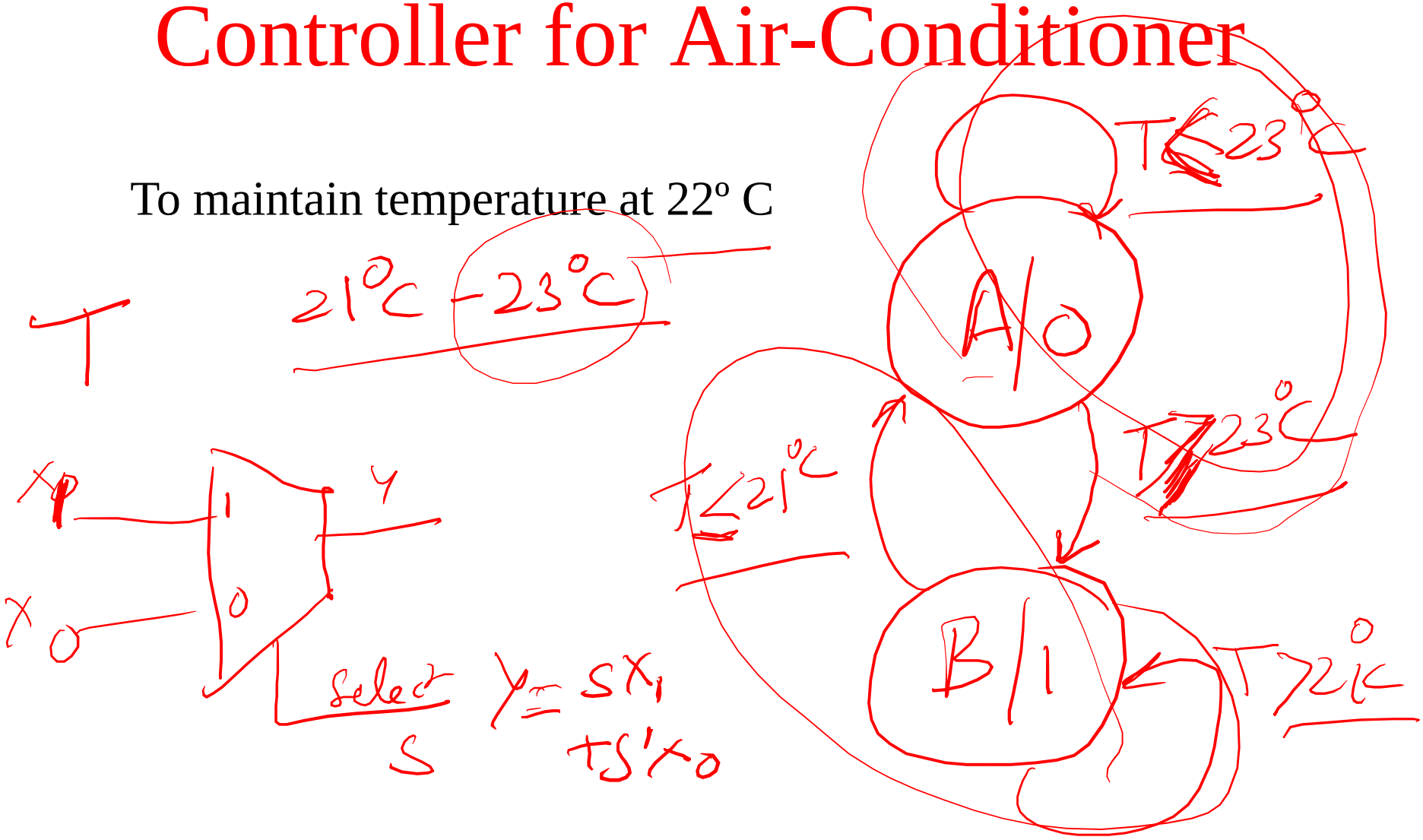

"S2H"
0/0
0101001010
X0
X7
S0
1/X/0
0/0
S1
S7
0/X/0
mod8
S0
1/0
S8
0/0
X/0
S2
S9
X/0
0/0
S1
1/0
0/0
S7
S14
0/1
X/0
9×8

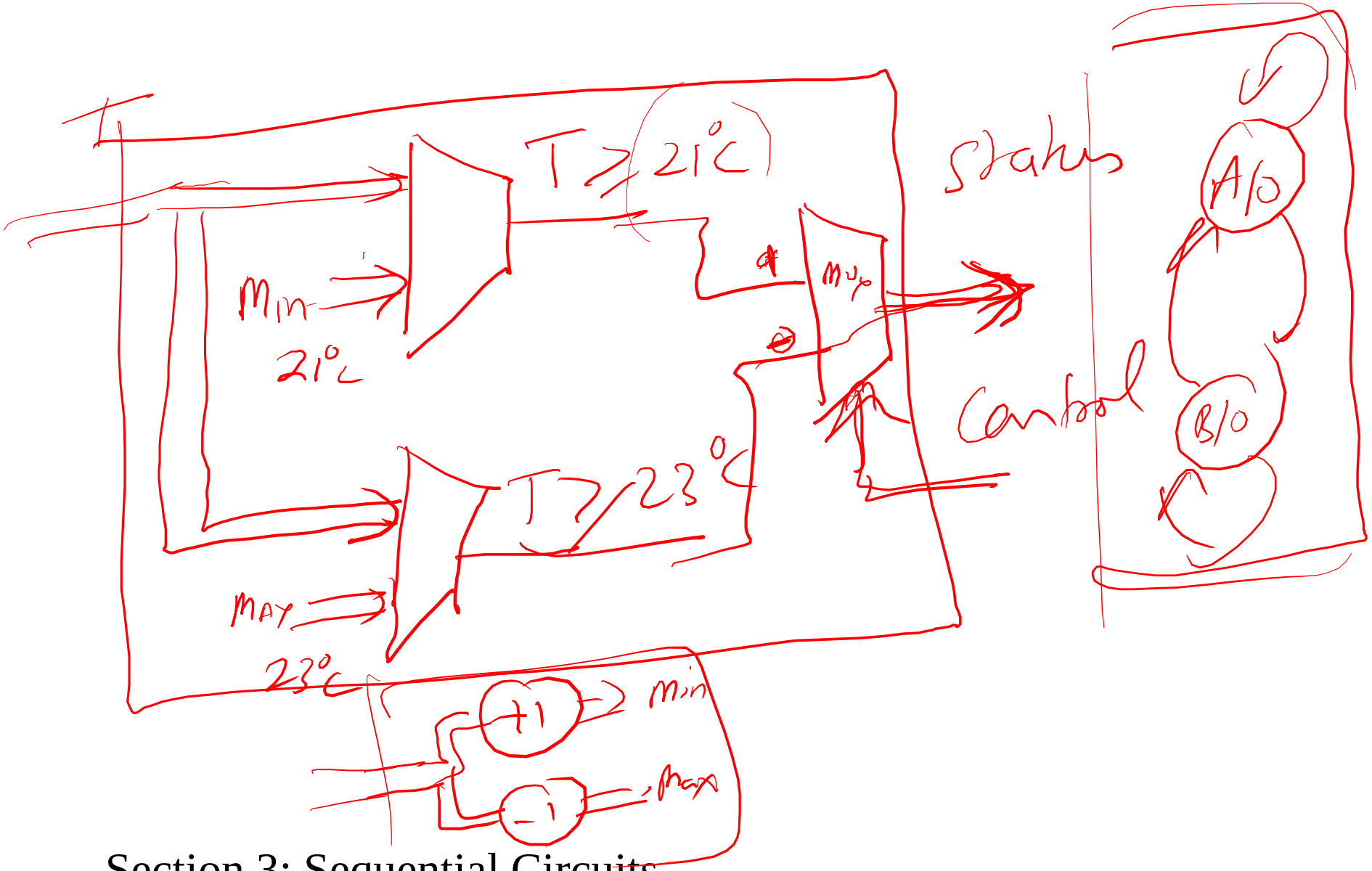
Data-Control Partition



Controller for Air-Conditioner

To maintain temperature at 22° C




T
T ≥ 21°C
Min
21°C
MUX
Status
A/D
Control
B/D
T ≥ 23°C
MAX
23°C
+1
Min
-1
Max

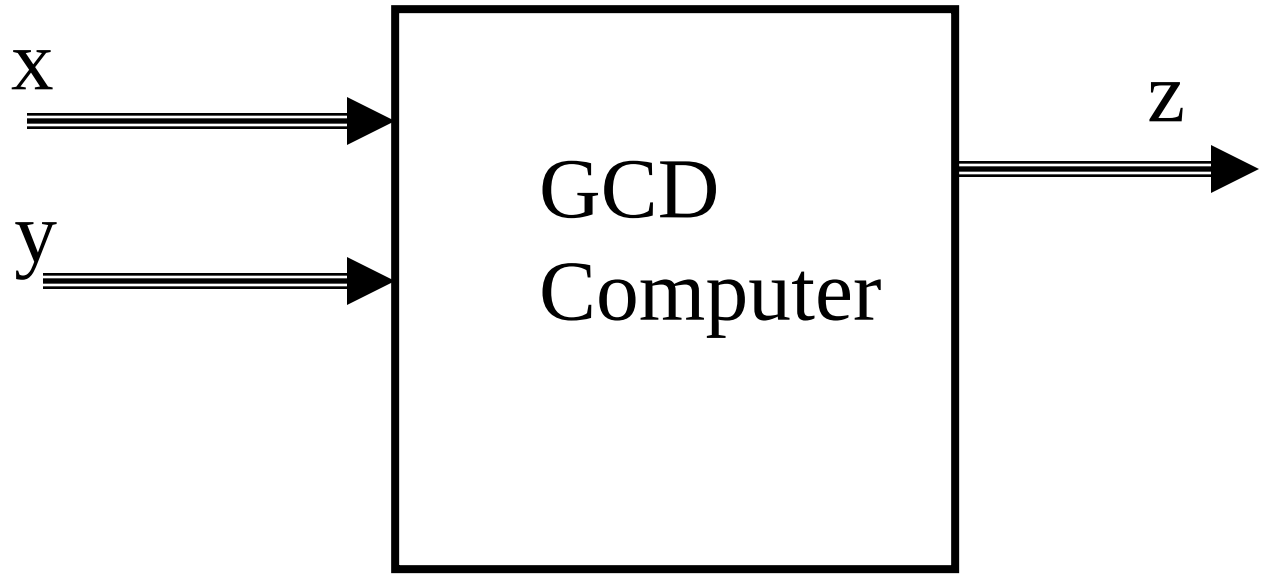

clk
E
Up/Dn
Clr
1
0
Dominates
Synchronous
level

Steps in System Design

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

Case Study1: 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.
```

Modified GCD Algorithm (RTL)

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

S1:while (R1 ≠ R2) do

S2: if (R1 > R2)

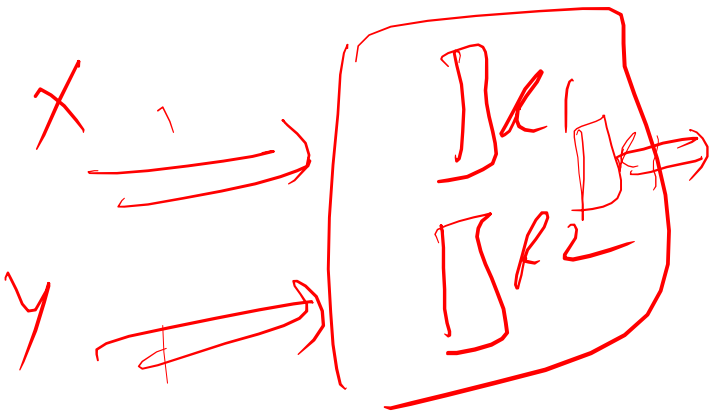
S3: then R1:= R1 - R2

S4: else R2:= R2 - R1

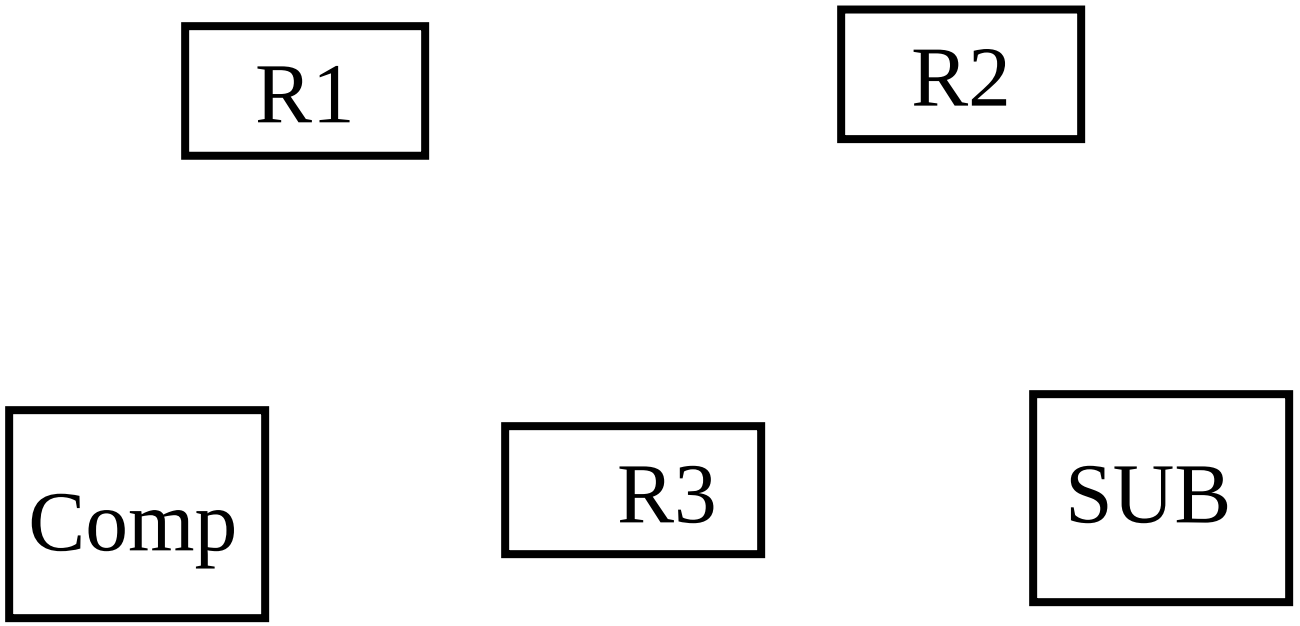
endif;

endwhile;

S5: R3:= R1;



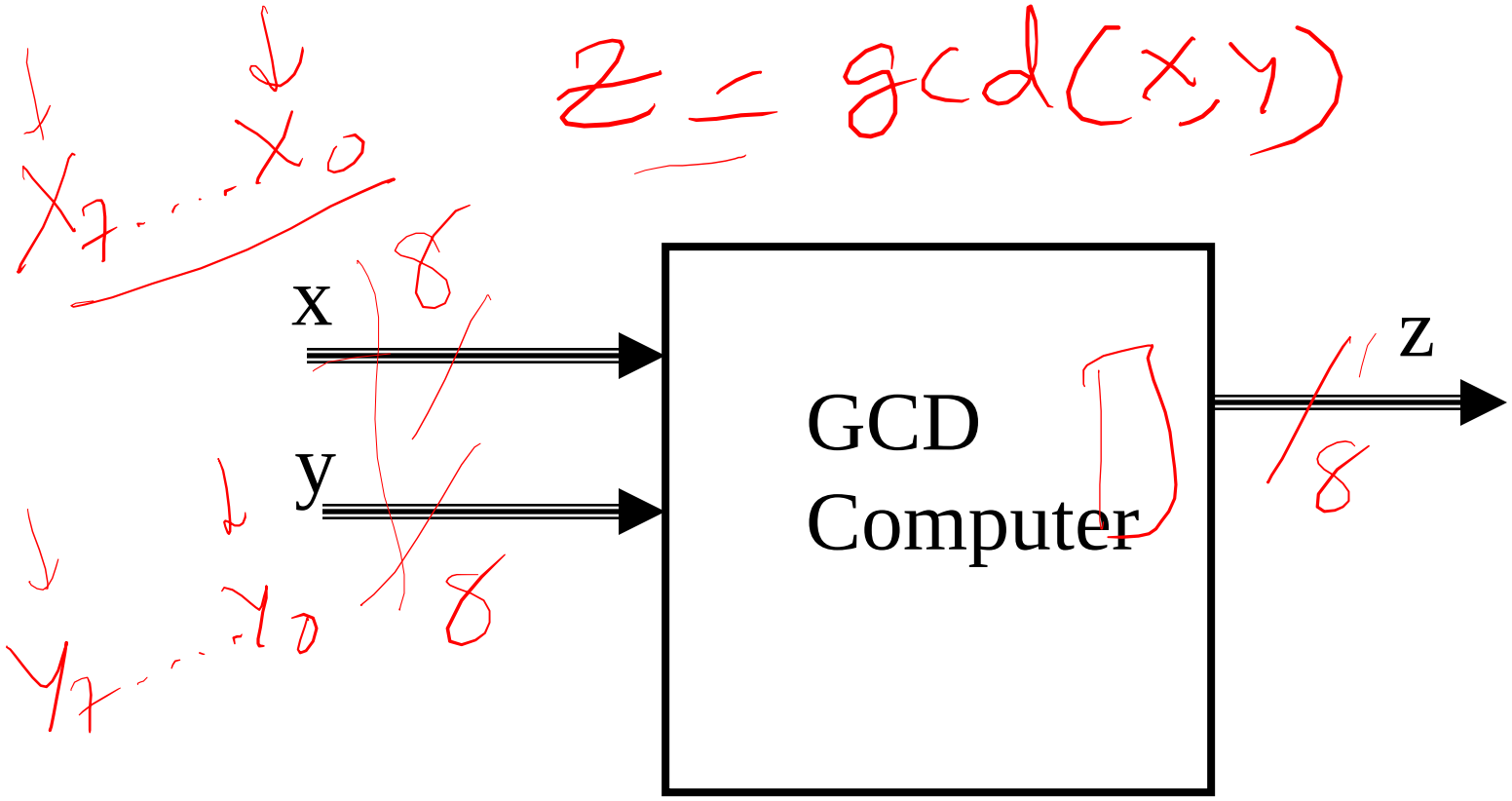
GCD Computer: Data Part



Lecture 25
System Design Case Studies

M. Balakrishnan
Dept. of Comp. Sci. & Engg.
I.I.T. Delhi

Case Study1: GCD Computer

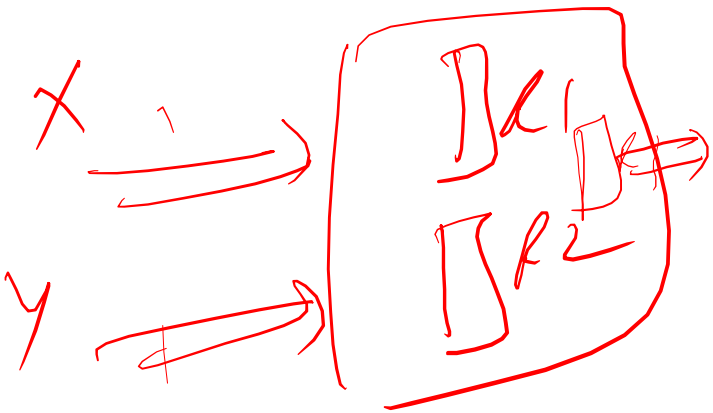


GCD Algorithm

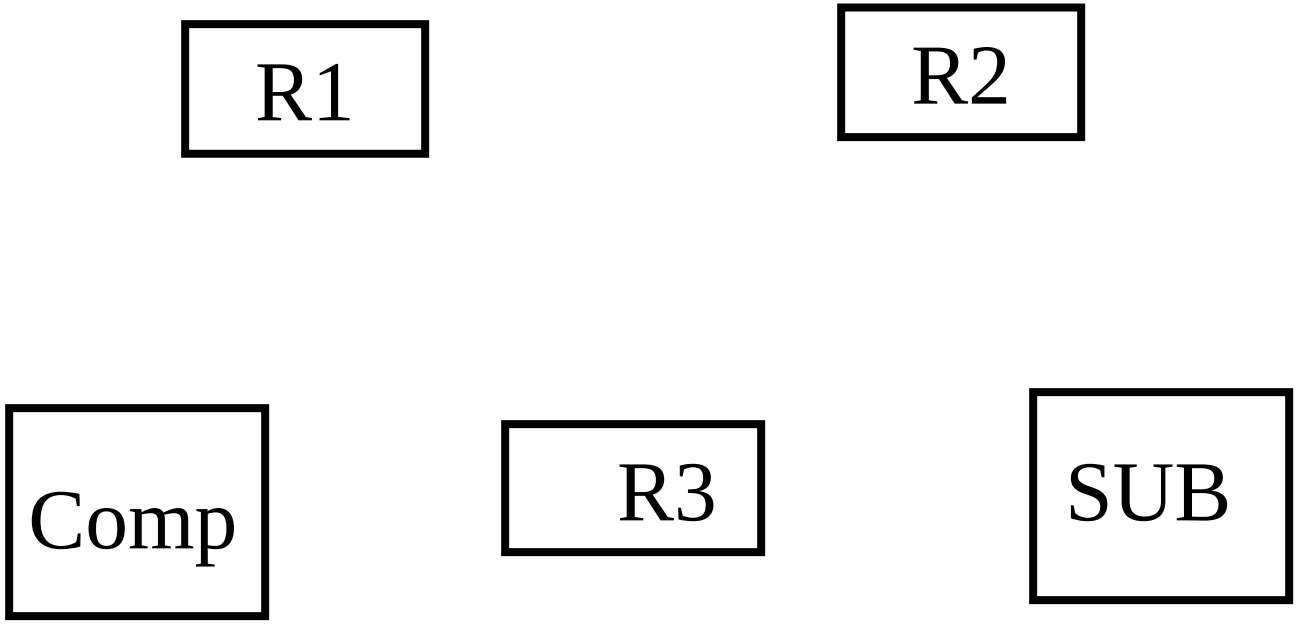
```
Input x, y;
while ( x ≠ y ) do
    if  ( x > y )     then  x := x - y
                      else  y := y - x
    endif;
endwhile;
z := x;
end.
```

Modified GCD Algorithm (RTL)

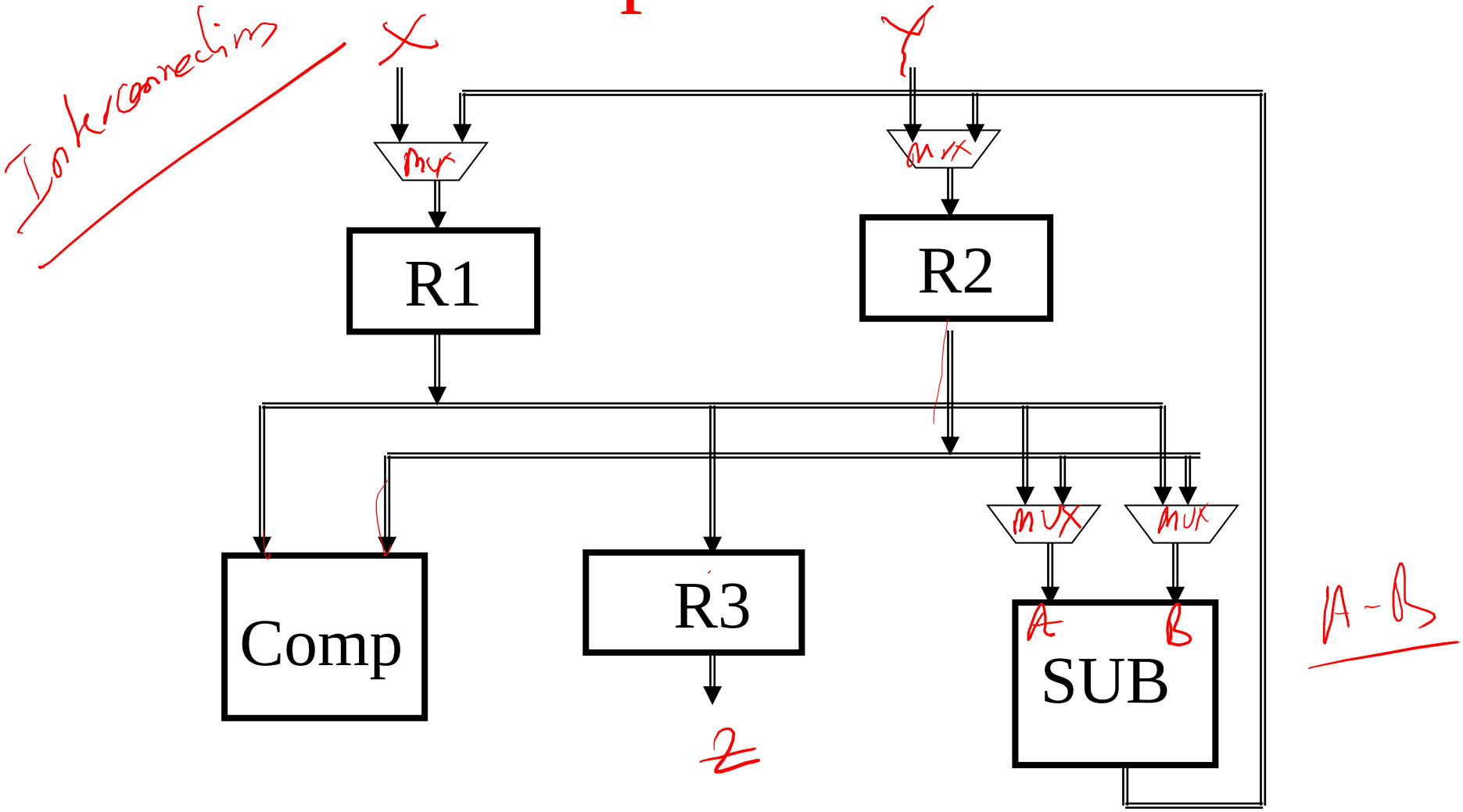
```
S0:R1:= x,  R2:= y;
S1:while ( R1 ≠ R2 ) do
   S2: if ( R1 > R2 )
   S3: then R1:= R1 - R2
   S4: else  R2:= R2 - R1
   endif;
endwhile;
S5: R3:= R1;
```



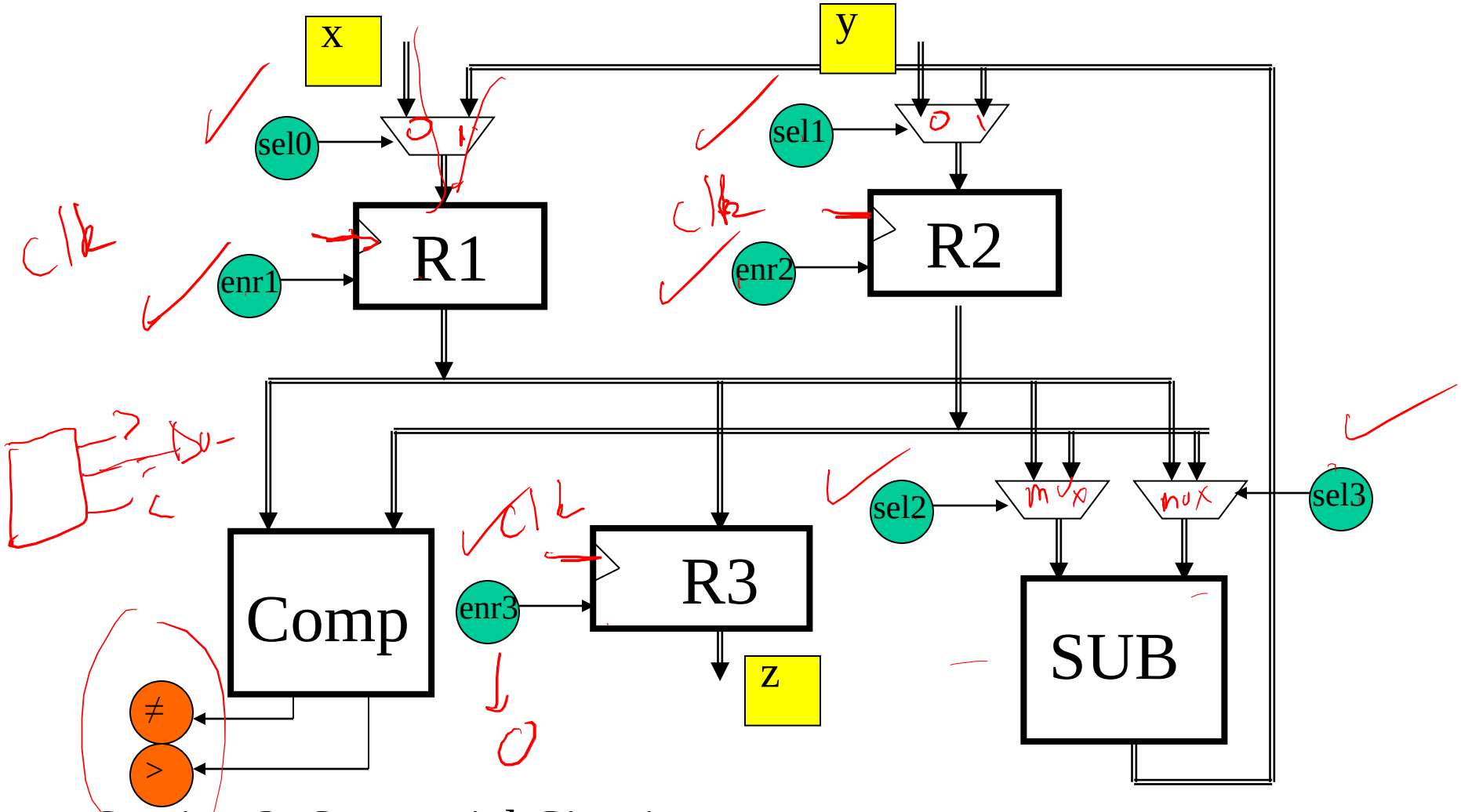
GCD Computer: Data Part



GCD Computer: Data Part



GCD Computer: Data Part



Control Part FSM

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

S1:while (R1 ≠ R2) do

S2: if (R1 > R2)

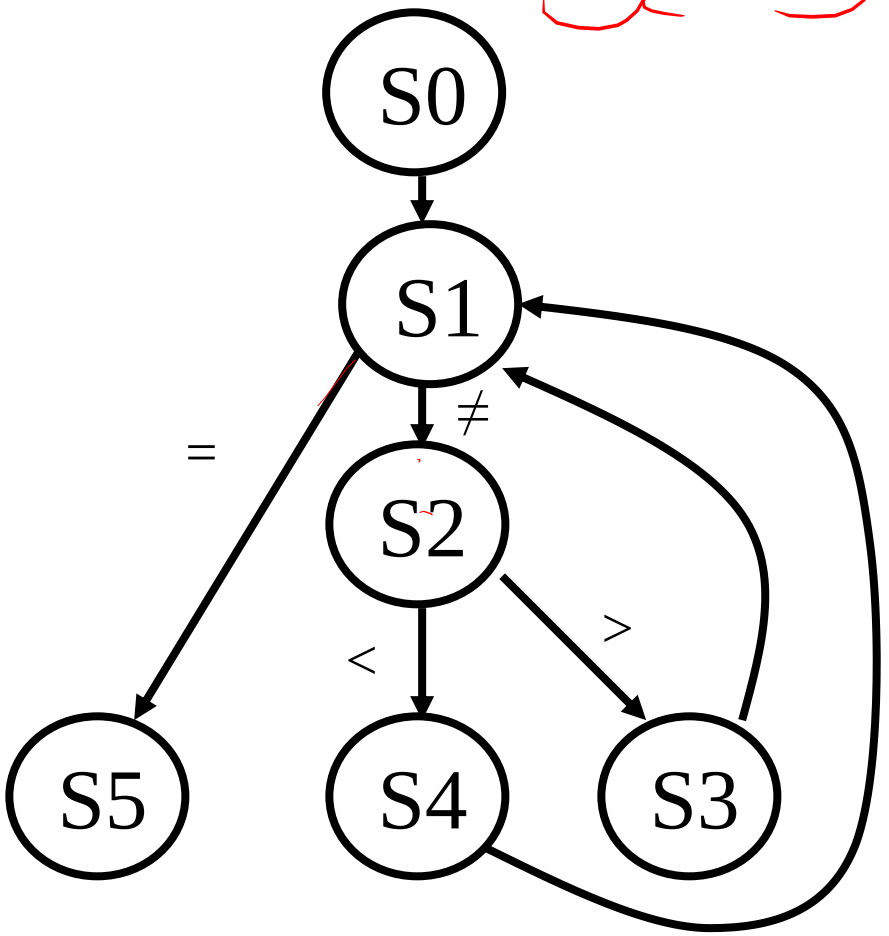
S3: then R1:= R1 - R2

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)

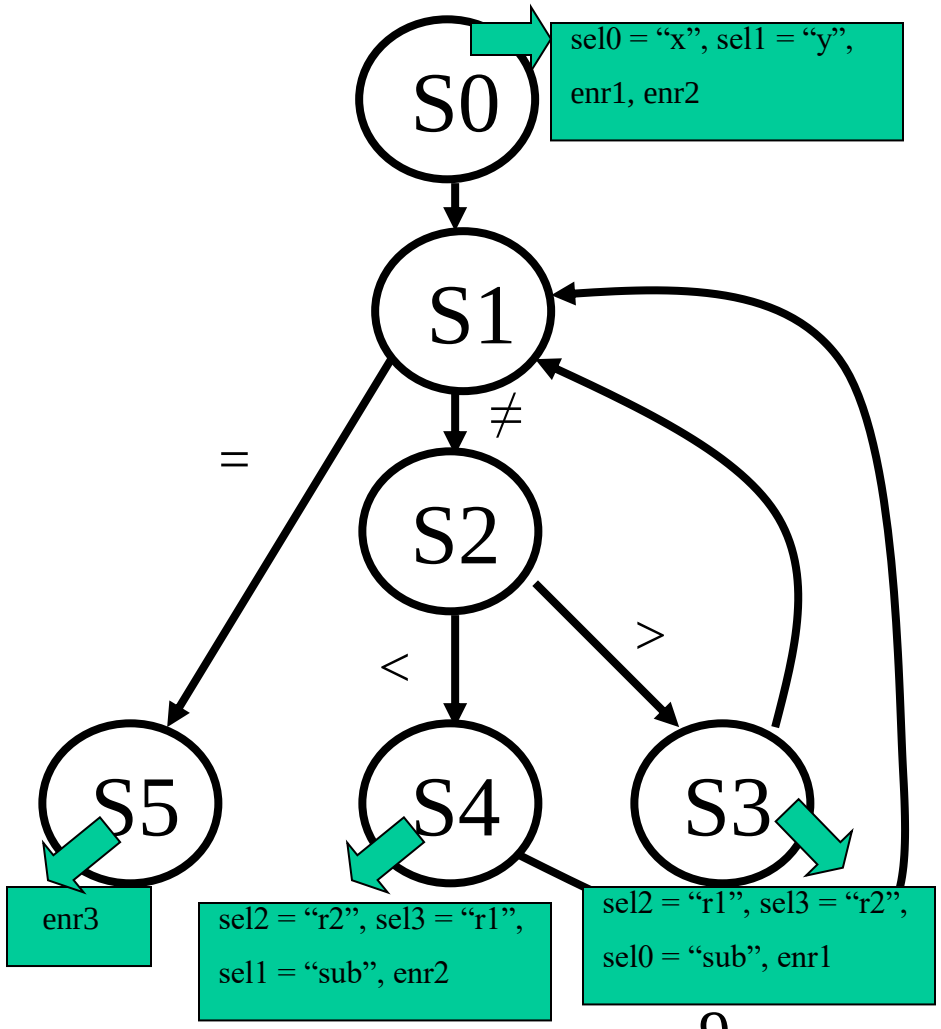
S3: then R1:= R1 - R2

S4: else R2:= R2 - R1

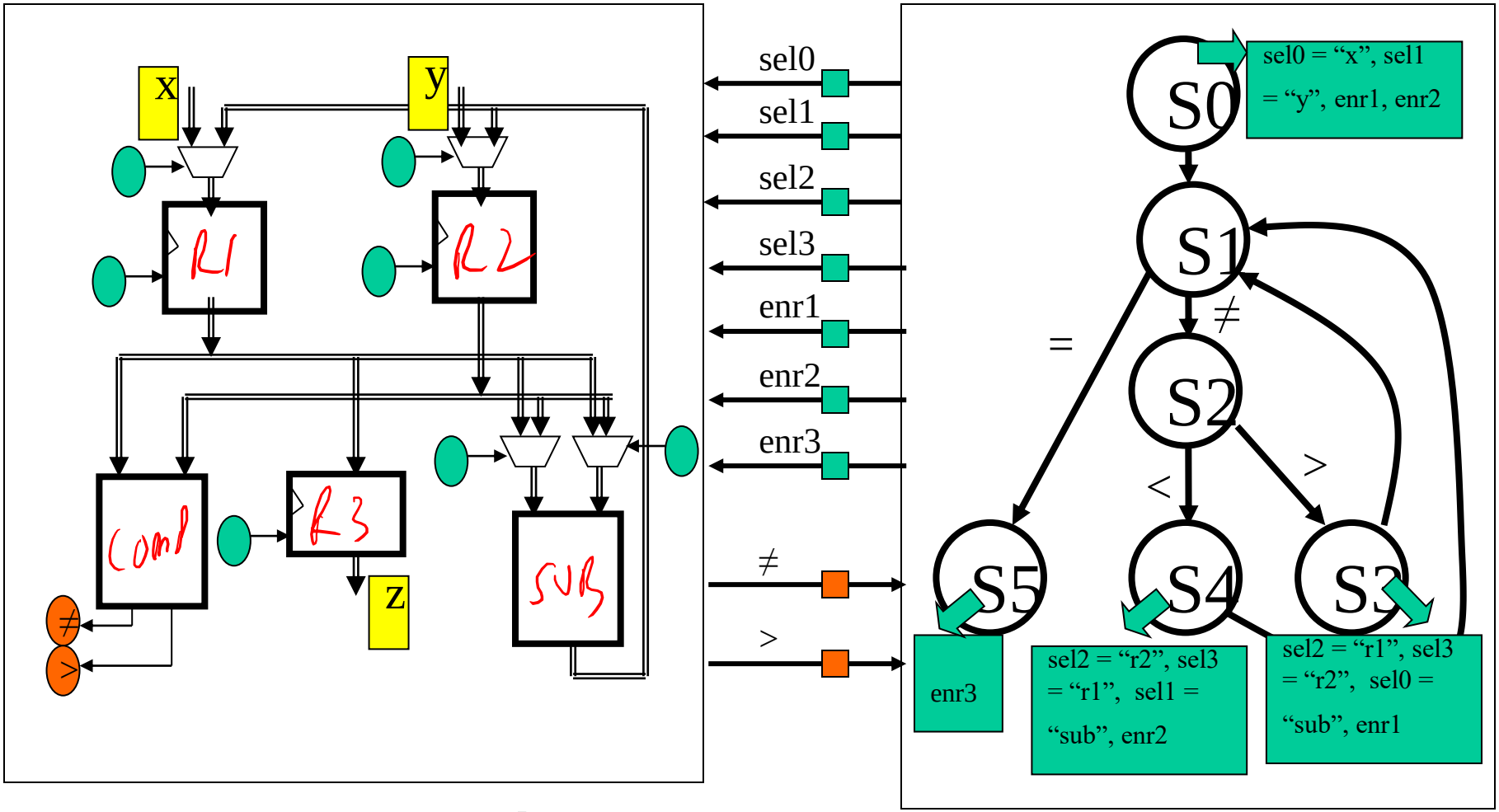
endif;

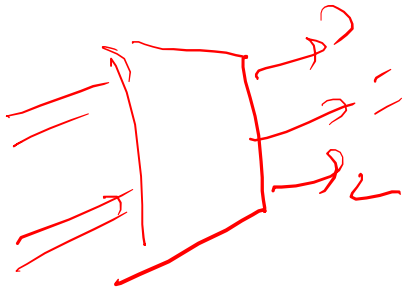
endwhile;

S5: R3:= R1;



Data-Control Interface





Modified RTL + FSM

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

S1: Case (R1 “Compare” R2)

S2: “>” R1:= R1 - R2,

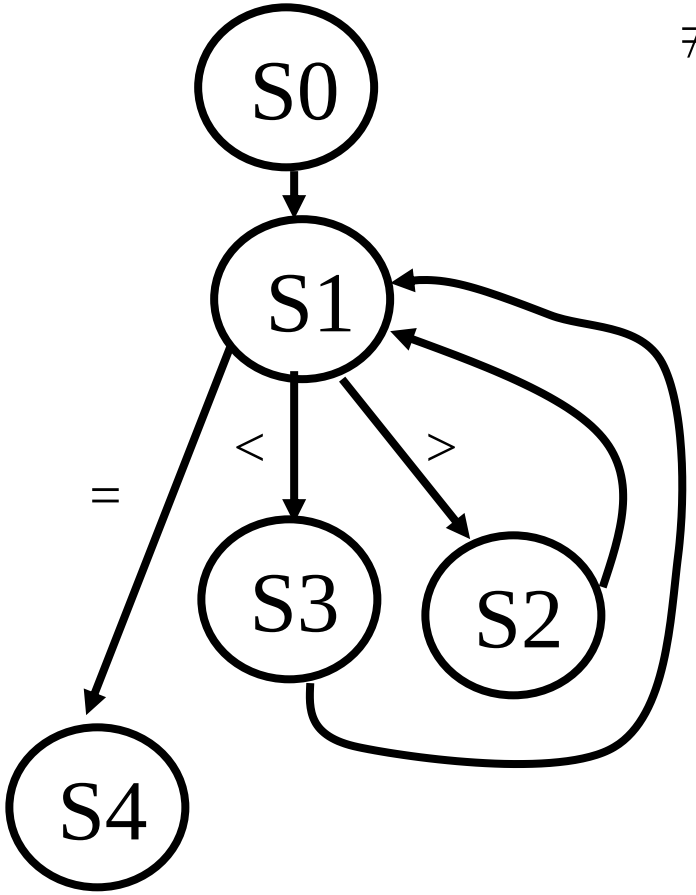
go to S1;

S3: “<“ R2:= R2 - R1,

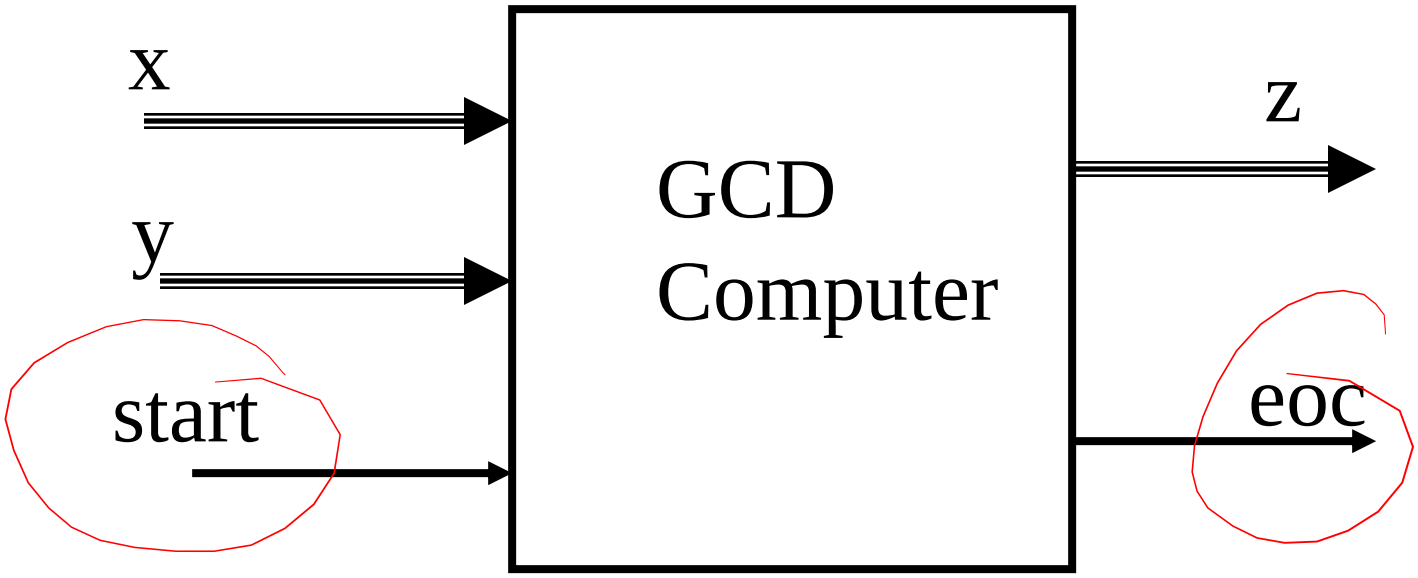
go to S1

S4: “=“ R3:= R1;

endcase;



GCD Computer: Interface



Modified FSM

Si: wait for “start”

S0: R1:= x, R2:= y, eoc := “0”;

S1:while (R1 ≠ R2) do

S2: if (R1 > R2)

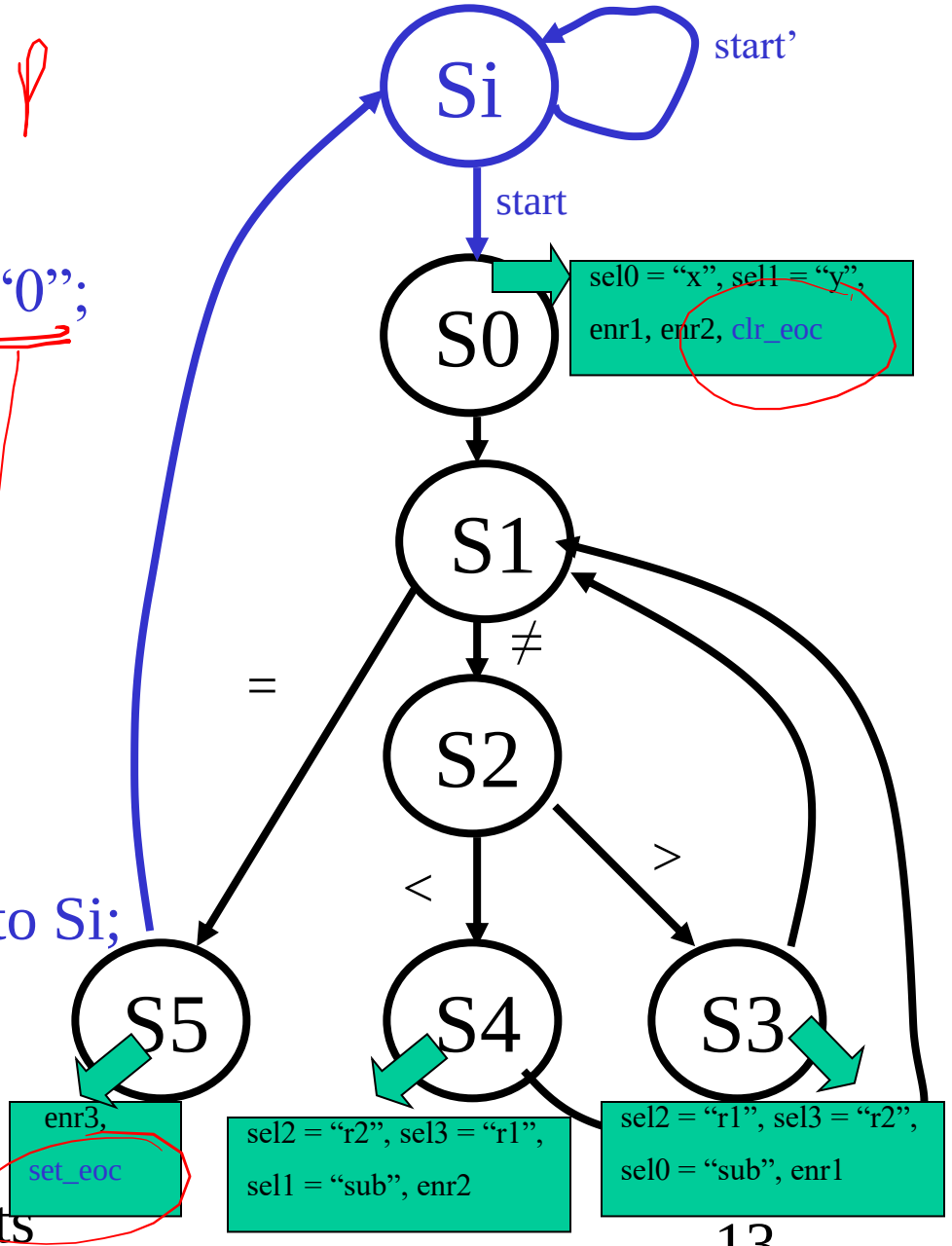
S3: then R1:= R1 - R2

S4: else R2:= R2 - R1

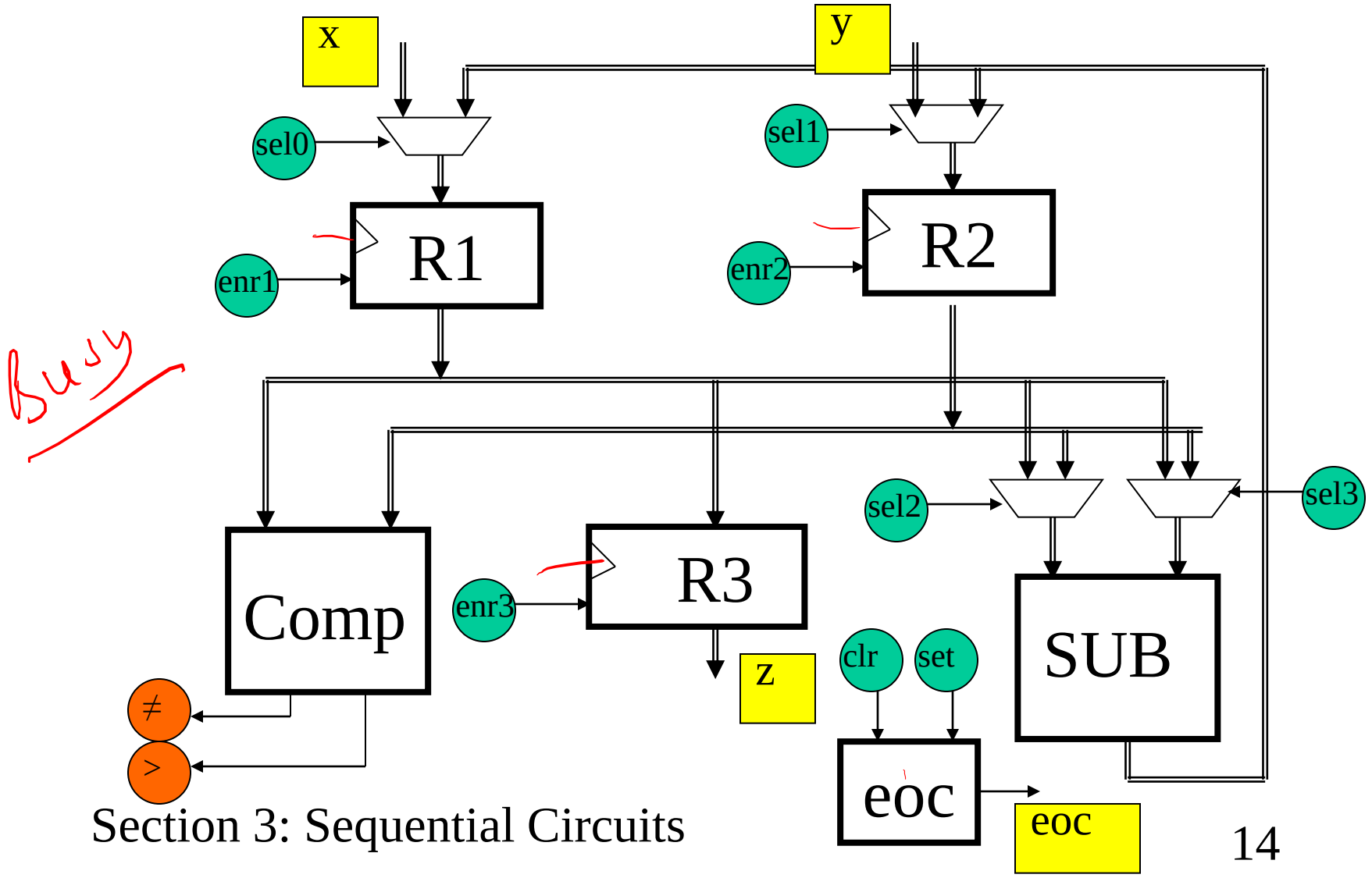
endif;

endwhile;

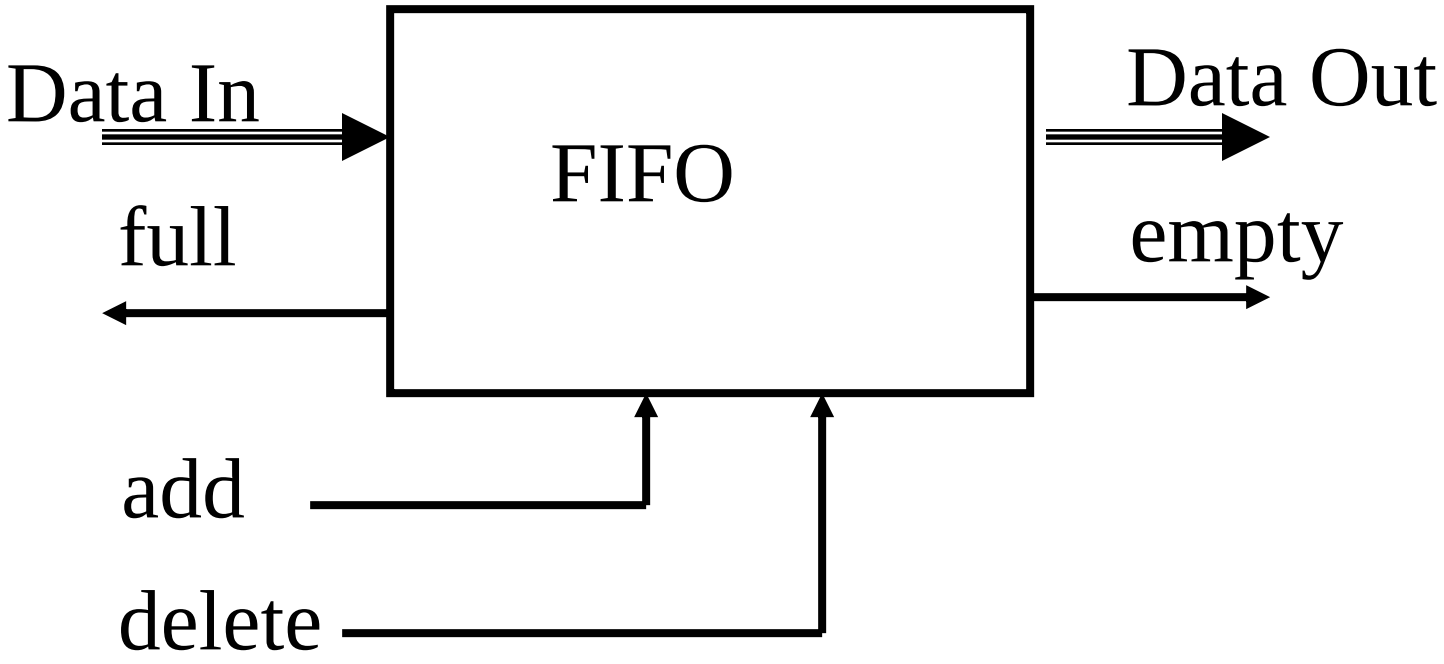
S5: R3:= R1, eoc := “1”, goto Si;



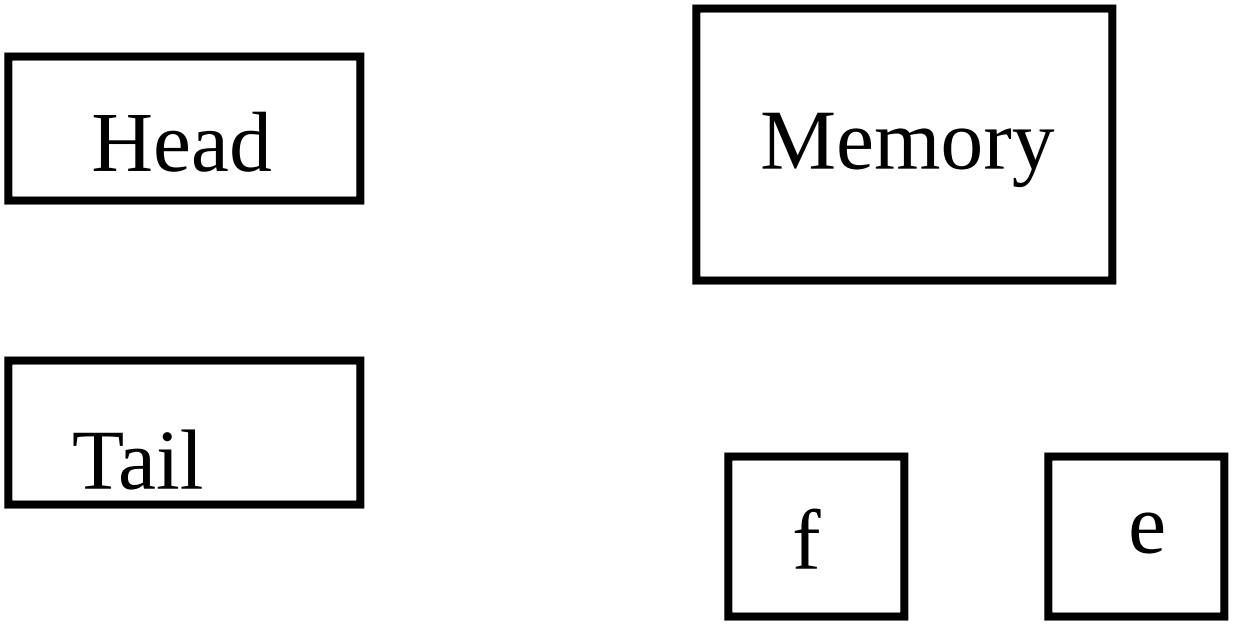
GCD Computer: Data Part



Case Study 2: FIFO



FIFO: Data Part



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.

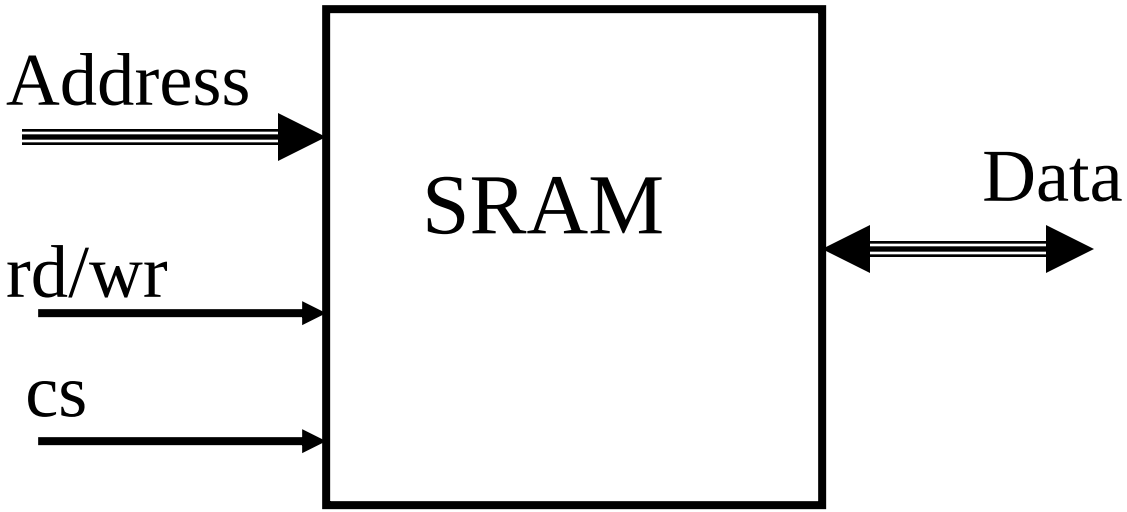
Lecture 26
Designing with Memories

M. Balakrishnan
Dept. of Comp. Sci. & Engg.
I.I.T. Delhi

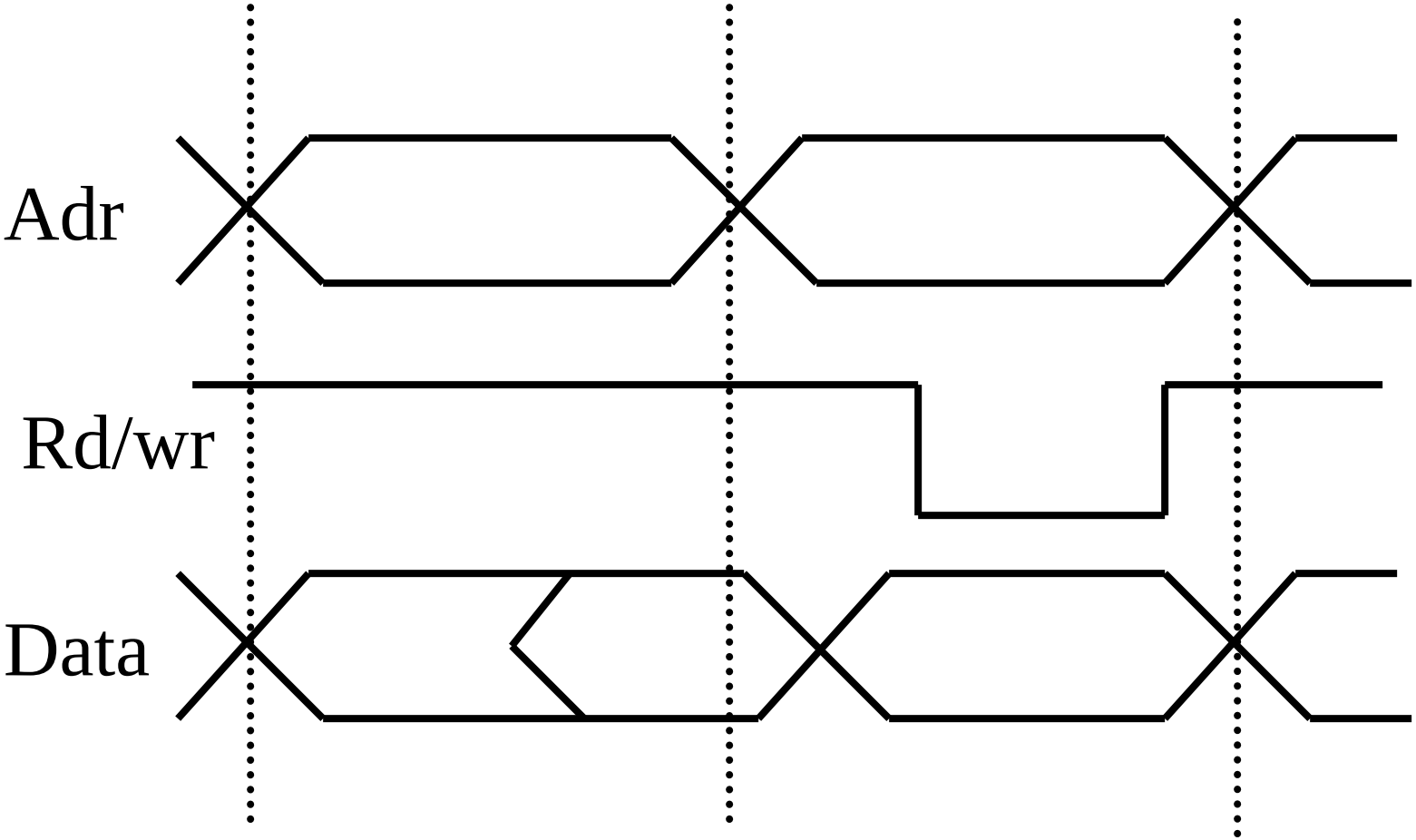
Classification of Memory Devices

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

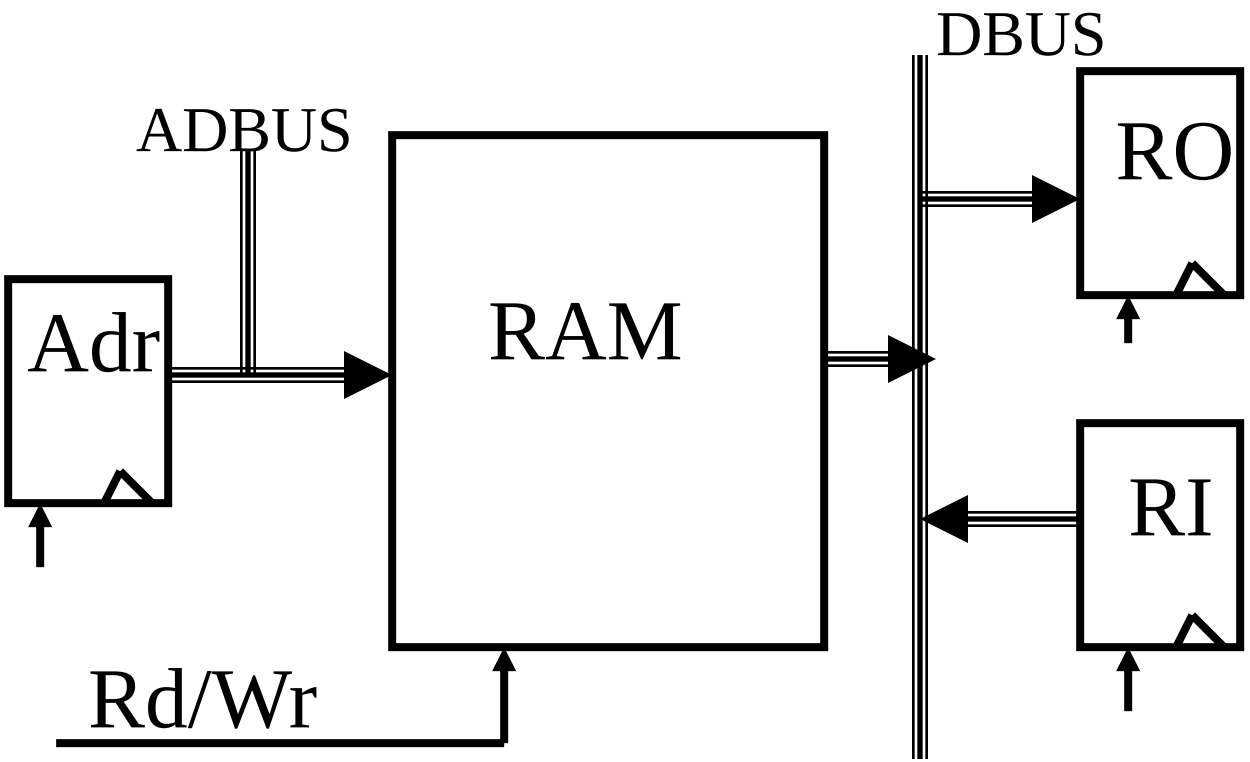
SRAM Device Signals



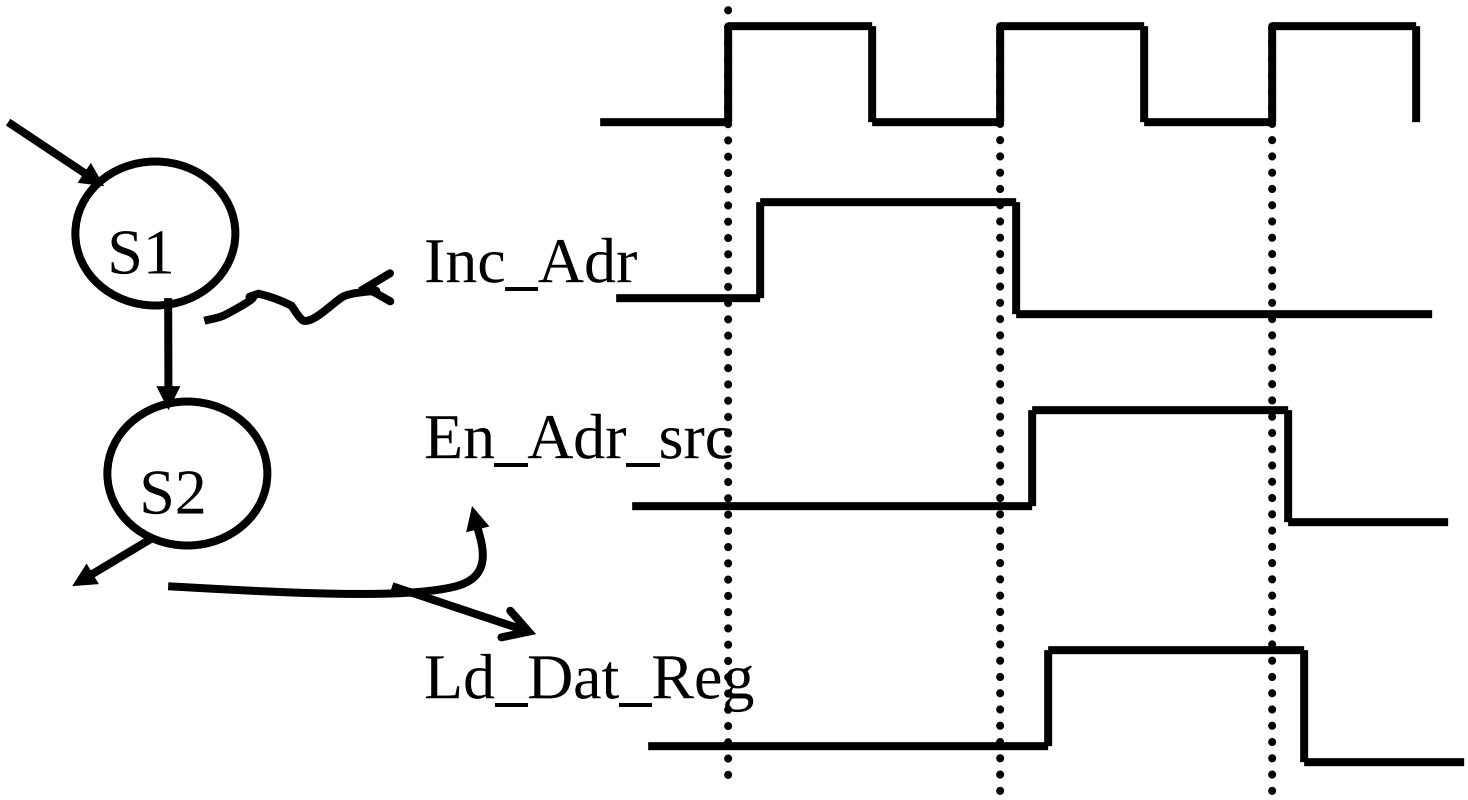
SRAM Timing



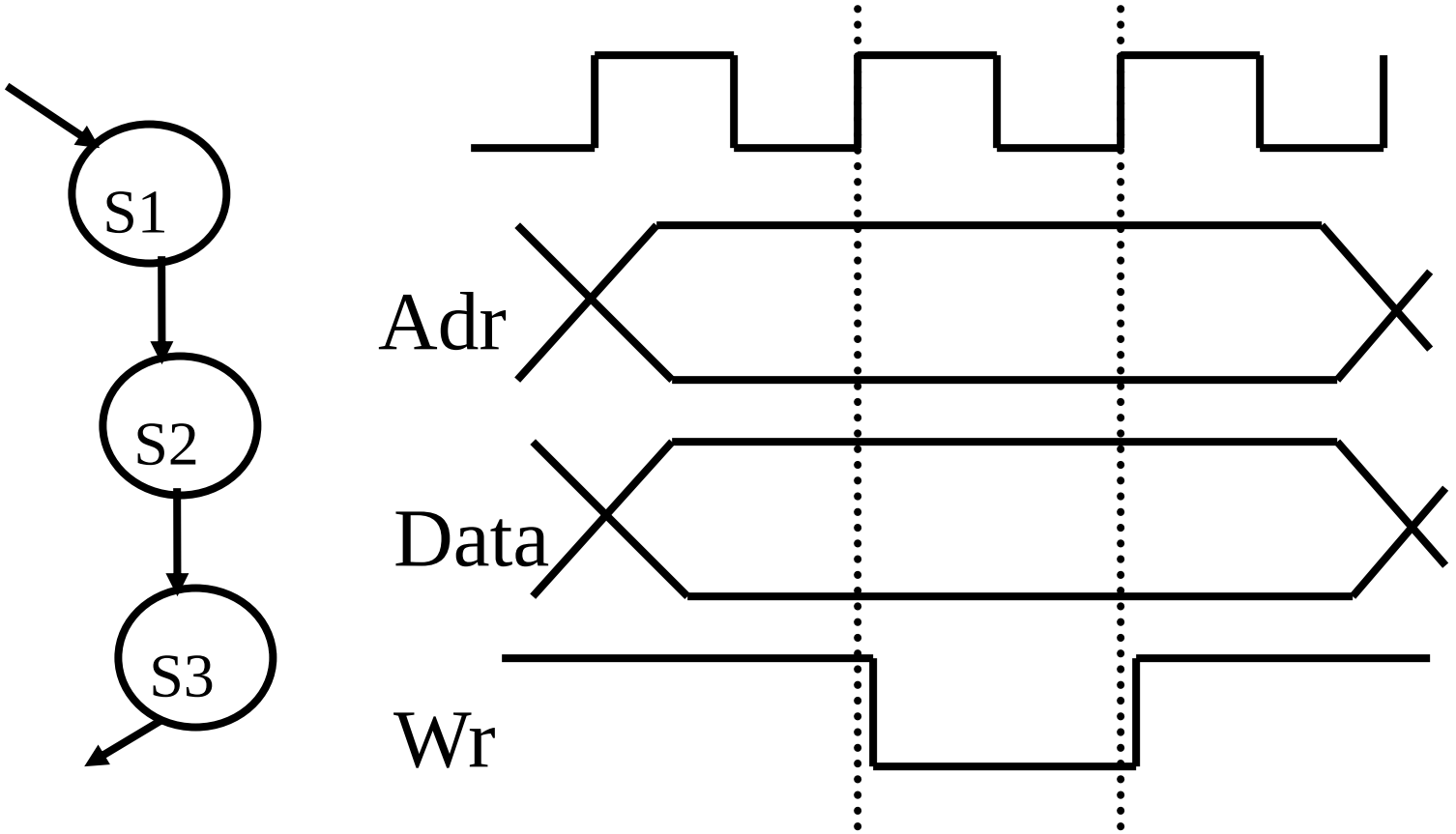
Circuit Example using Memory

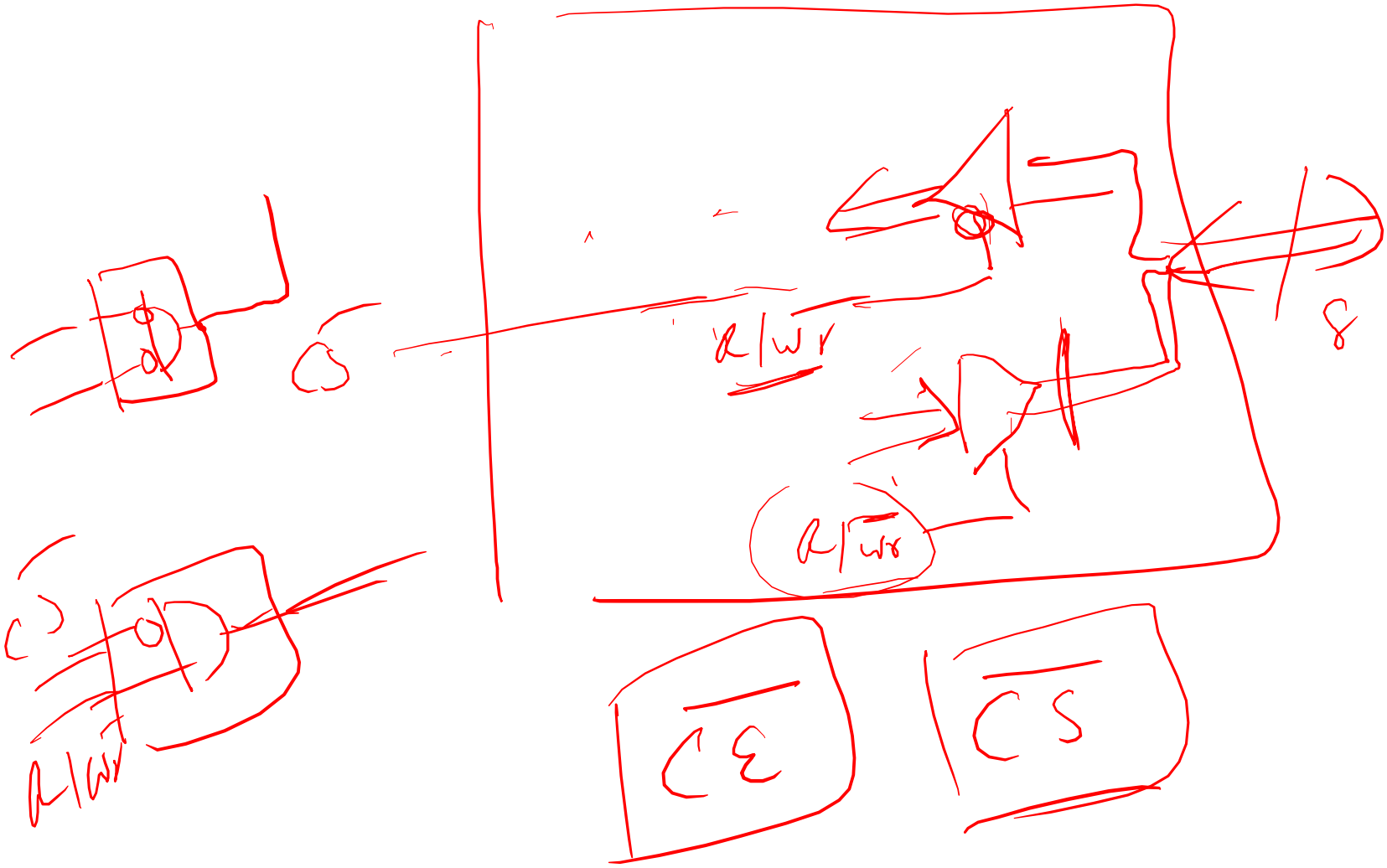


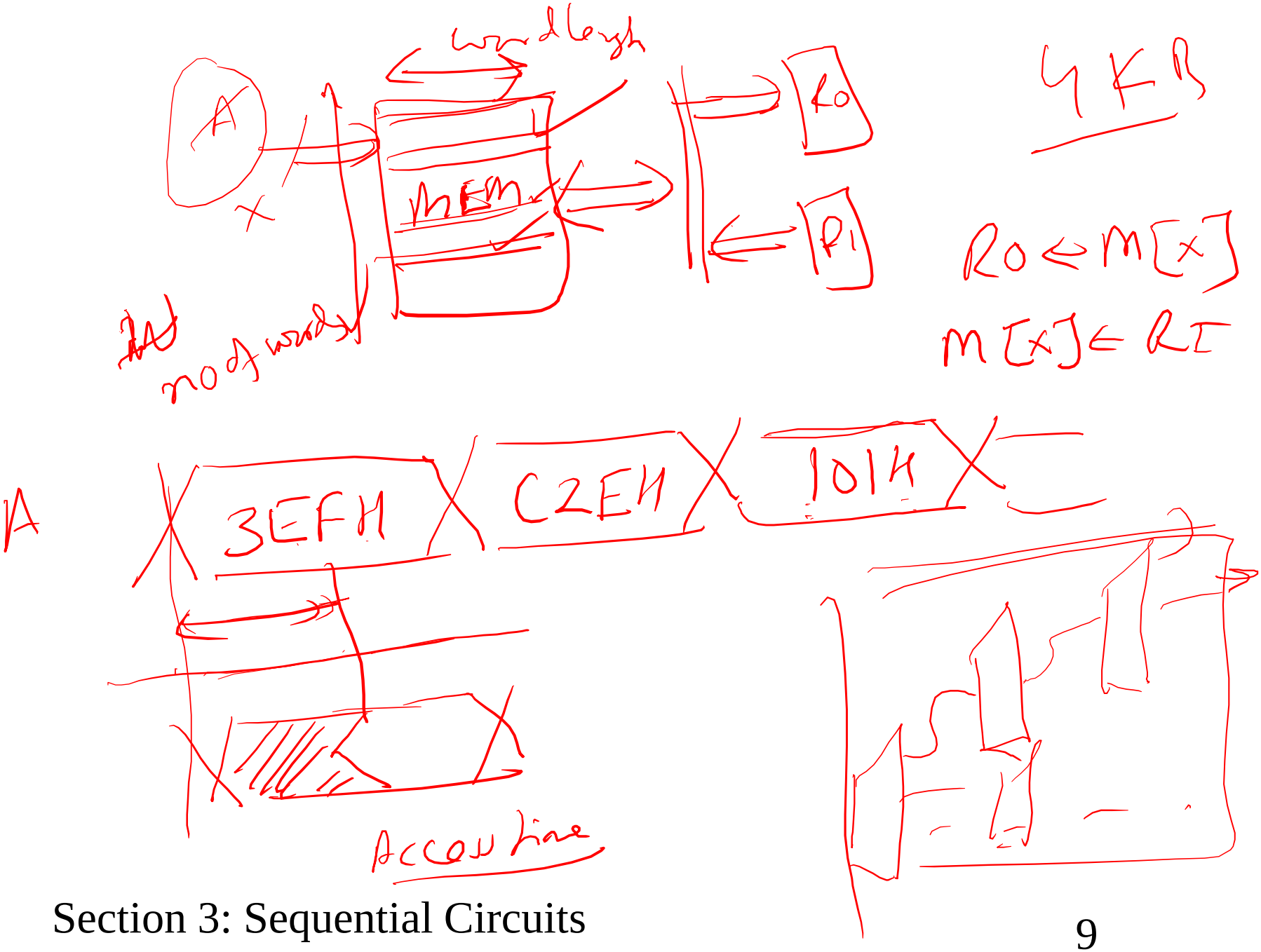
Reading Memory in a SM

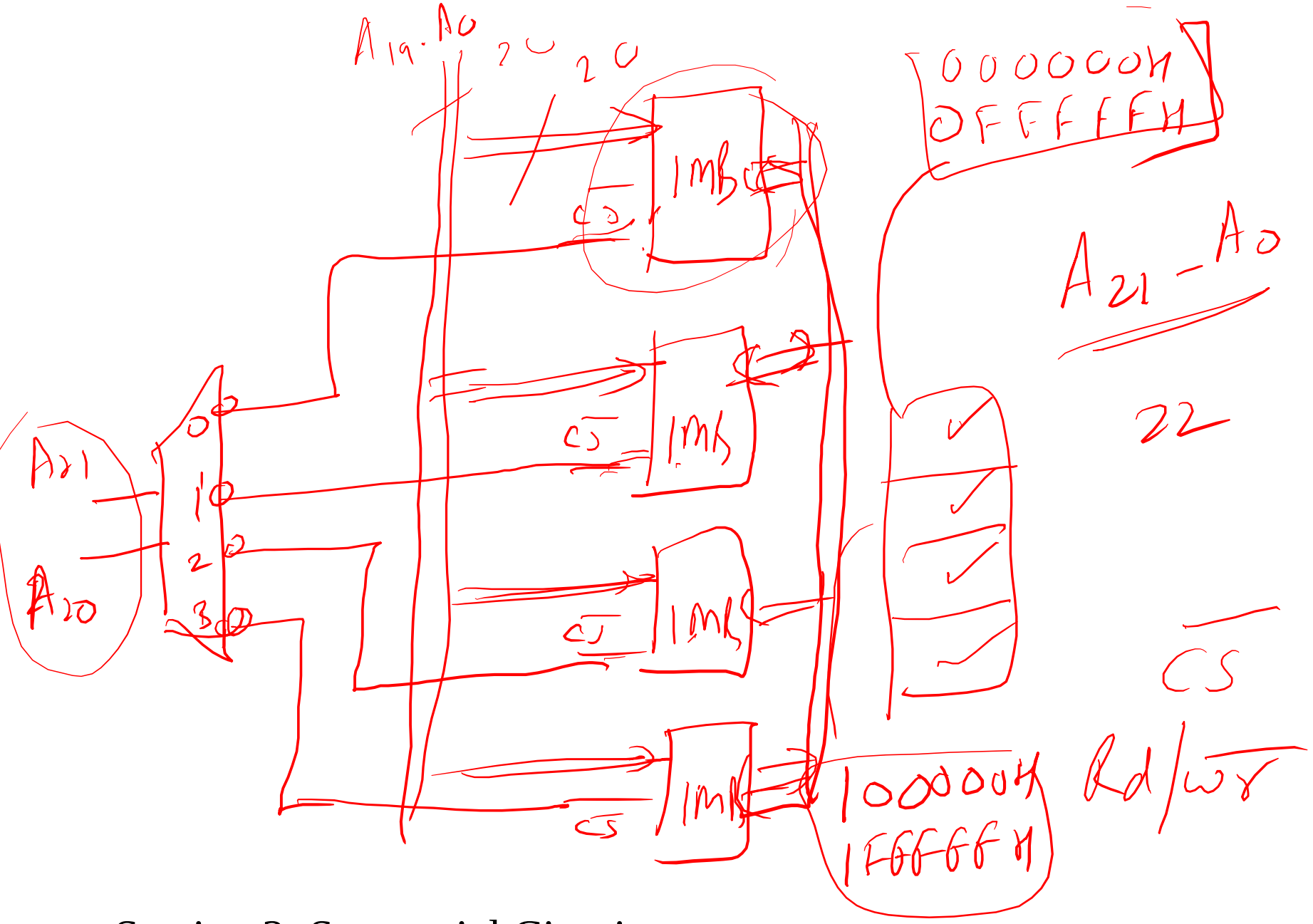


Writing Memory in a SM

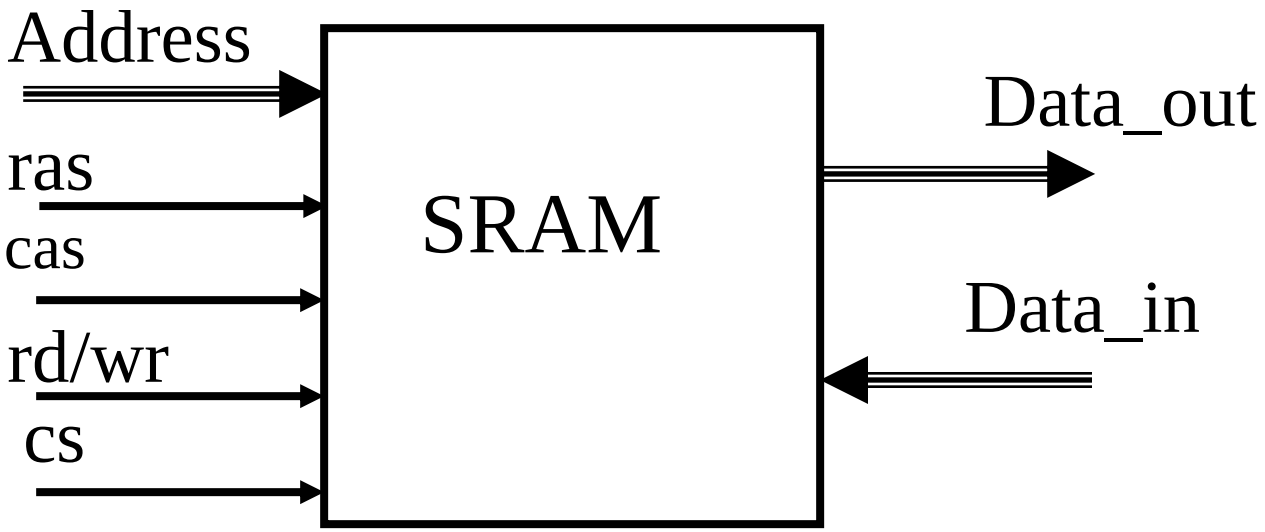



D
CS
R/Wr
8
D
R/Wr
R/Wr
CE
CS

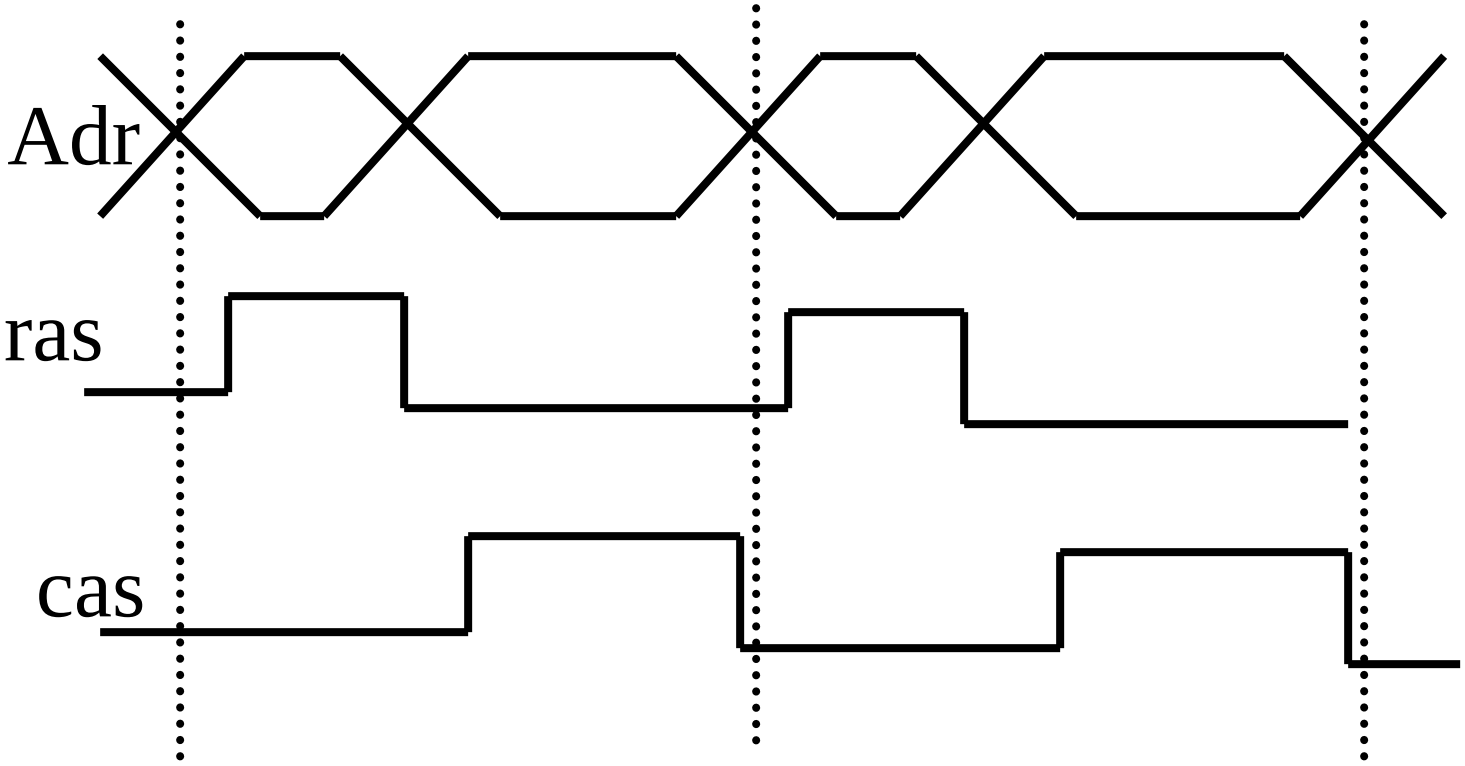

word length
A
x
MEM
R0
R1
4 KB
R0 ← M[x]
M[x] ← R1
no of words
3EFH
C2EH
101H
Access line


A19 - A0
20
20
1MB
CS
1MB
CS
1MB
CS
1MB
CS
A21
A20
0
1
2
3
00000OH
OFFFFFH
A21 - A0
22
CS
100000H
1FFFFFH
Rd/Wr

Dynamic RAM Device Signals



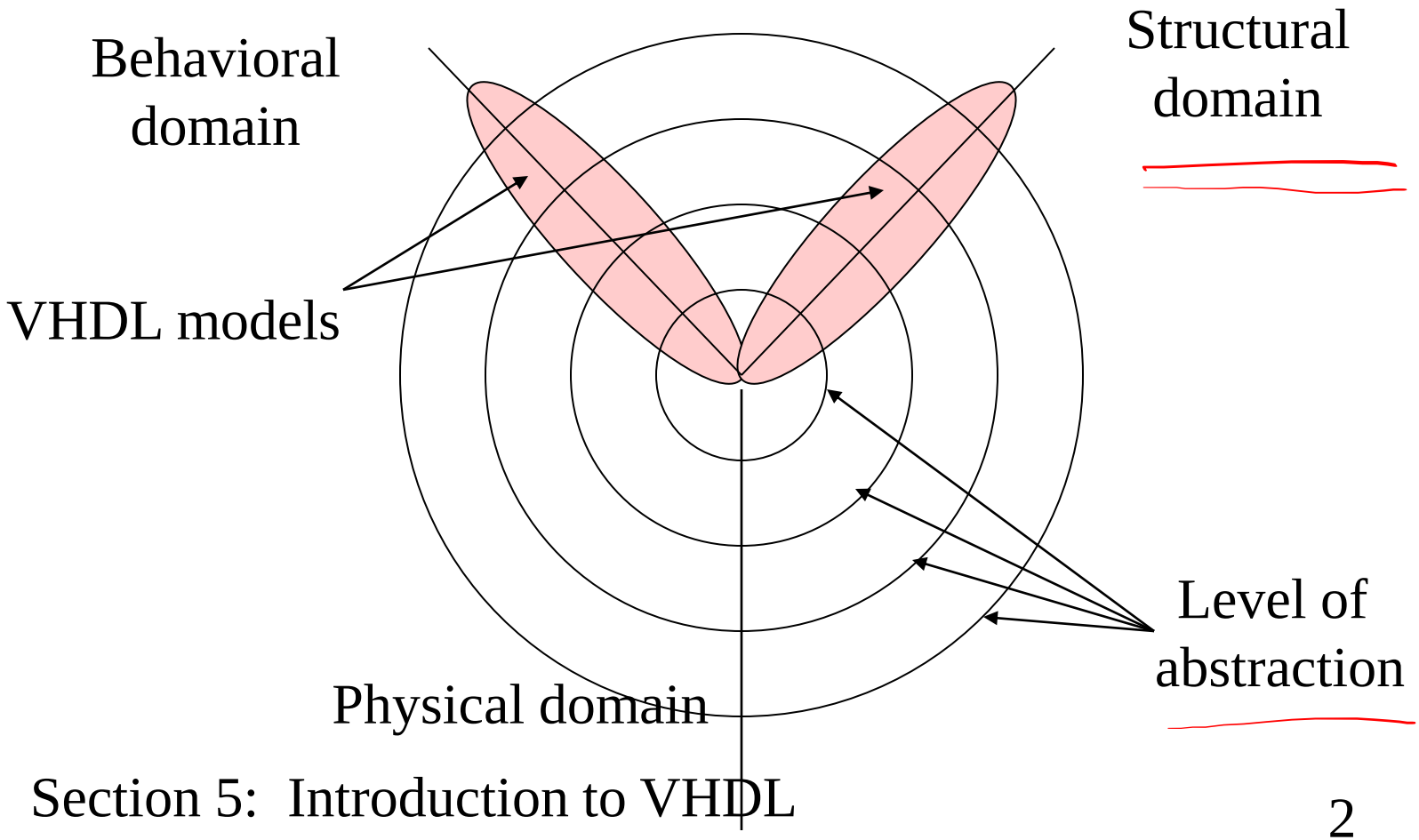
DRAM Timing



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

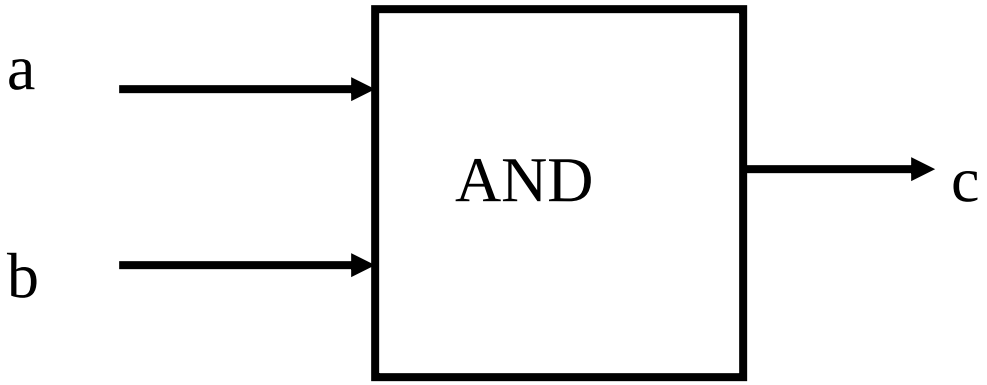
VHDL supports description of components as well as systems at various levels of abstraction

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

Modularity

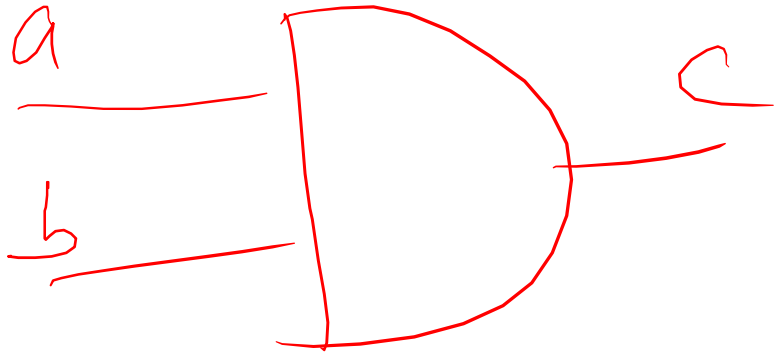
- Every component in VHDL is referred to as an entity and has a clear interface
- The interface is called an entity declaration
- The "internals" of the component are referred to as the architecture declaration
- There can be multiple architectures at even different levels of abstraction associated with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



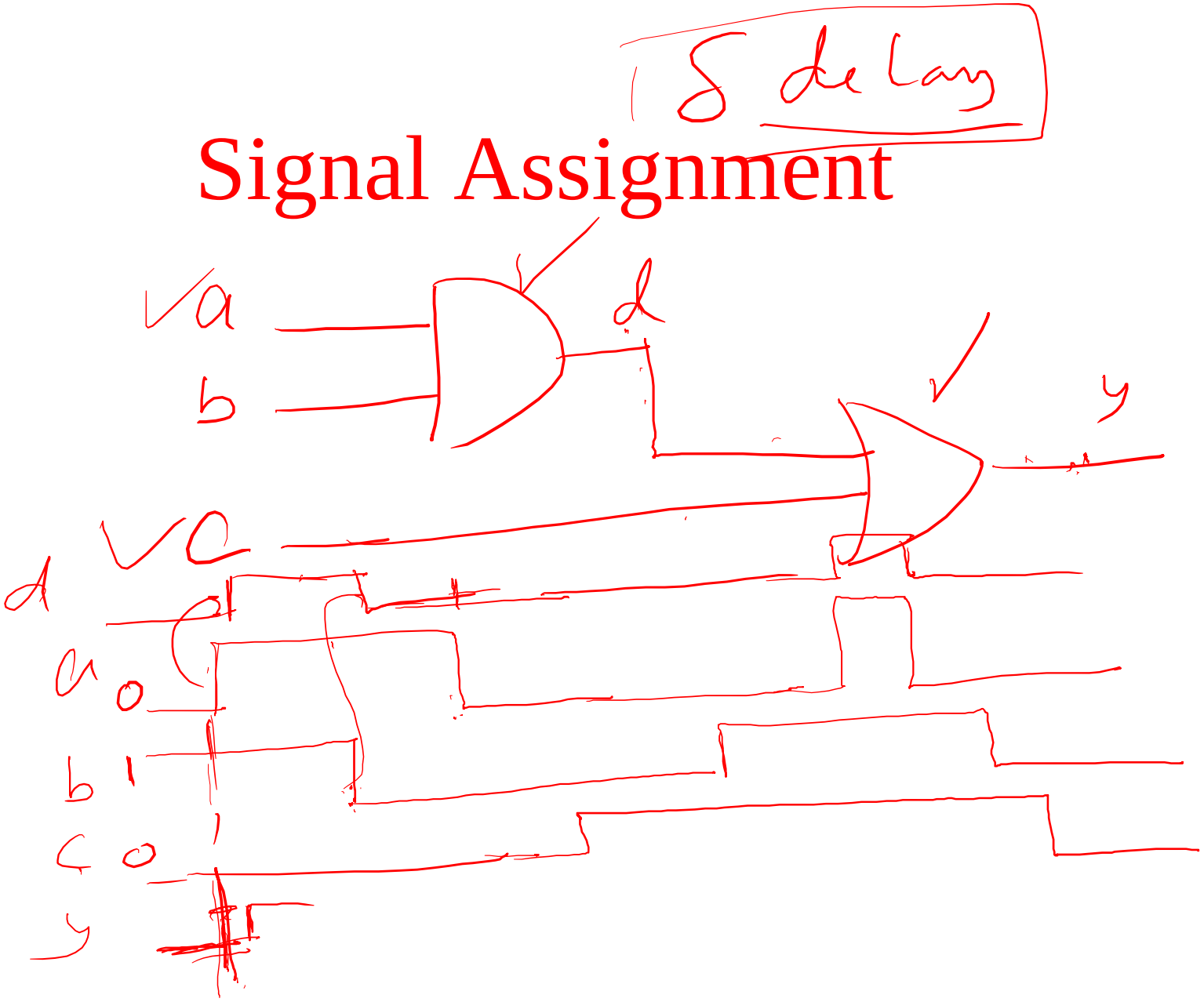
after 1ns;

Concurrent assignment

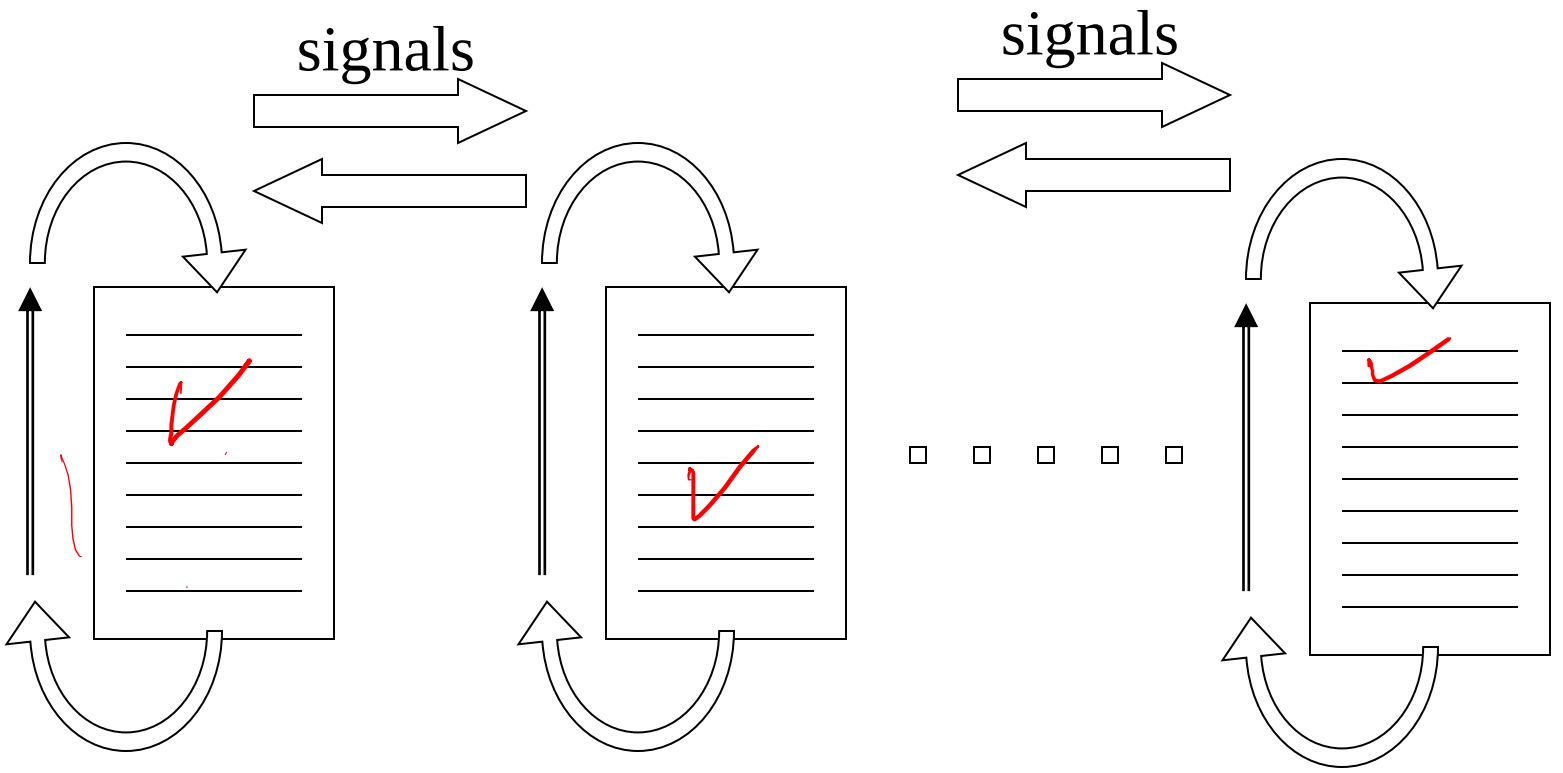
Events

Float Transaction

Signal Assignment



Concurrency in VHDL Descriptions



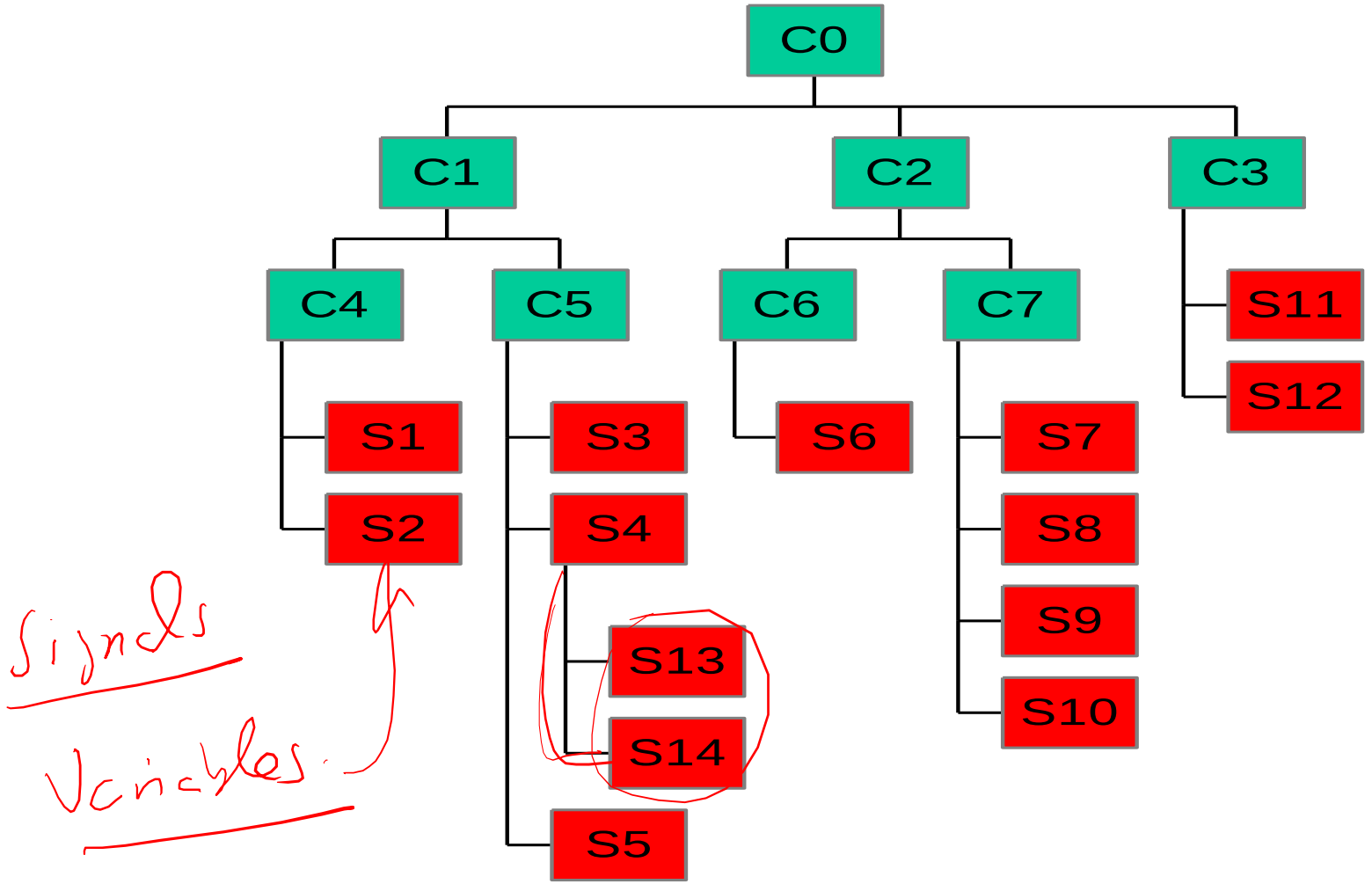
Concurrent and Sequential Computations

- Processes are concurrent
- Sequential activity within each process

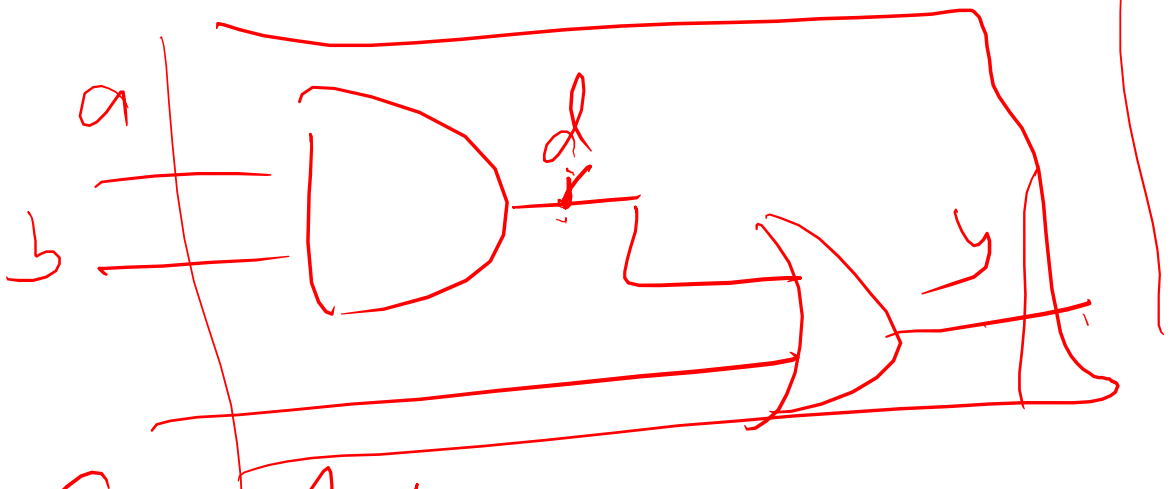
Nesting of statements :

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

Hierarchy in VHDL



Worksheet



t+δ

t+δ, t+2δ

~~d <= a AND b;~~

y <= c OR d;

d <= a AND b;

Lecture 28: Modeling Styles in VHDL

M. Balakrishnan

Dept. of Comp. Sci. & Engg.

I.I.T. Delhi

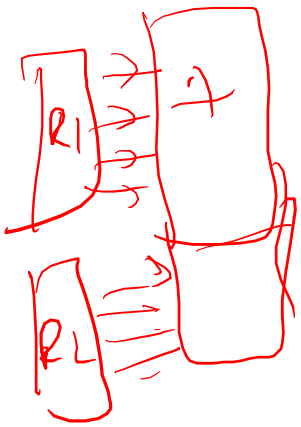
Modeling Styles

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

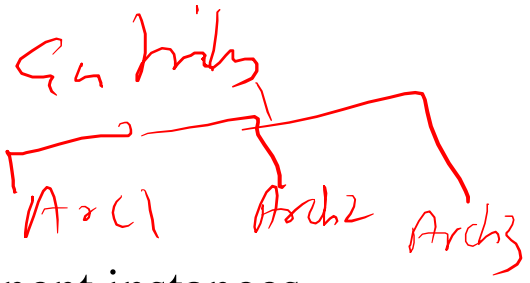
Modeling Choices in VHDL

- 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

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 => execution order)
- **Sequential statements**
- **Control constructs alter normal sequential flow**

Called Behavioral description in VHDL

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

Called Data flow description in VHDL

Concurrent Statements in VHDL

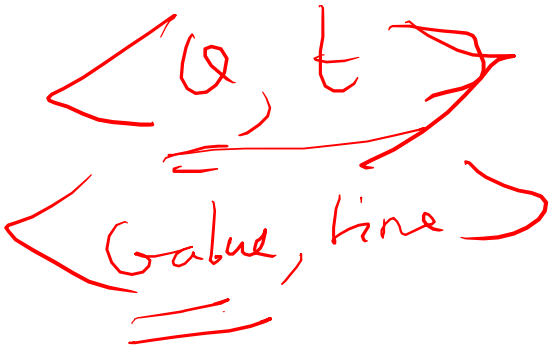
- process statement -- behavior
- concurrent procedure call -- behavior
- concurrent signal assign. -- data flow
- component instantiation -- structure
- generate statement -- structure
- block statement -- nesting
- concurrent assertion stmt -- error check

Example: 1-bit Full Adder

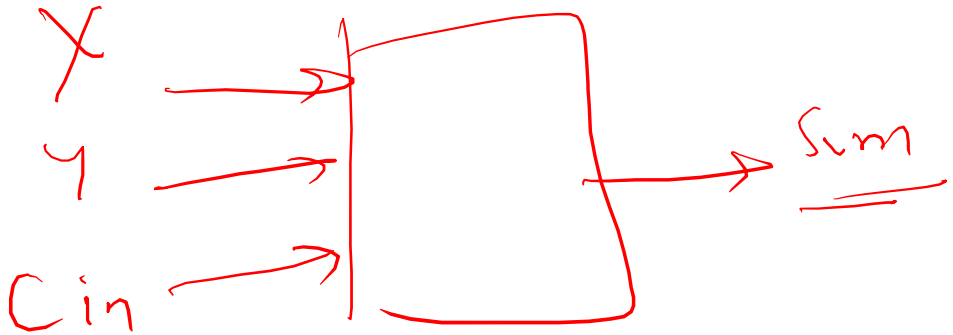
```
entity FullAdder is
    port (X, Y, Cin: in bit;     -- Inputs
          Cout, Sum: out bit);   -- Outputs
end FullAdder;
```

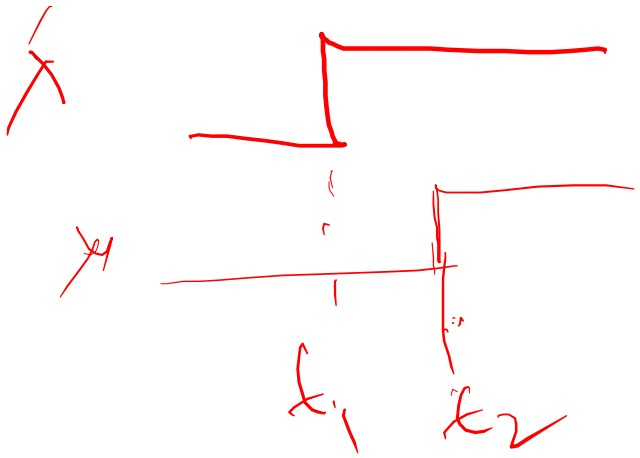
Example: 1-bit Full Adder (contd.)

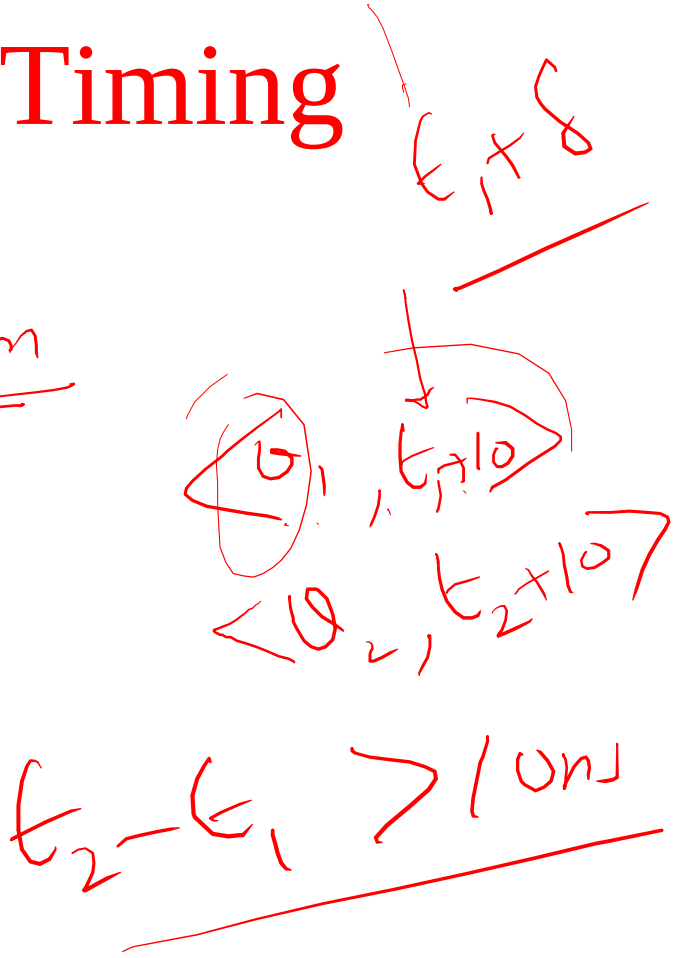
```
Architecture Equations of FullAdder is
begin        -- Concurrent Assignment
   Sum <= X xor Y xor Cin after 10 ns;
   Cout <= (X and Y) or (X and Cin) or (Y
      and Cin) after 15 ns;
end Equations;
```



1-bit Full Adder Timing

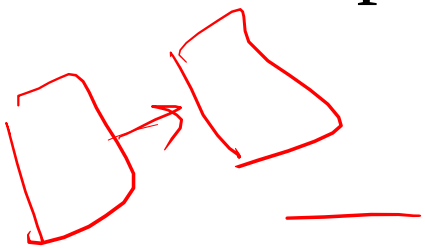


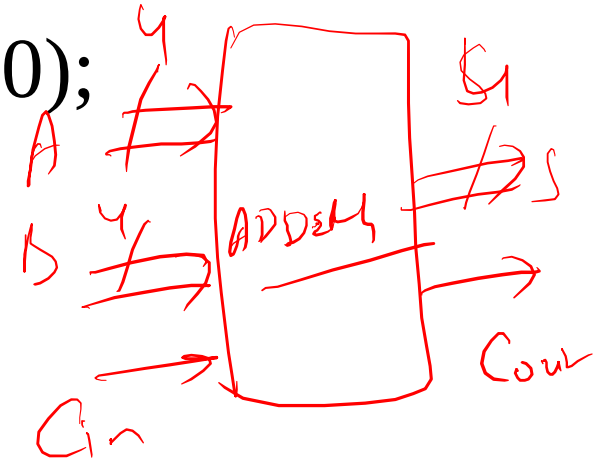




Example: 4-bit Adder

```
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;
```



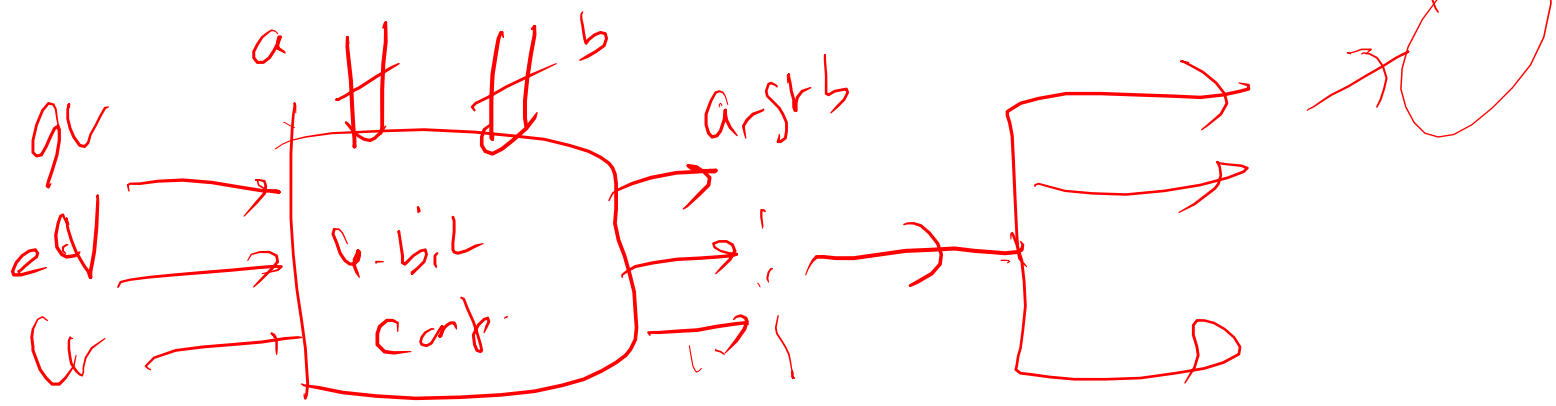


Example: 4-bit Adder (contd.)

```
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;
```

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;
```



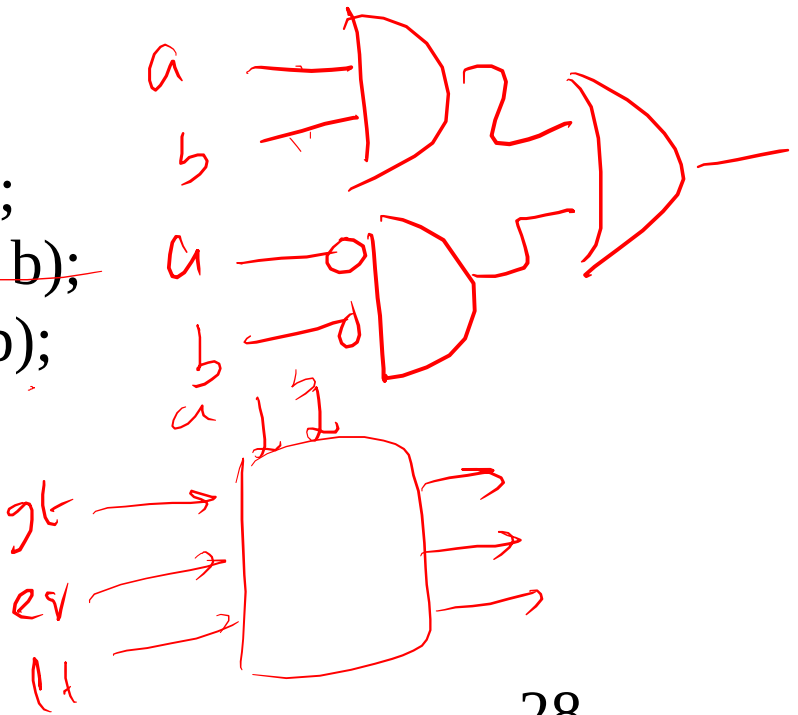
Doubt in Syntax ???

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;
```

Example: 1-bit Comparator (data flow)

```
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;
```

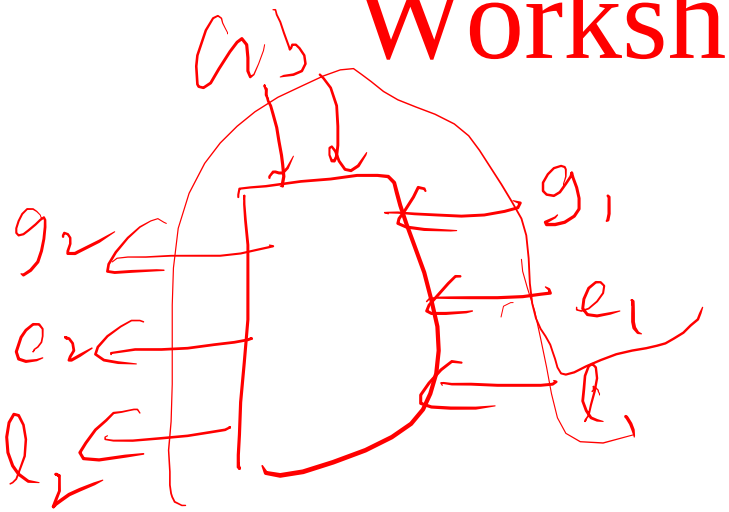


References

- 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, Morgon Kaufmann

Worksheet



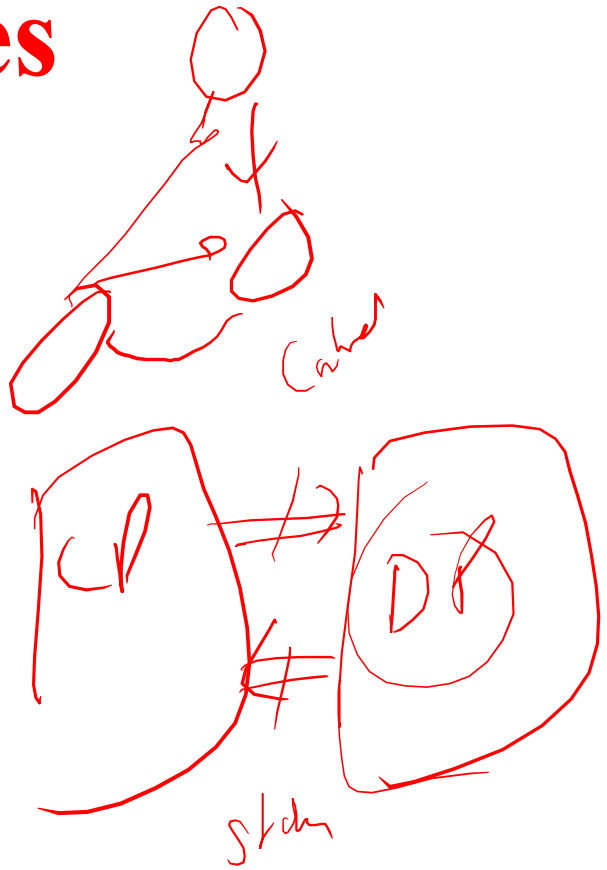
$$\begin{cases} g_2 = g_1 + e_1 \cdot \overline{a\,b} \\ e_2 = e_1 \cdot (ab + \bar{a}\bar{b}) \\ l_2 = l_1 + e_1 \cdot \bar{a} b \end{cases}$$

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

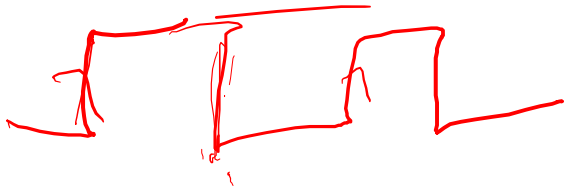
- process statement -- behavior
- concurrent procedure call -- behavior
- concurrent signal assign. -- data flow
- component instantiation -- structure
- generate statement -- structure
- block statement -- nesting
- concurrent assertion stmt -- error check

Example: D Flip-Flop

```
entity DFF is
   port (D, CLK: in bit;
         Q: out bit; QN: out bit := '1') ;
end  DFF;
```

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;
```

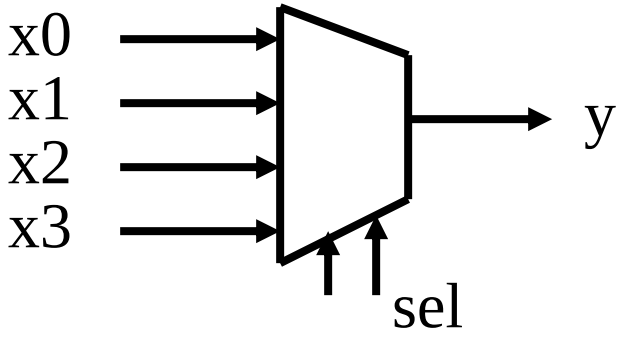


Timing Diagram

Data Flow & Process Statement

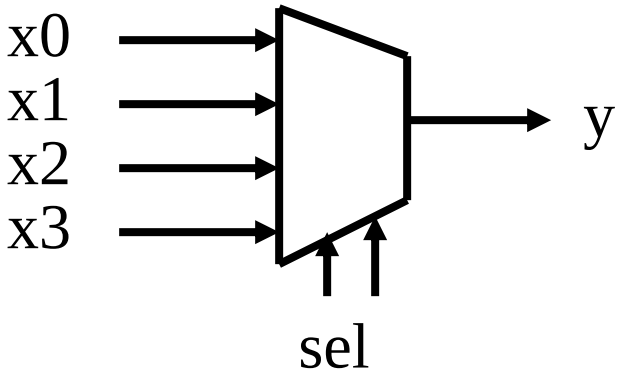
Concurrent Conditional Assignment: 4 to 1 Multiplexer

```
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
```



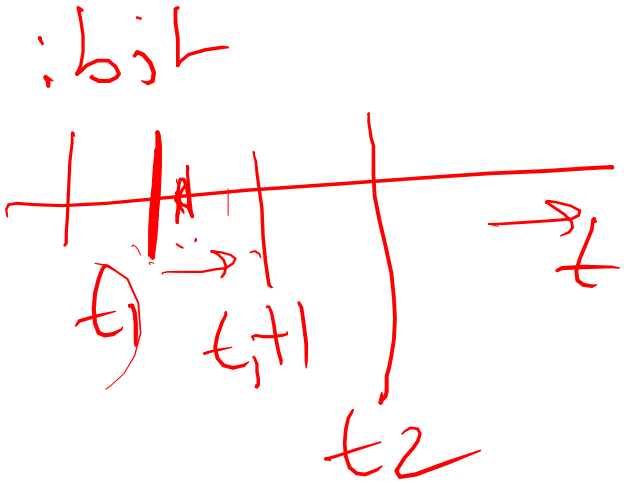
Note allowed only within a Process statement

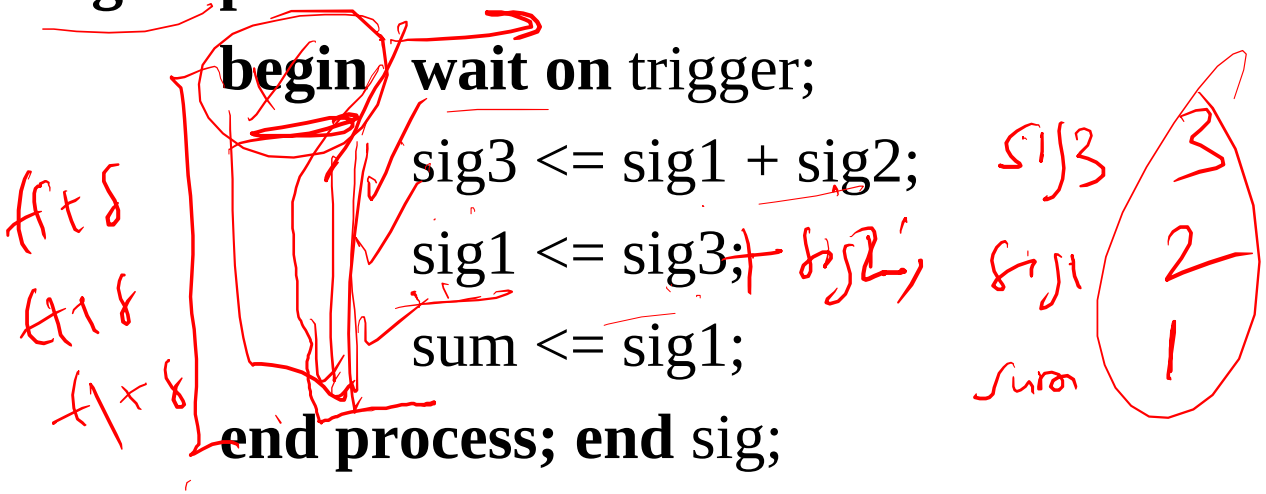
Variables And Signals

```
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;
```

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;
               sum <= sig1;
        end process; end sig;
```

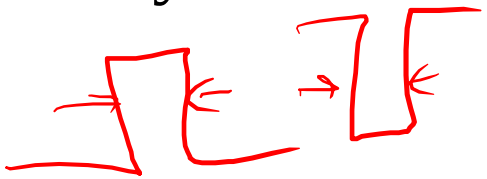




Inertial and Transport Delays

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

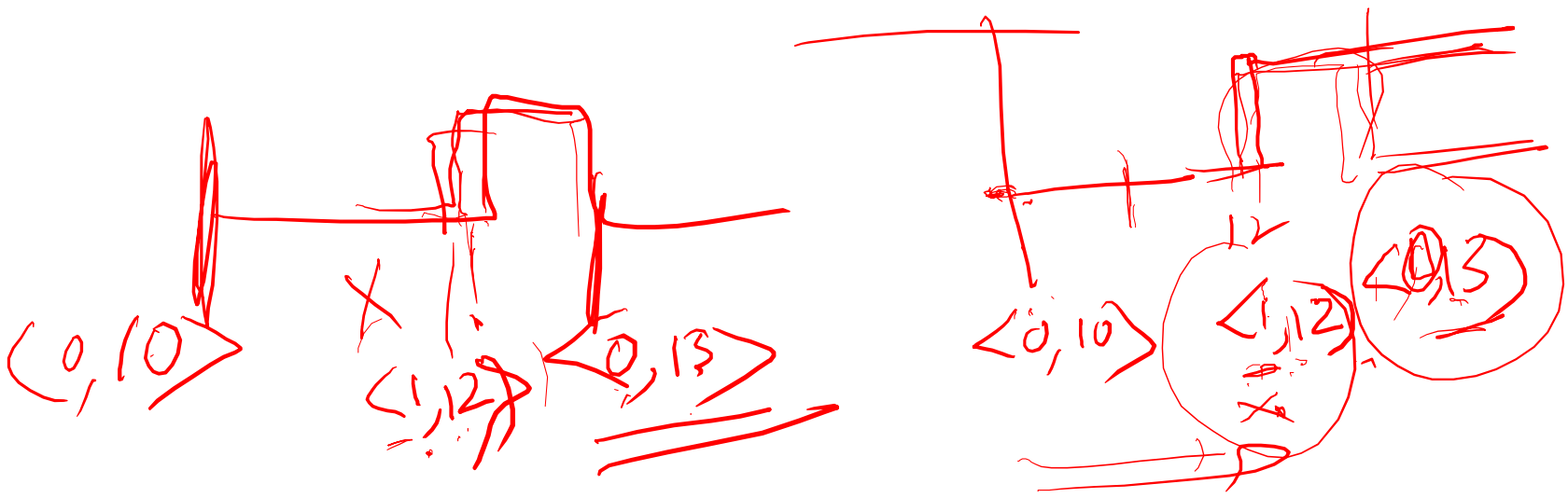
y2 <= x **after** 5 ns; -- inertial delay



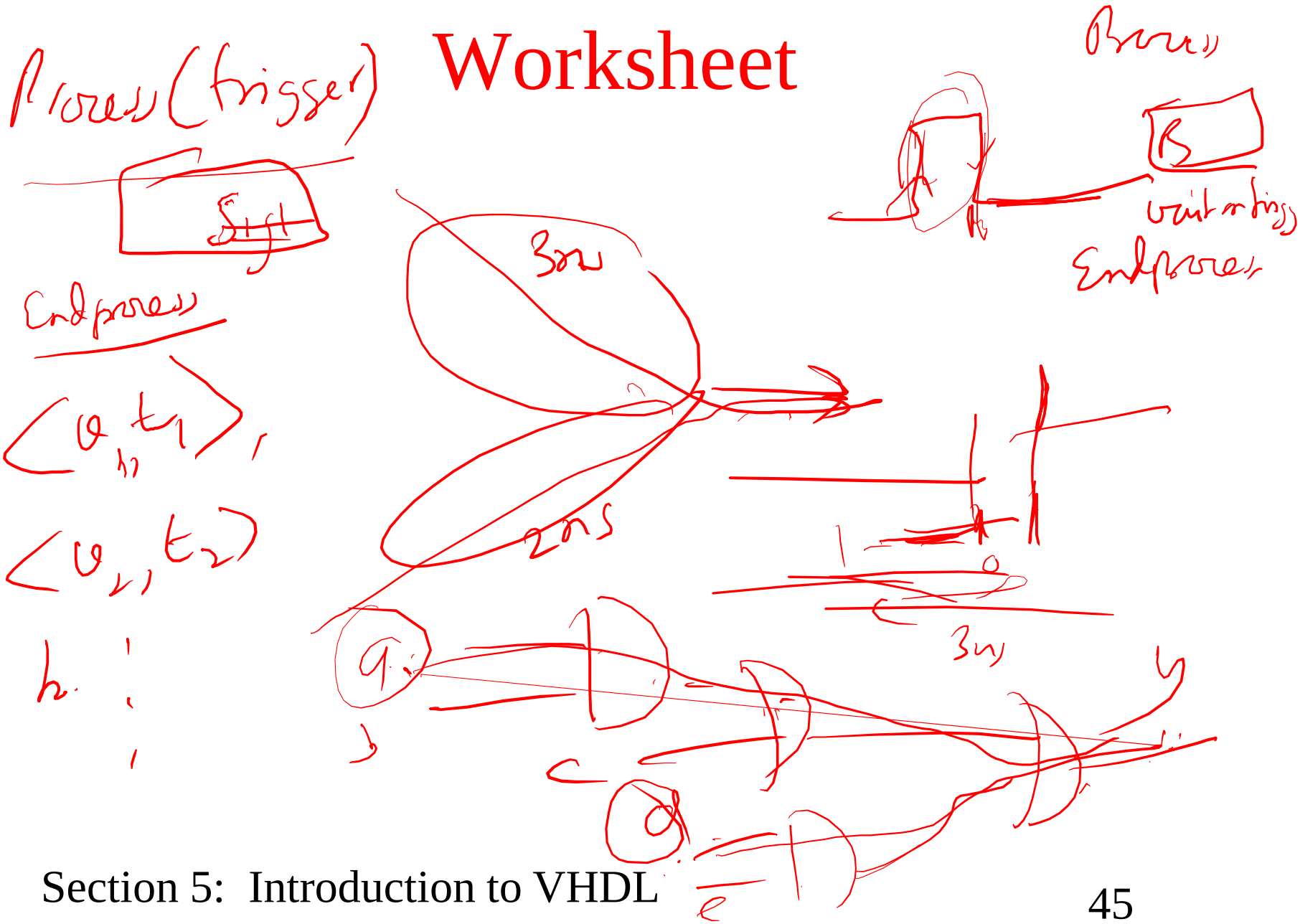
y3 <= **reject 2 ns** x **after** 5 ns; -- mix of inertial and transport delay but rejection of pulse less than 2 ns

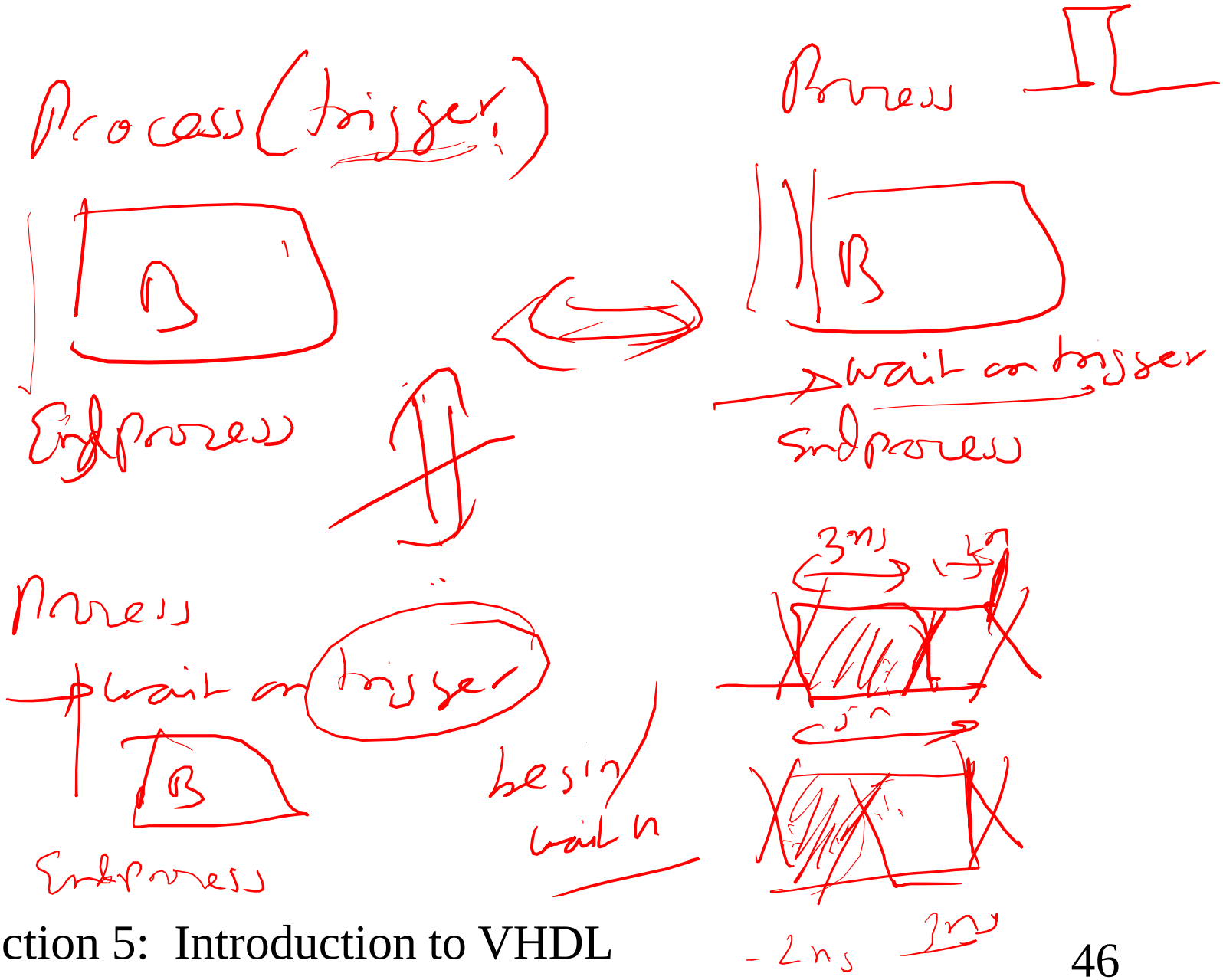
Inertial & Transport Delays (contd)

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

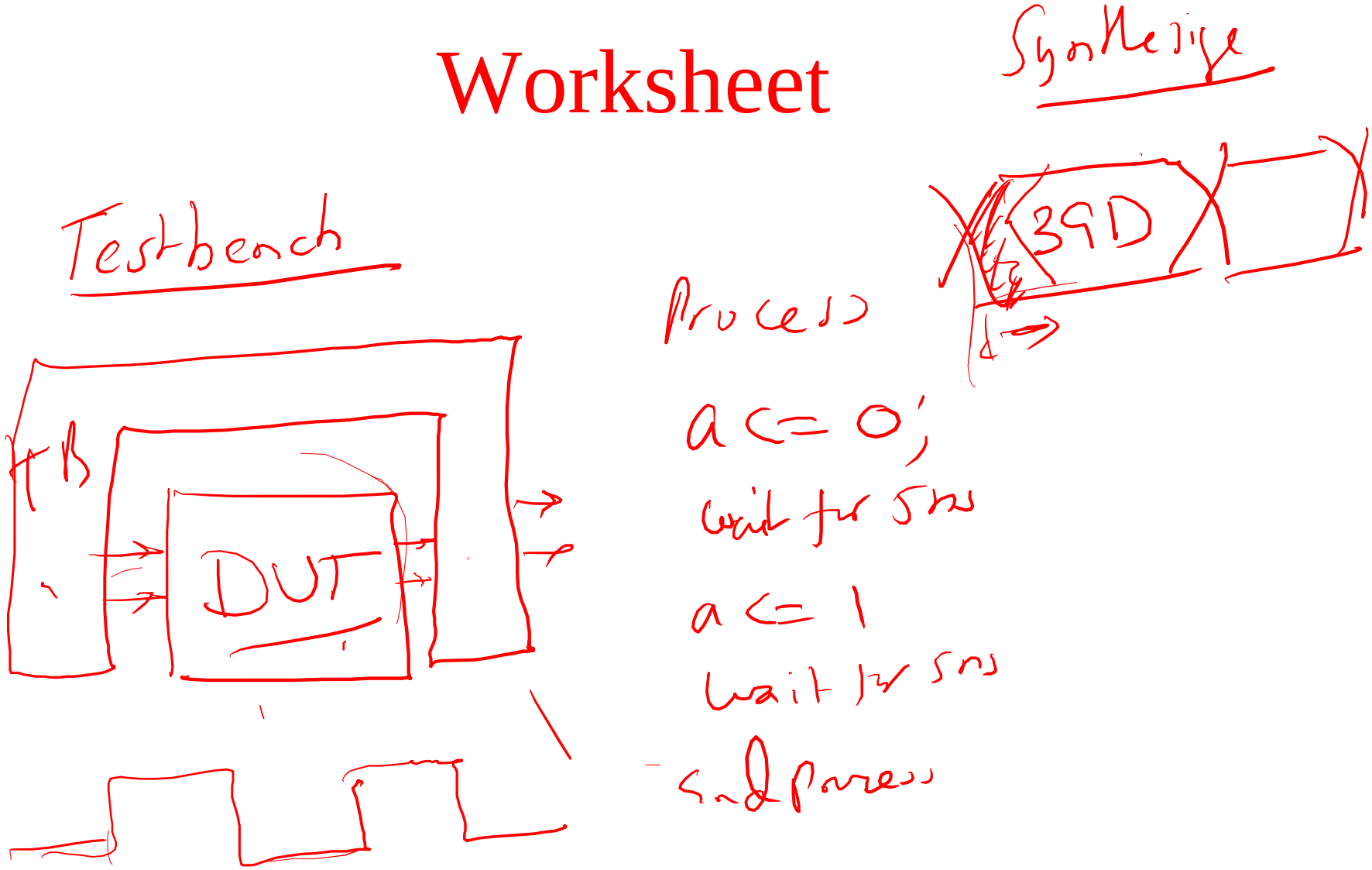


Worksheet




Process (trigger)
B
EndProcess
Process
B
wait on trigger
endprocess
Process
wait on trigger
B
EndProcess
begin
wait
3ns
2ns
3ns

Worksheet


Synthesize
39D
Testbench
DUT
Process
a <= 0;
wait for 5ns
a <= 1
wait for 5ns
end process

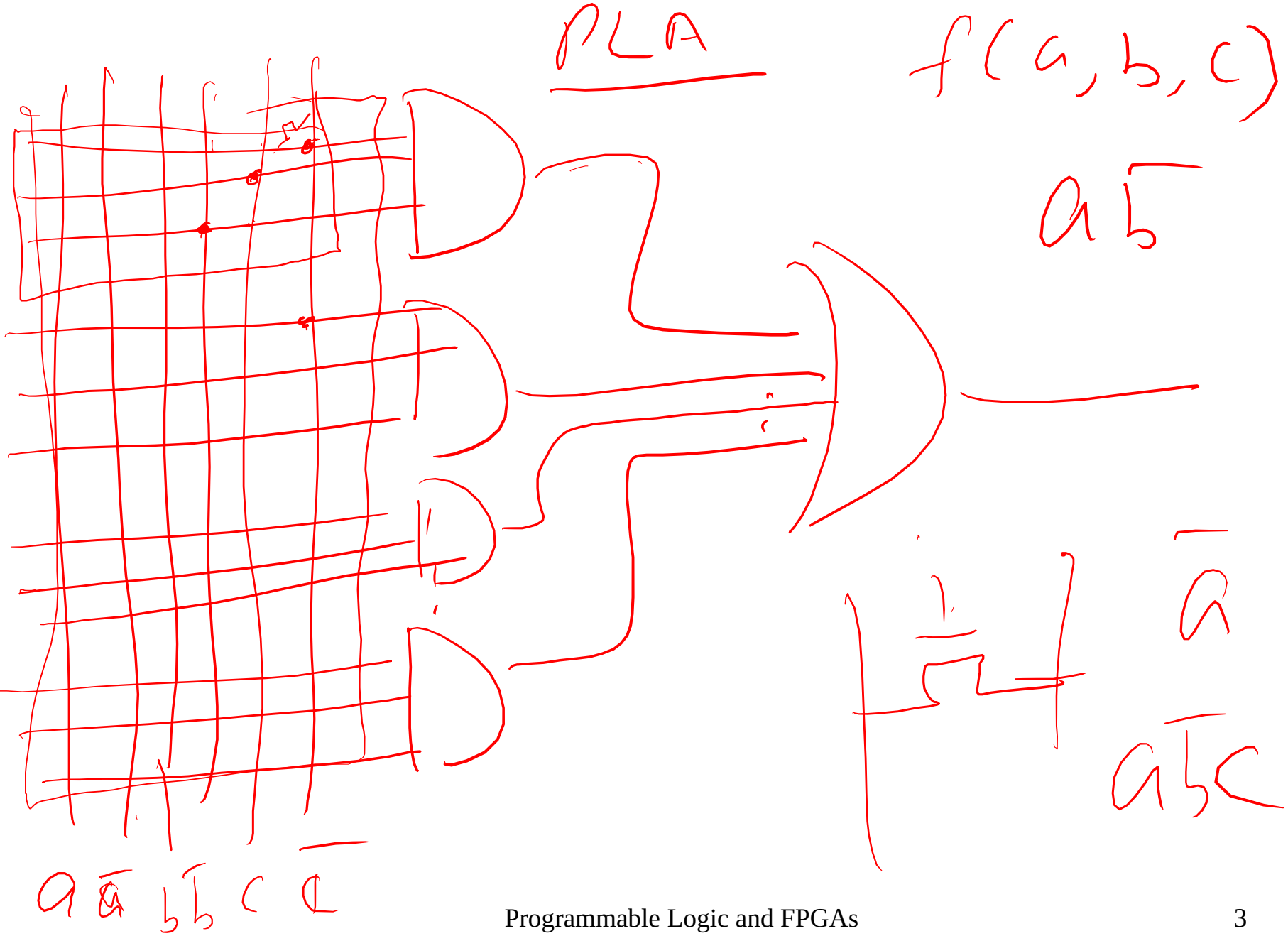
Lecture 32: Programmable Logic

M. Balakrishnan

Logic Implementation Options

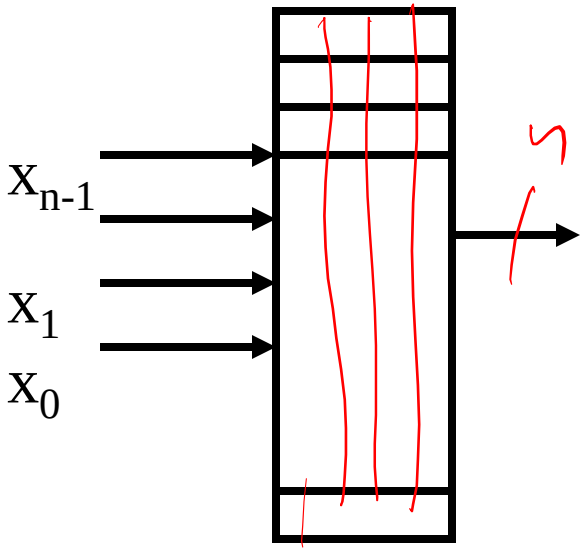
To implement a n-variable function (with k minterms)

- 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

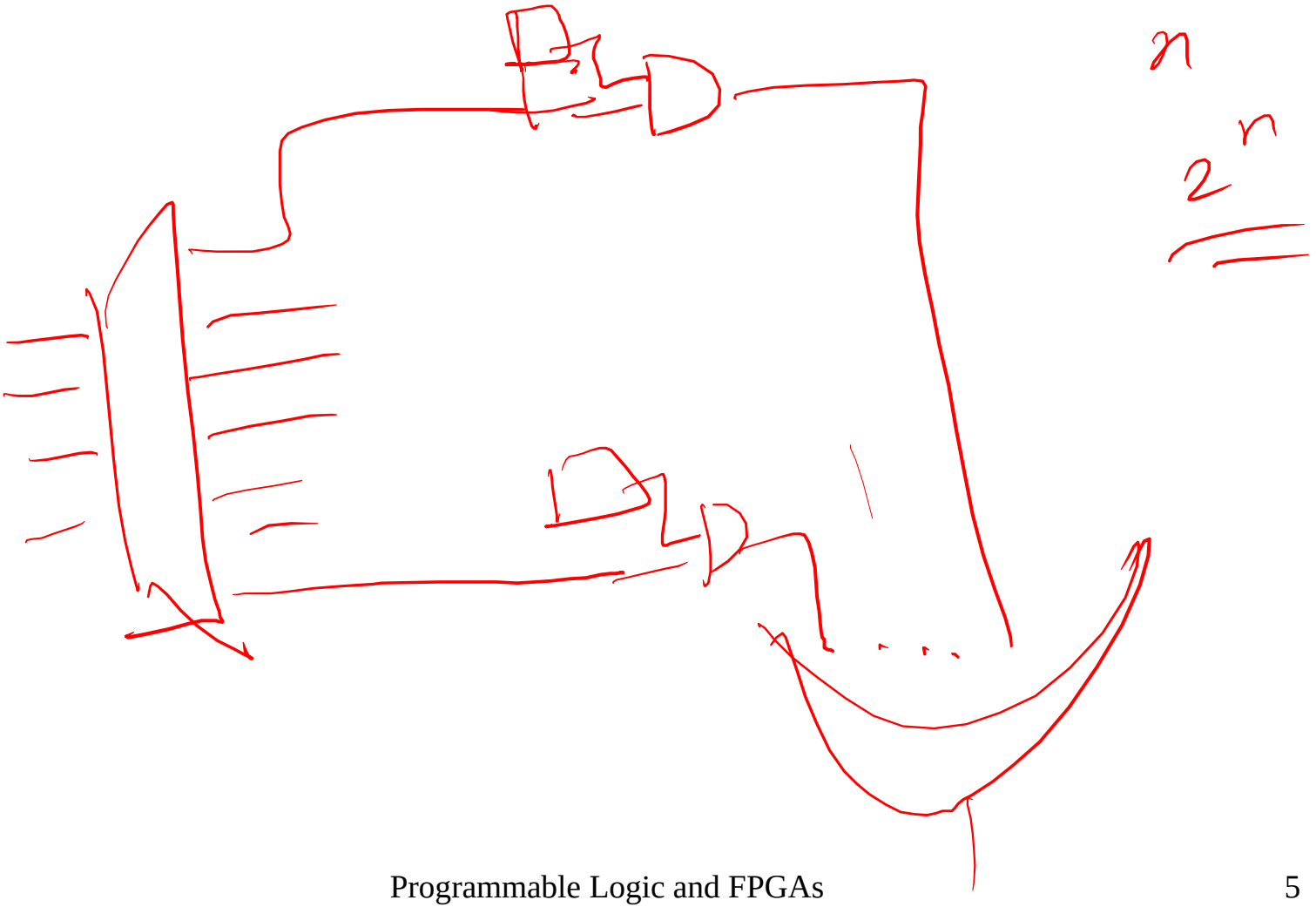

PLA
f(a, b, c)
$a\bar{b}$
$\bar{a}$
$a\bar{b}c$
$a\bar{a}b\bar{b}c\bar{c}$

ROM (Read Only Memory)

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

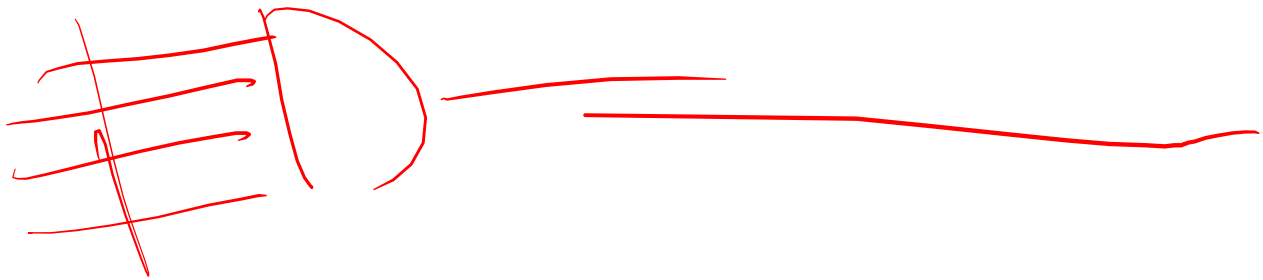


ROM Design Example



Implementing Logic Using ROMs

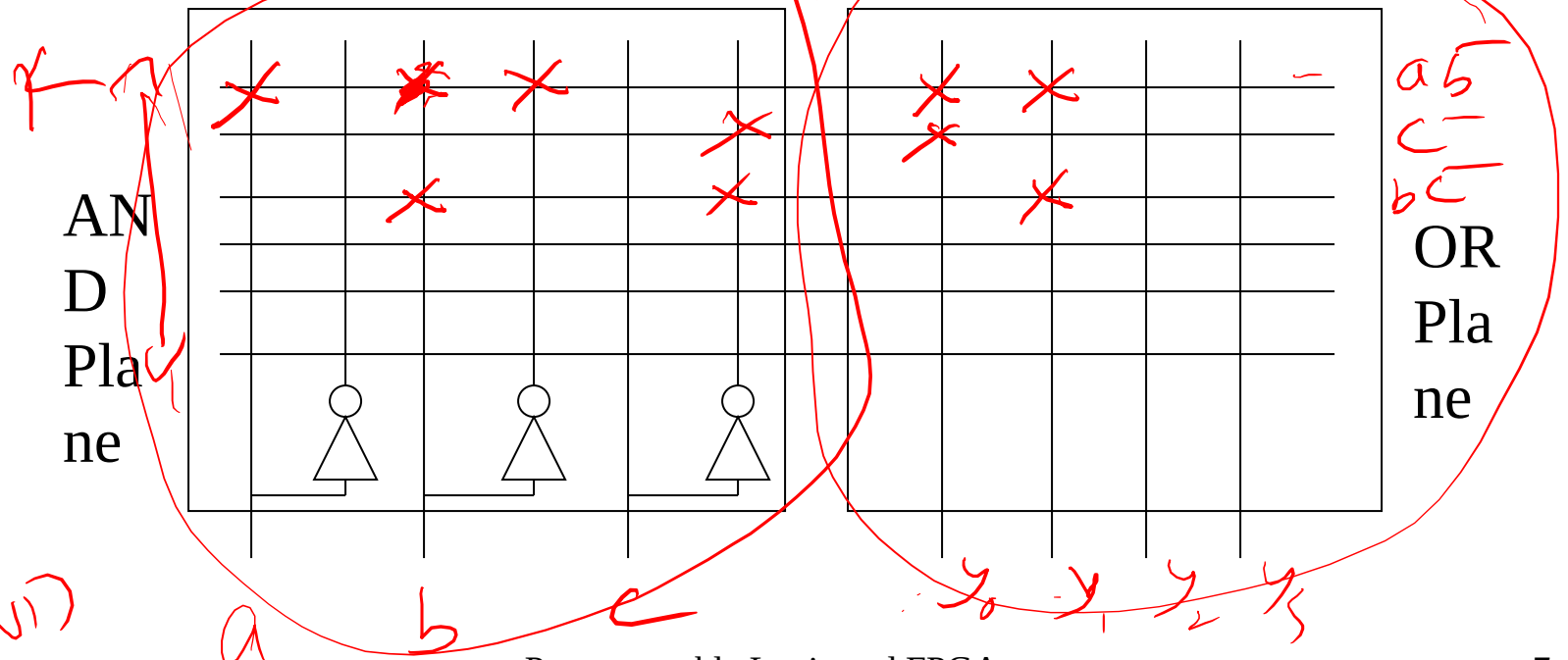
- Direct implementation of the function
- Extremely flexible as reprogramming can implement a completely different function



PLA (Programmable Logic Arrays)

PLA Specification: n × k × m

(N inputs, K product terms and M outputs)

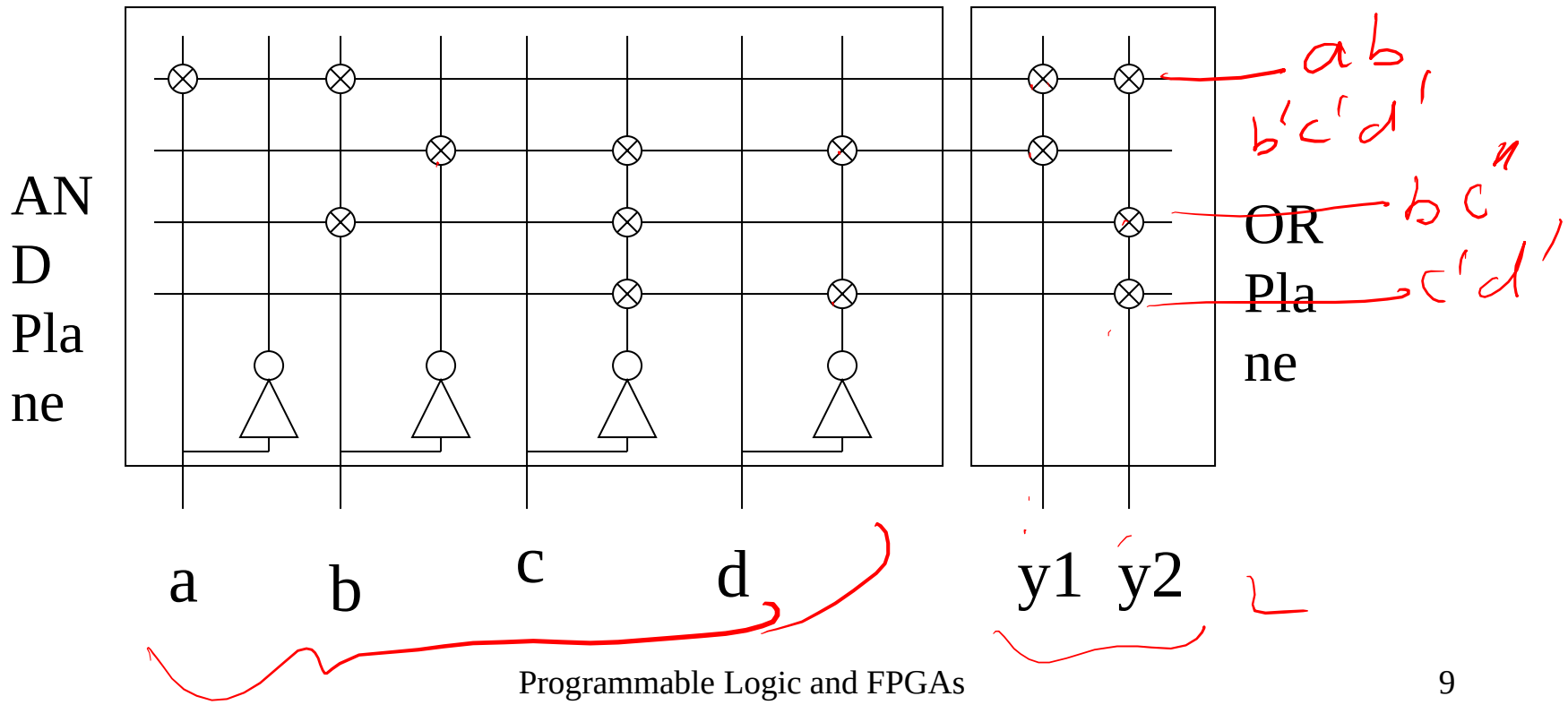


Implementing Logic Using PLAs

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

Implementing Logic Using PLAs (Example)

y1= ab + b'c'd' y2 = 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?

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.)?

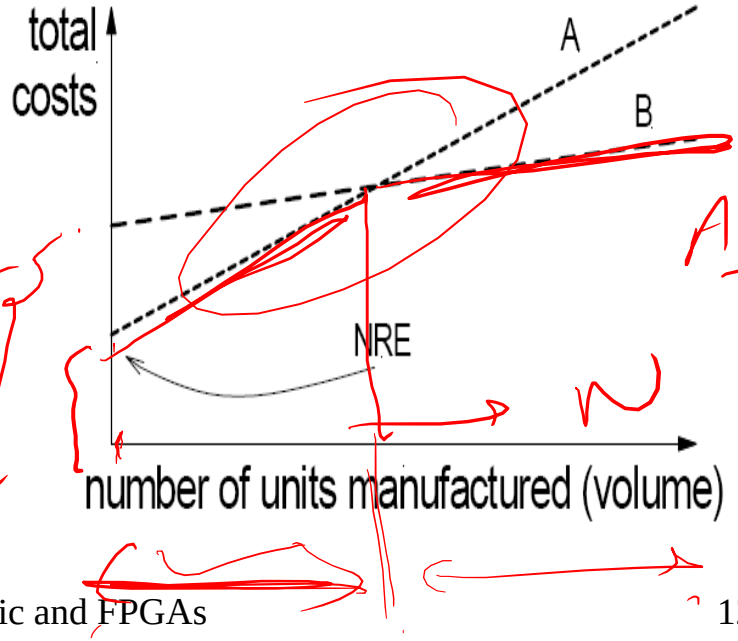
Need to worry about two kinds of costs:

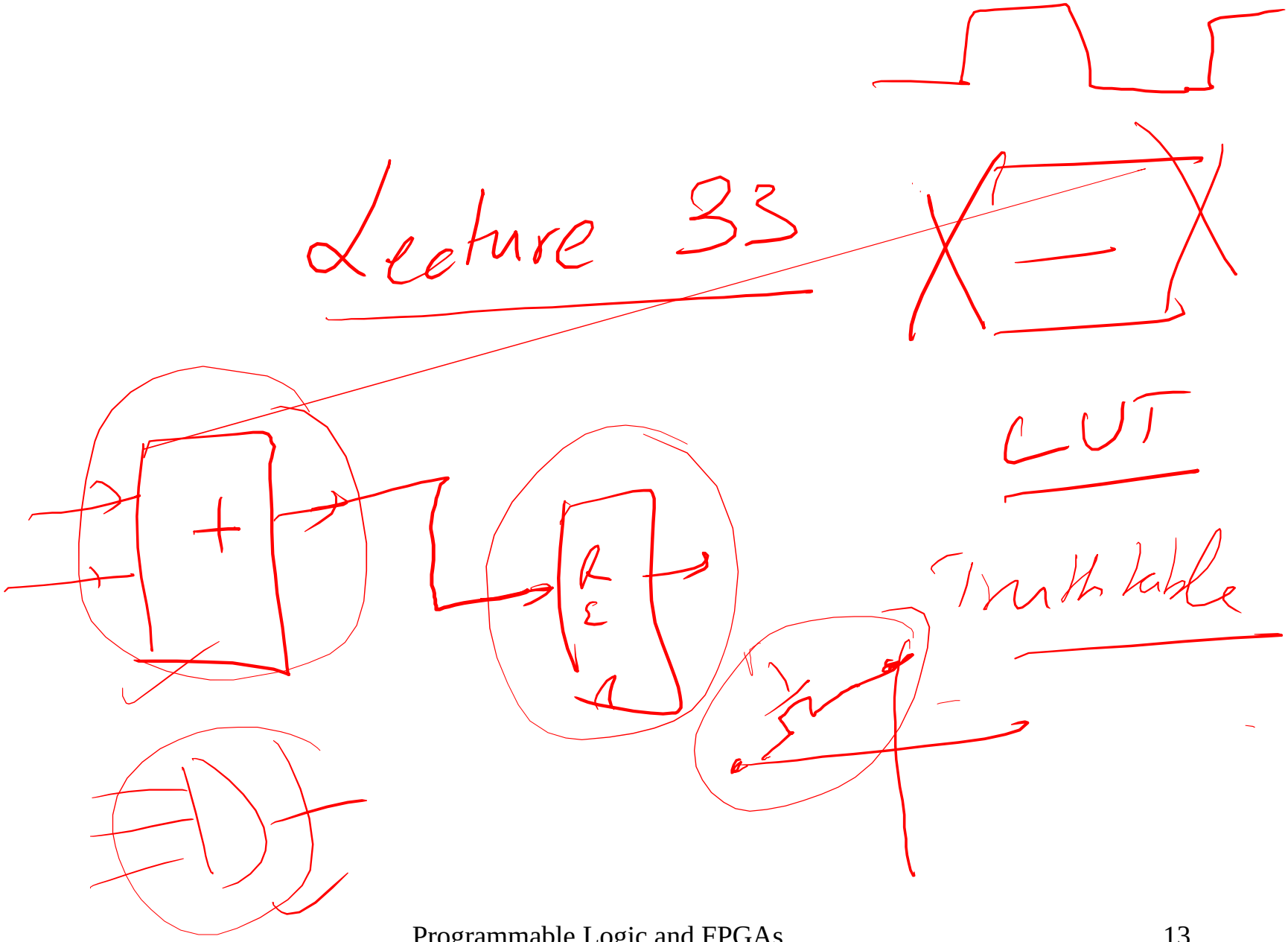
a. Cost of development, sometimes called non-recurring engineering (NRE)

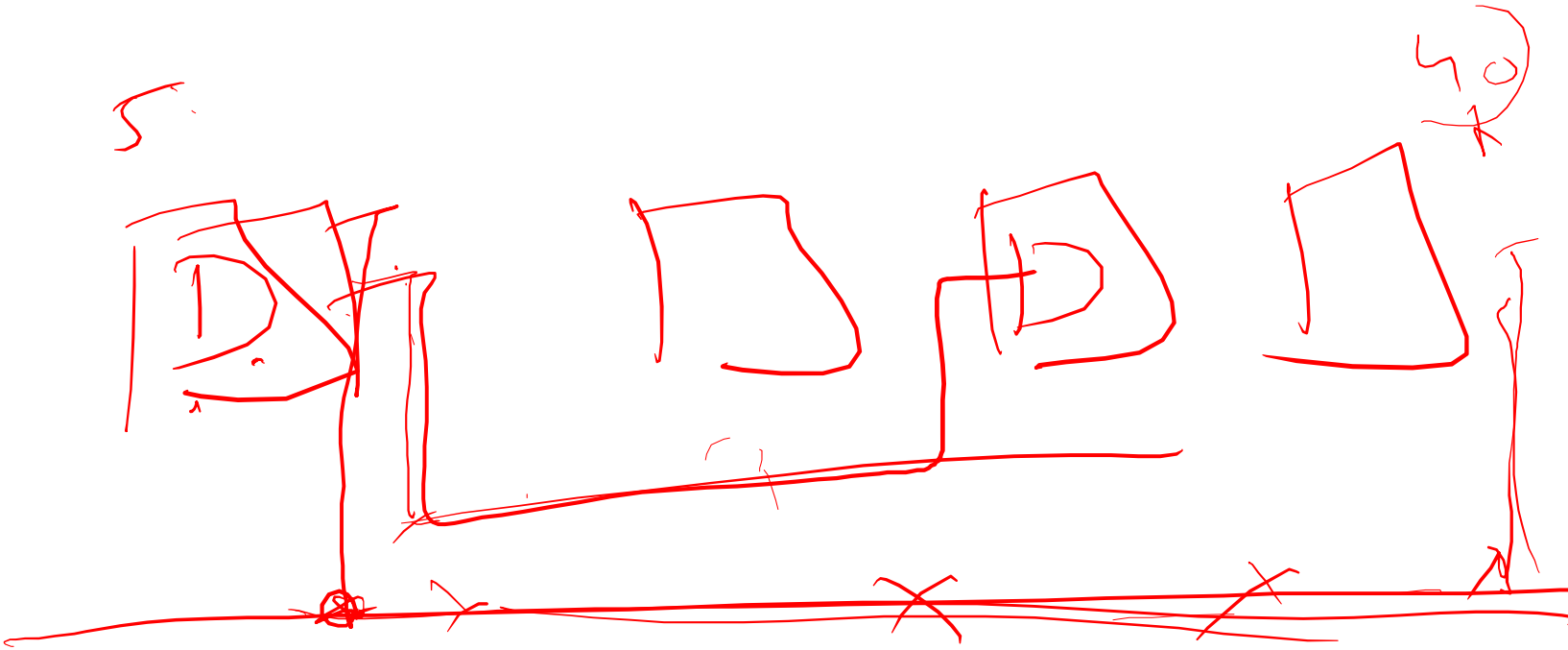
b. Cost of manufacture or Recurring cost (RC)

A: FPGAs (low NRE, high RC)

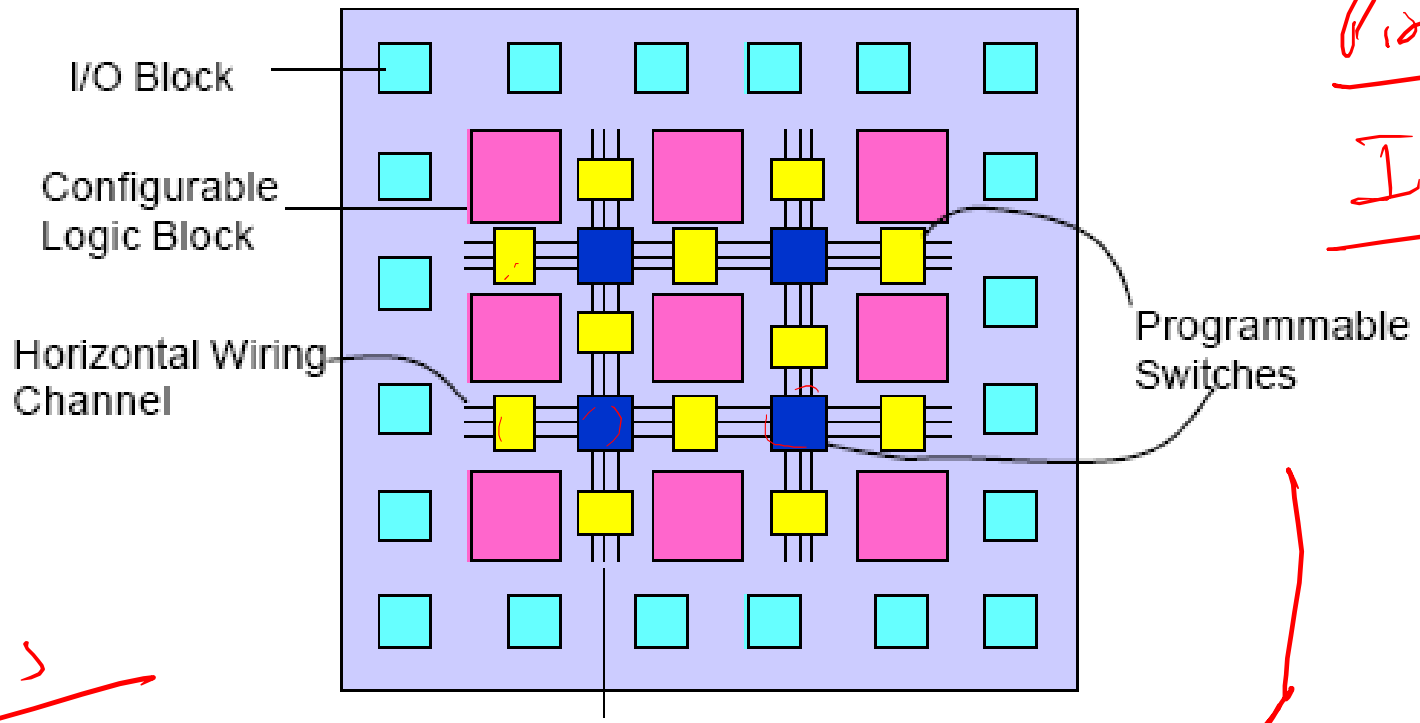
B: ASICs (high NRE, low RC)



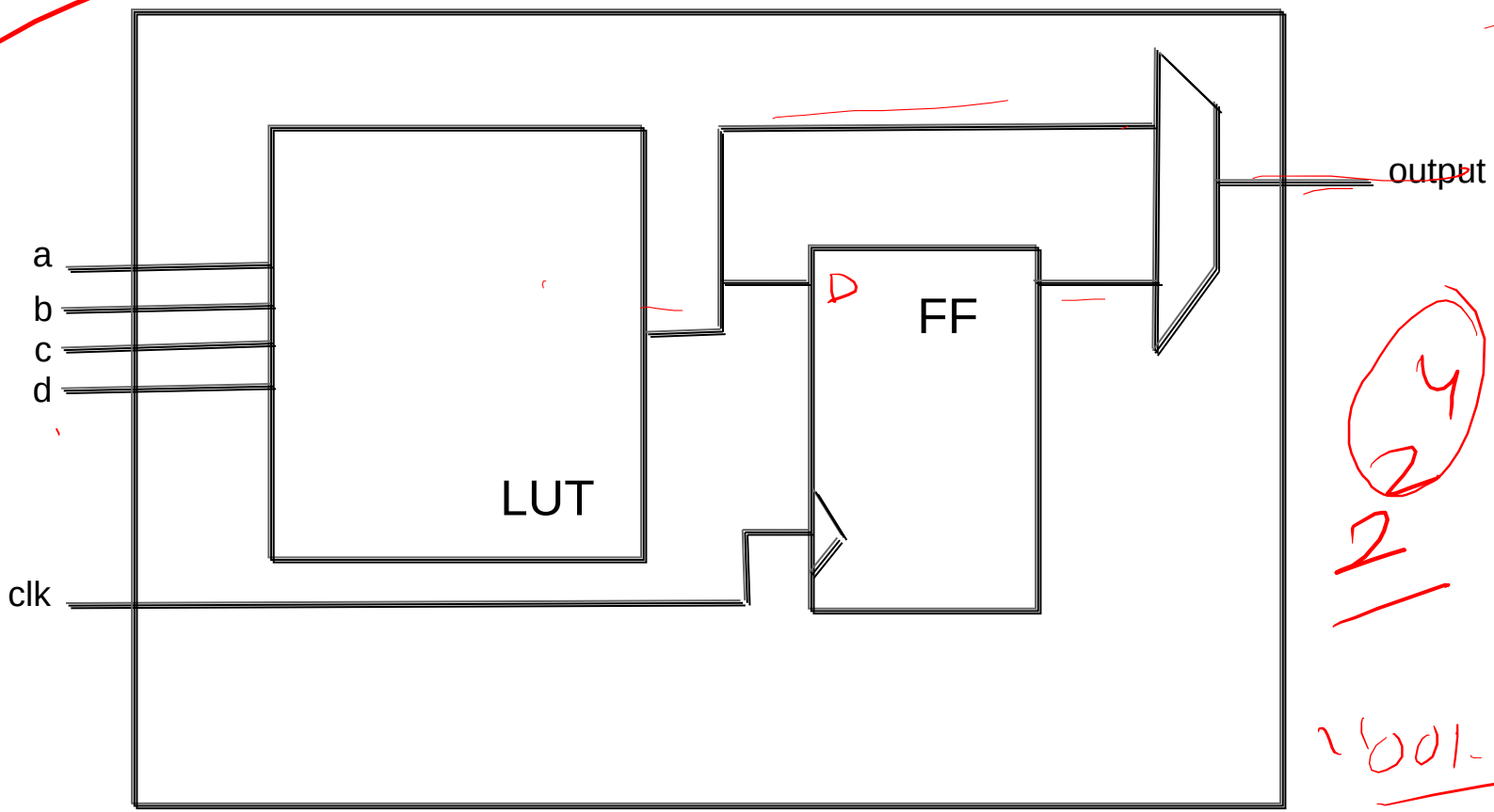

Lecture 33
LUT
Truth table



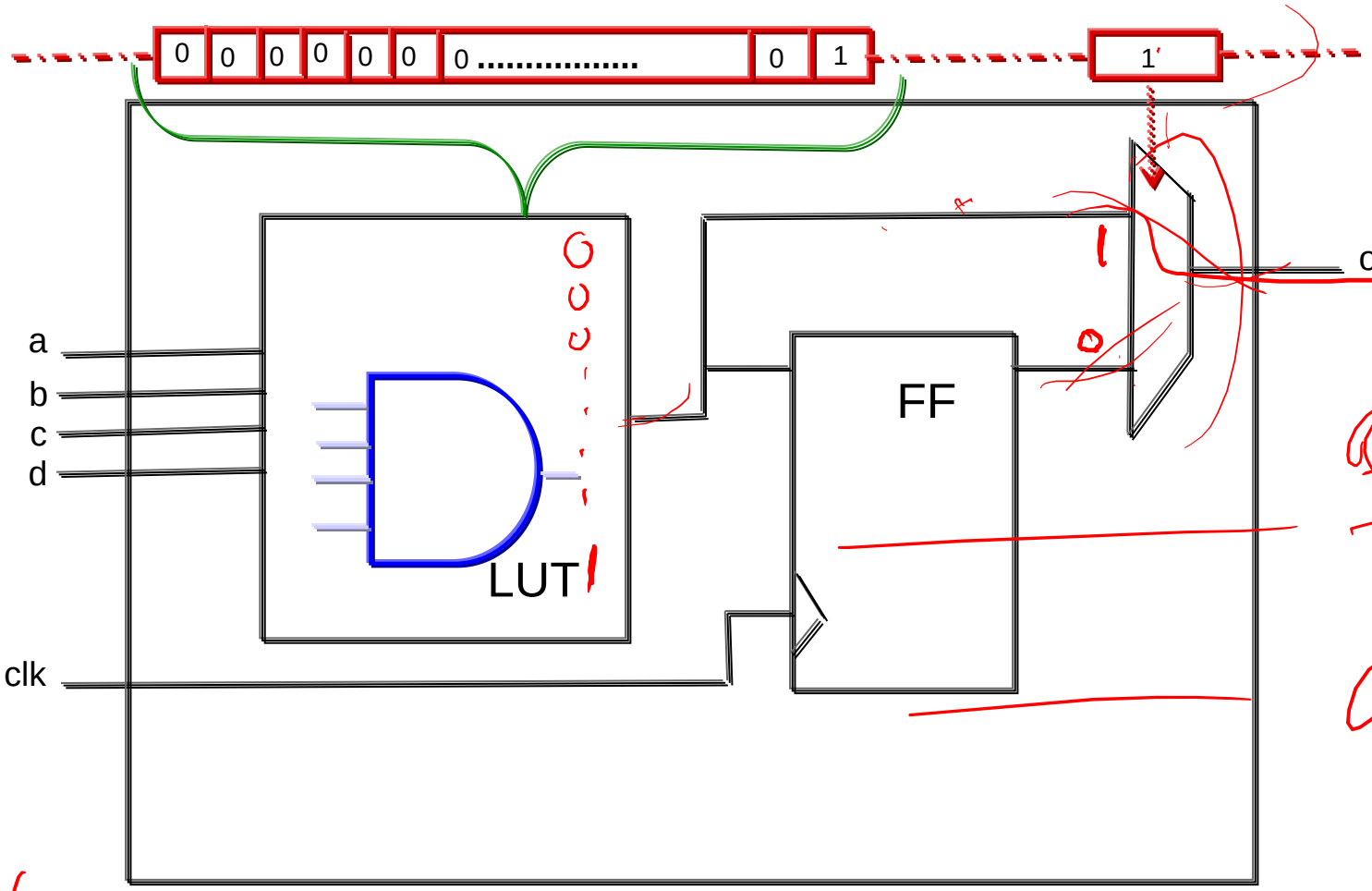
FPGA Blocks


I/O Block
Configurable
Logic Block
Horizontal Wiring
Channel
Programmable
Switches

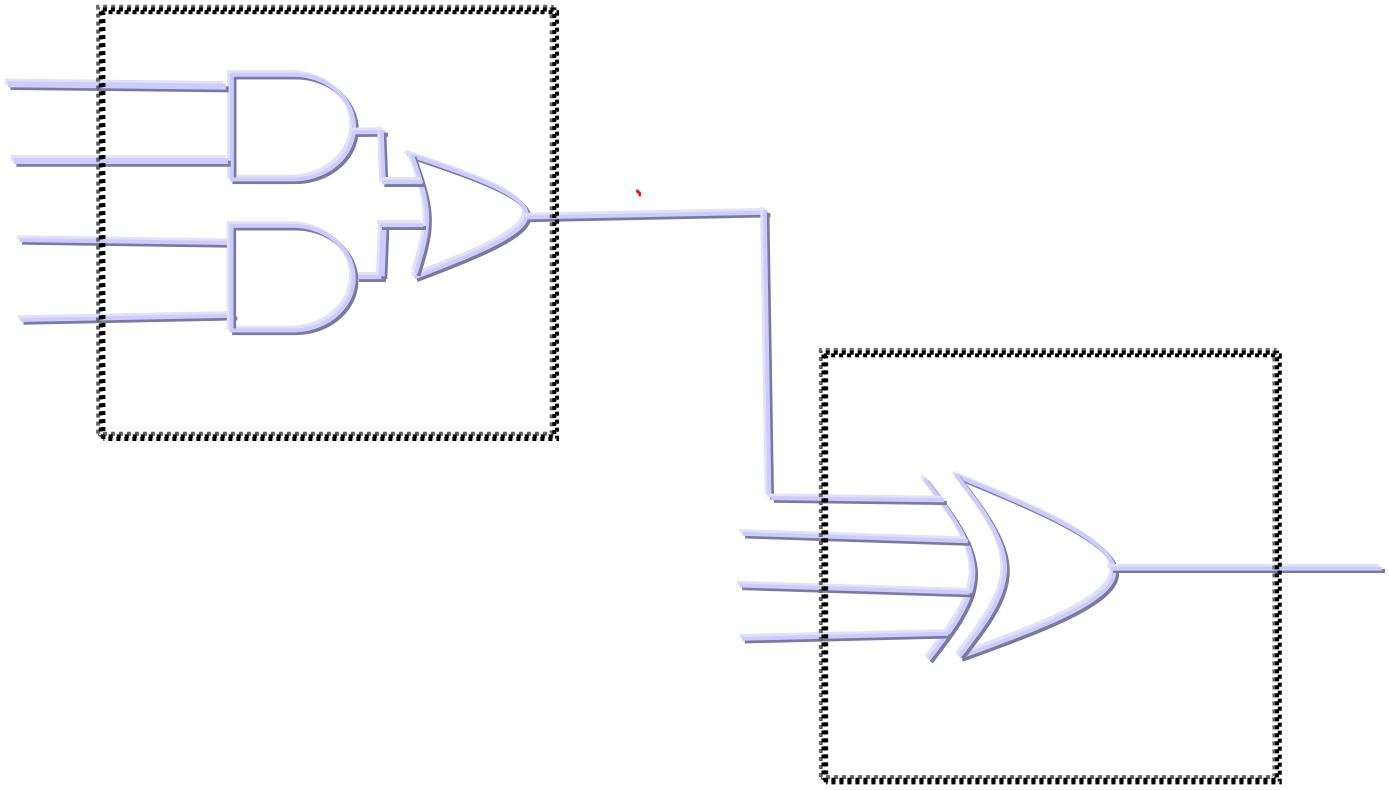
A Simple FPGA Logic Block



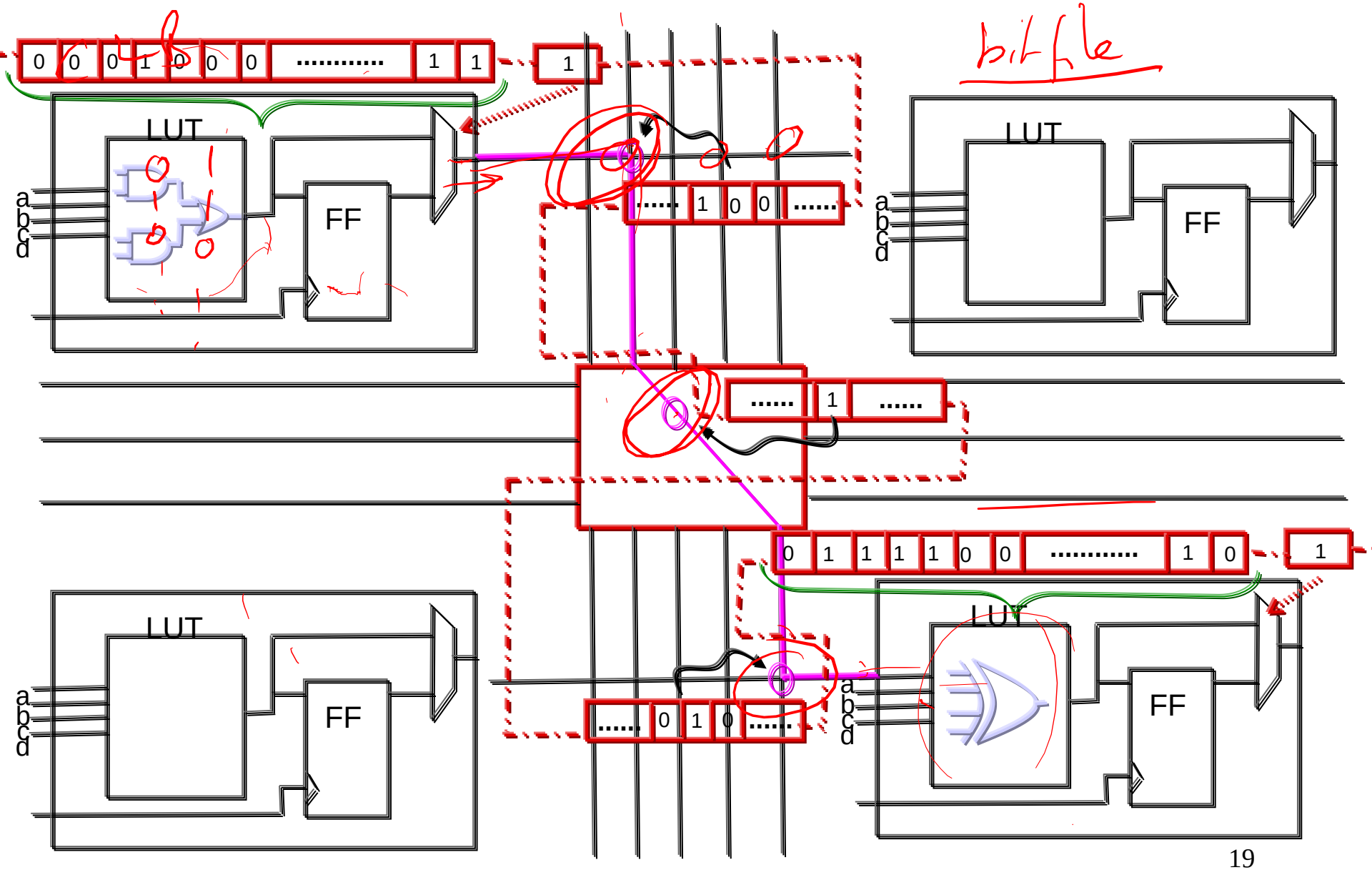
A Simple FPGA Logic Block



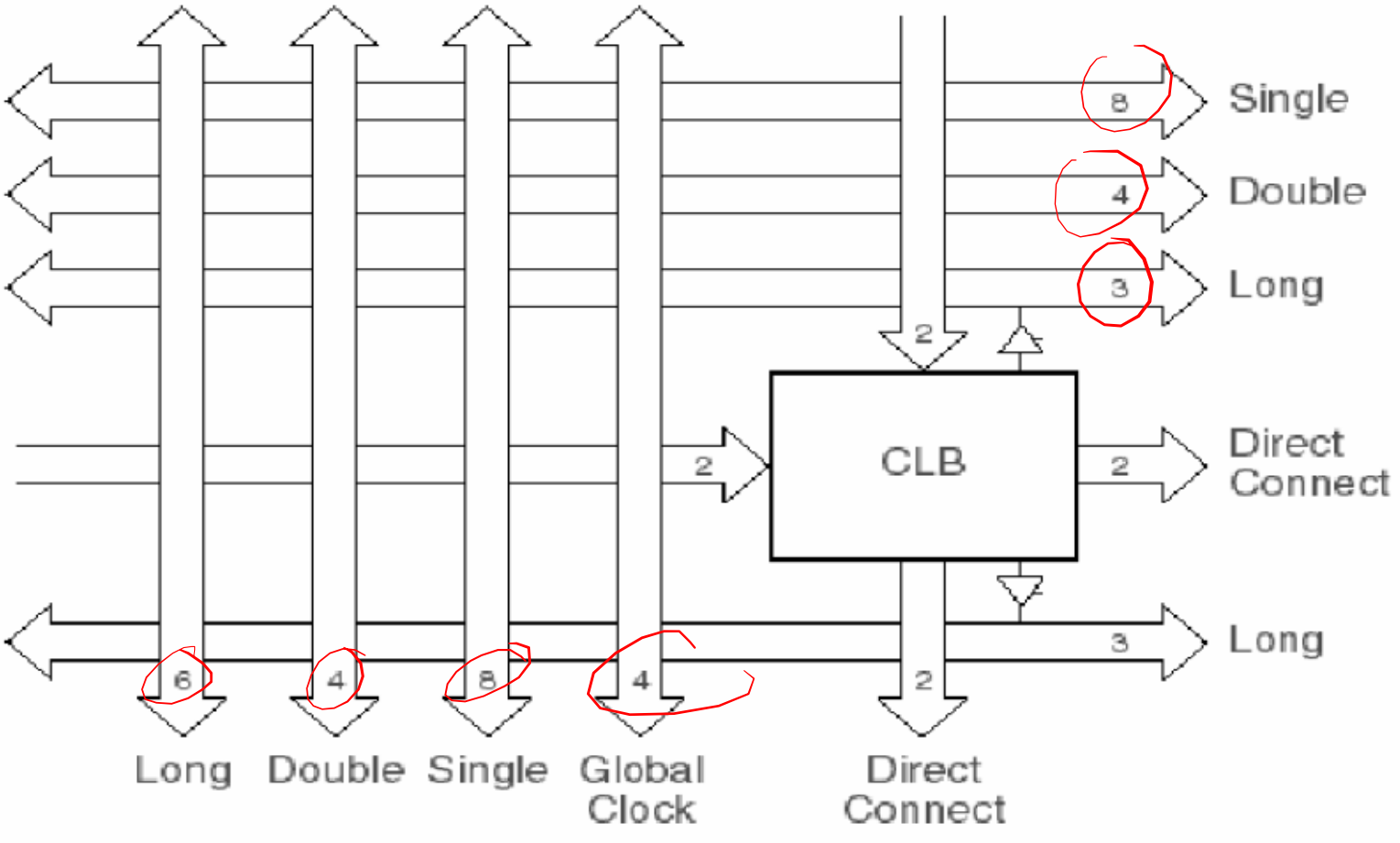
Simple Circuit



LUT Configuration bits



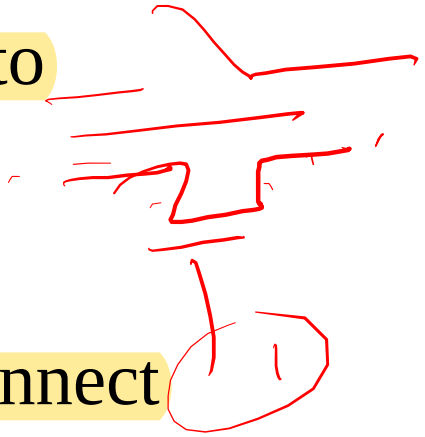
Interconnections



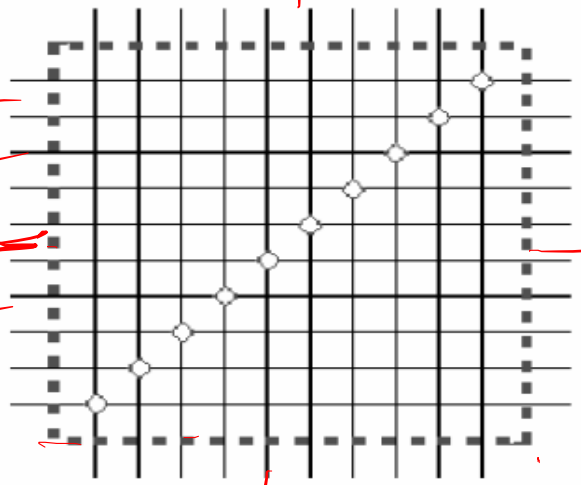
Source: xilinx.com

Programmable Interconnects

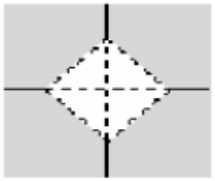
- **Connection box**
  - Connects input/output of logic block to interconnect channels.
- **Switch box**
  - Enables the connection of two interconnect lines.
- **Transmission gate (or a pass transistor) is used for each connection.**



Programmable Switching Matrix



programmable switch element



turning the corner, etc.

Methods of Interconnection

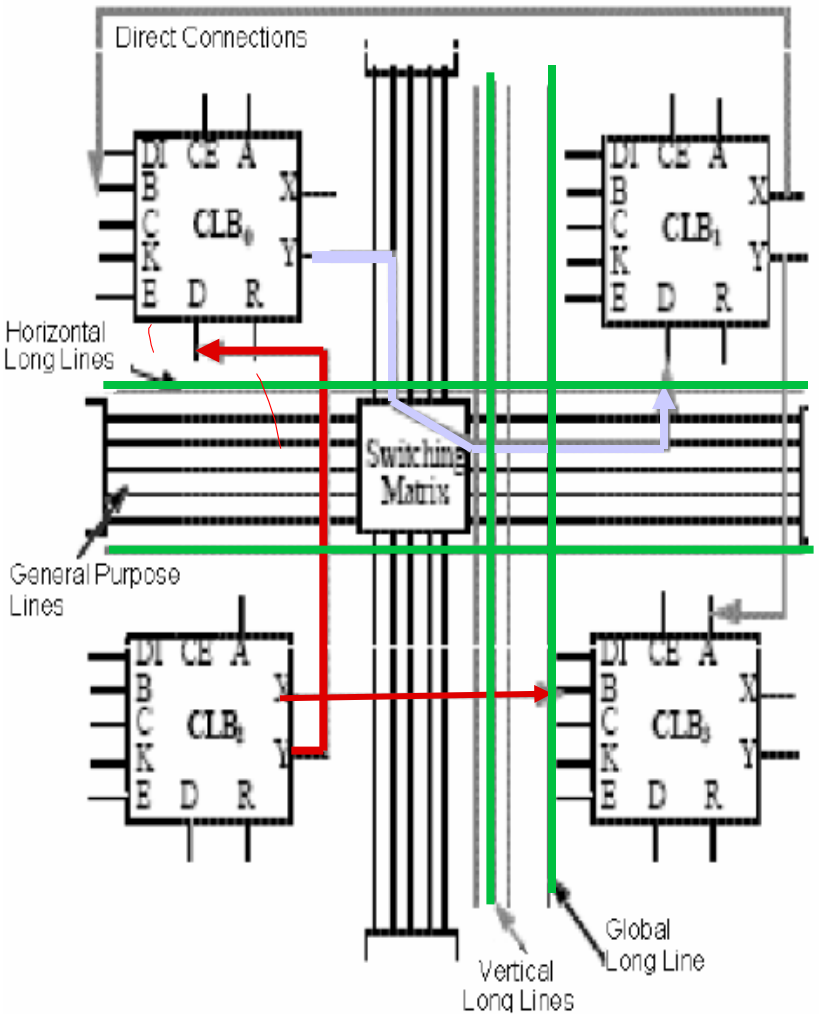
Direct Interconnect

- Connects adjacent CLBs

Long line Interconnect

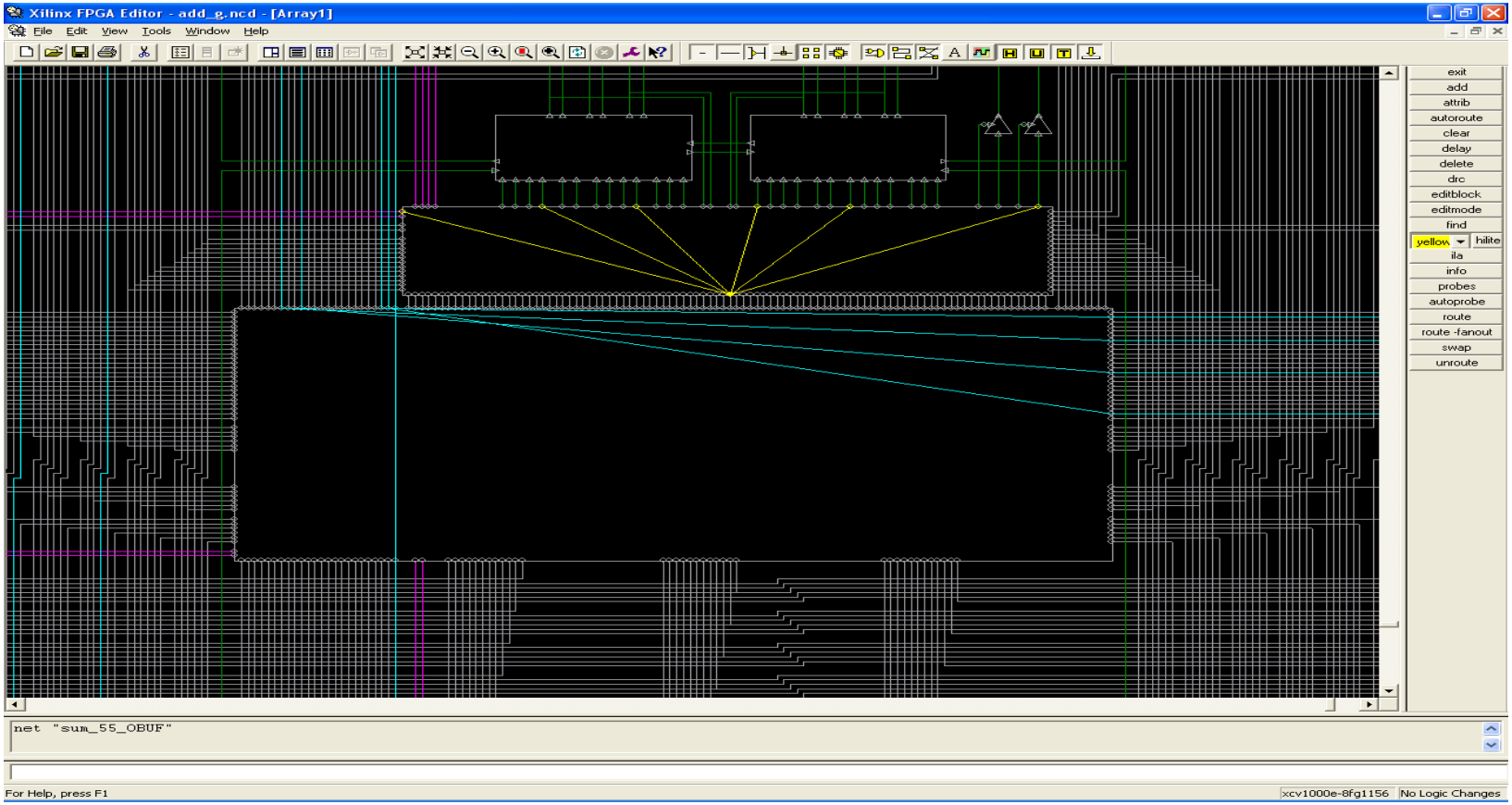
- Time critical signals through direct interconnects without going through switch boxes ➔ very fast
- Horizontal, vertical & Global long lines

-Global long lines for clocks and resets



Source: xilinx.com

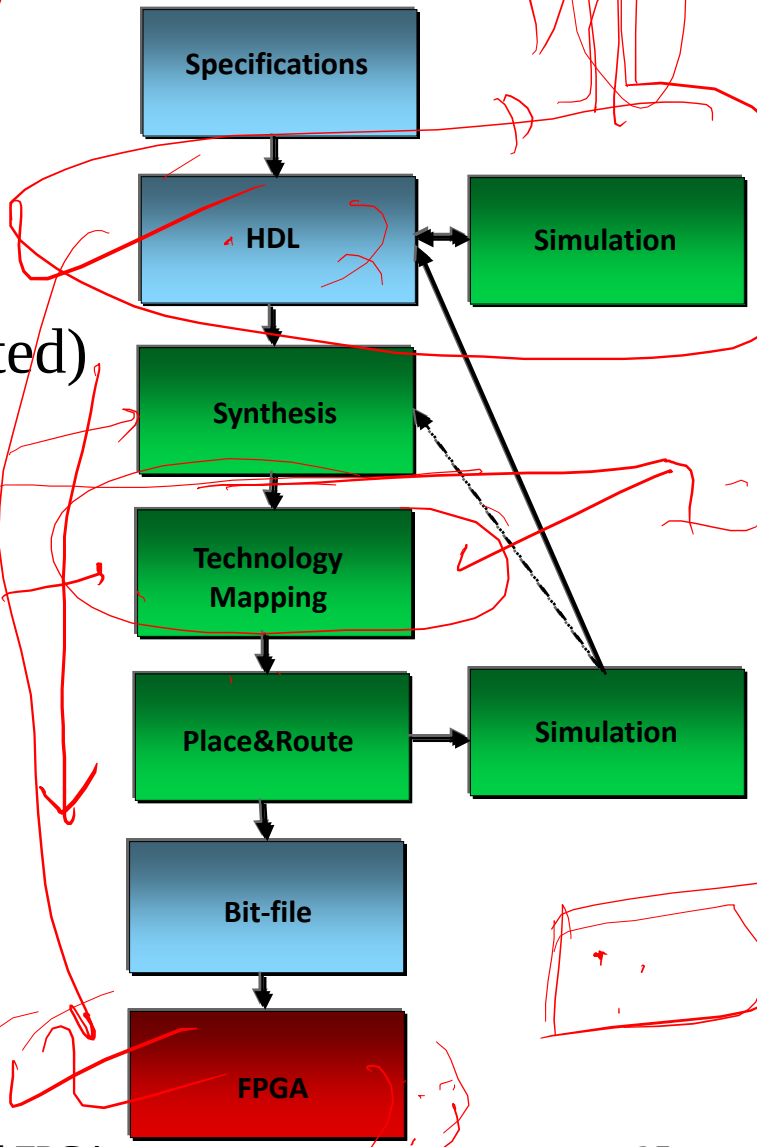
Programmable Interconnects



Snapshot from FPGA Editor

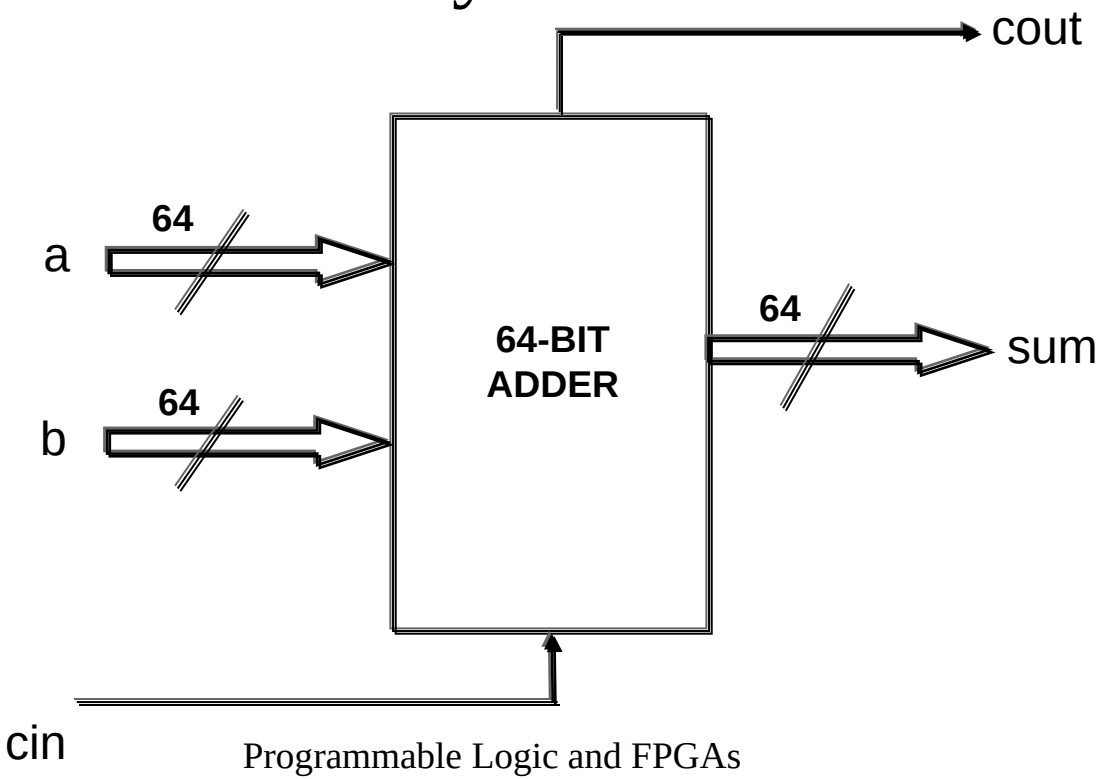
FPGA Design Flow

- 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

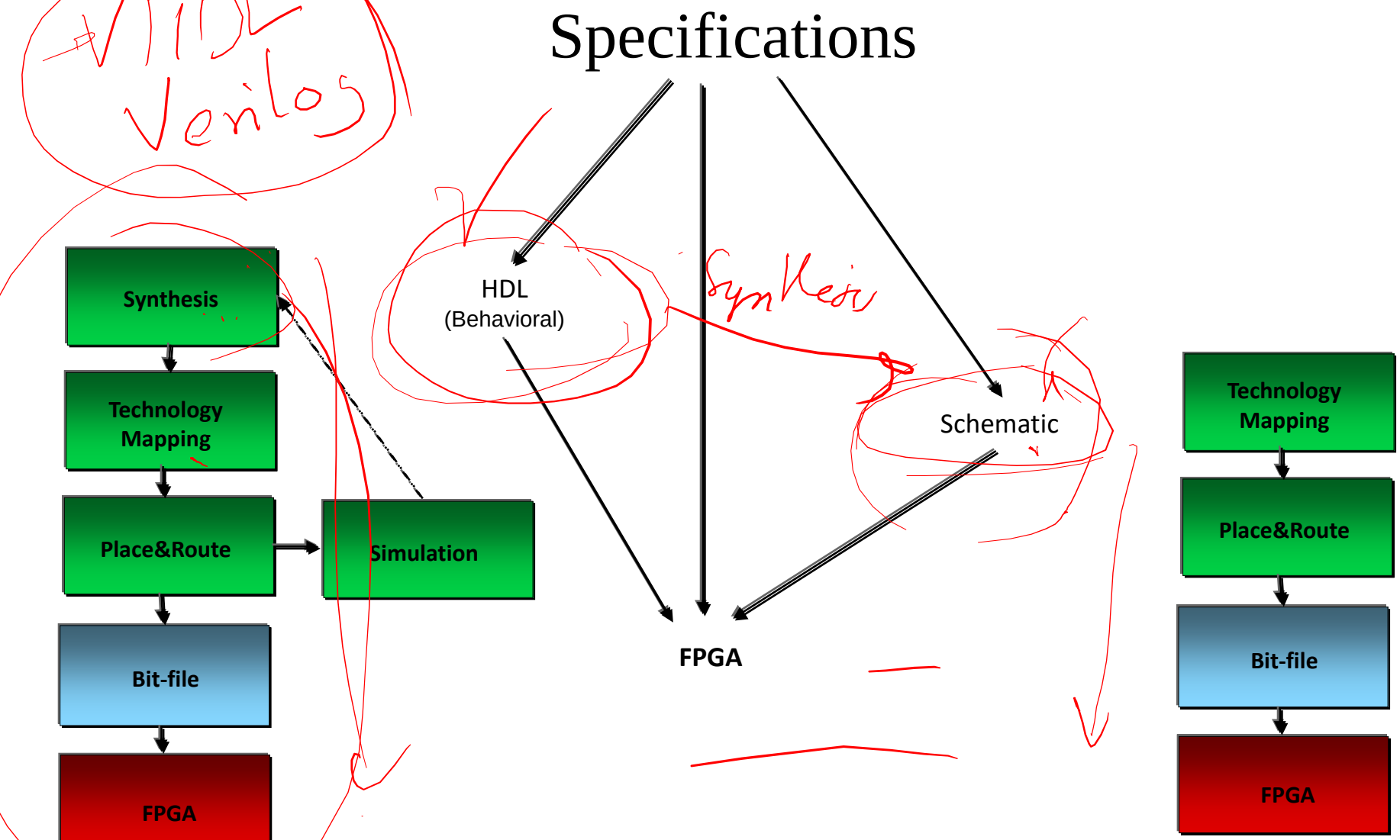


Specifications

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

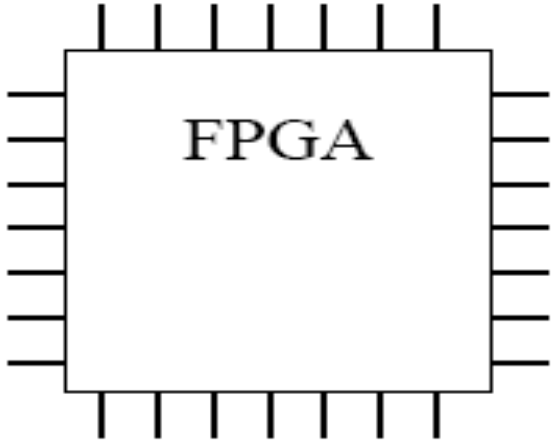


Specifications to FPGA

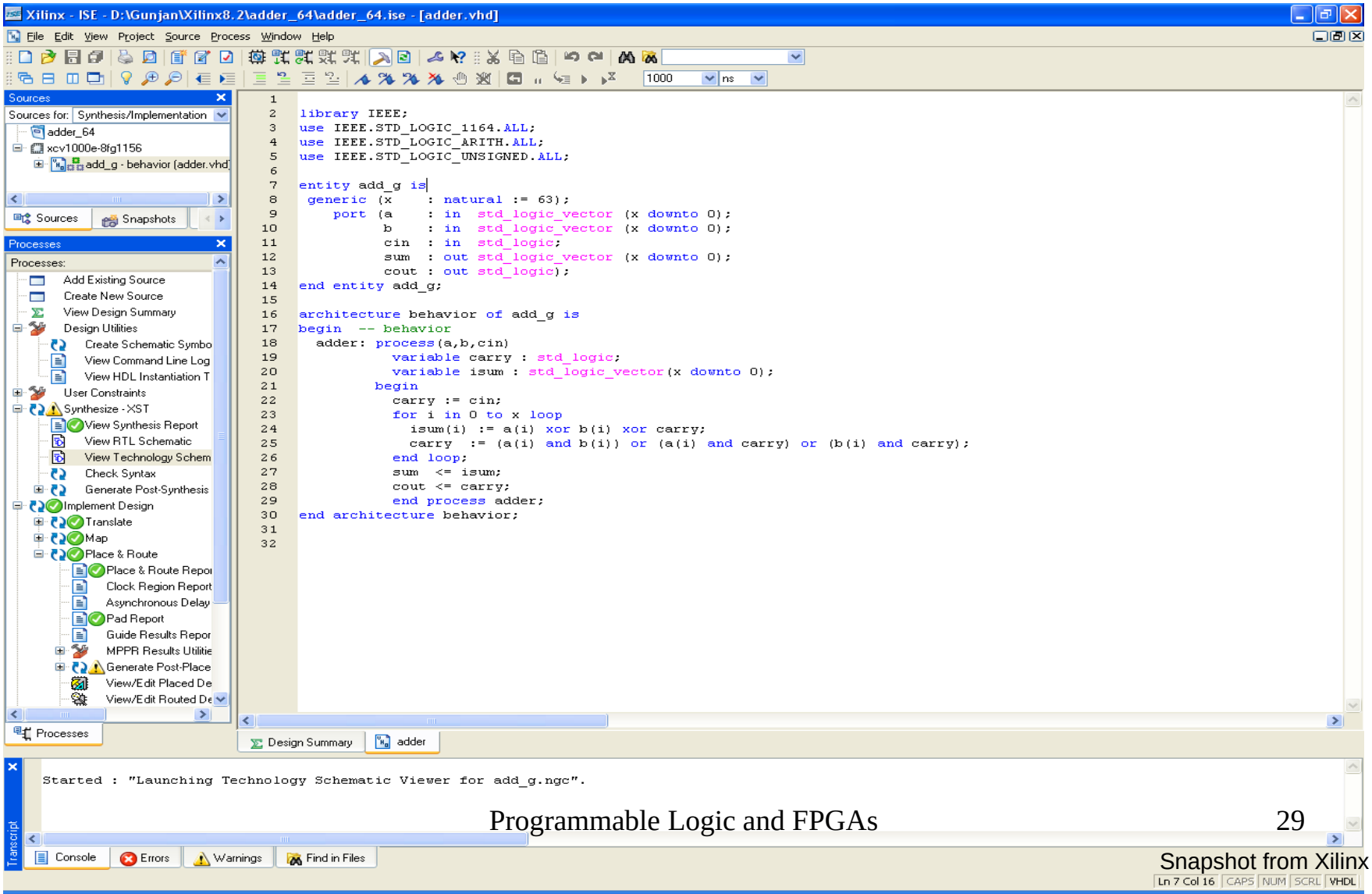


Specs (VHDL) to FPGA

```
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



Snapshot from Xilinx ISE

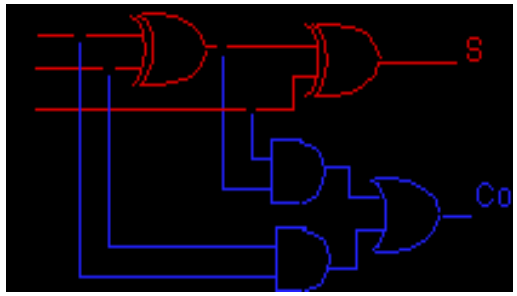
Synthesizing the Design

- 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;
```

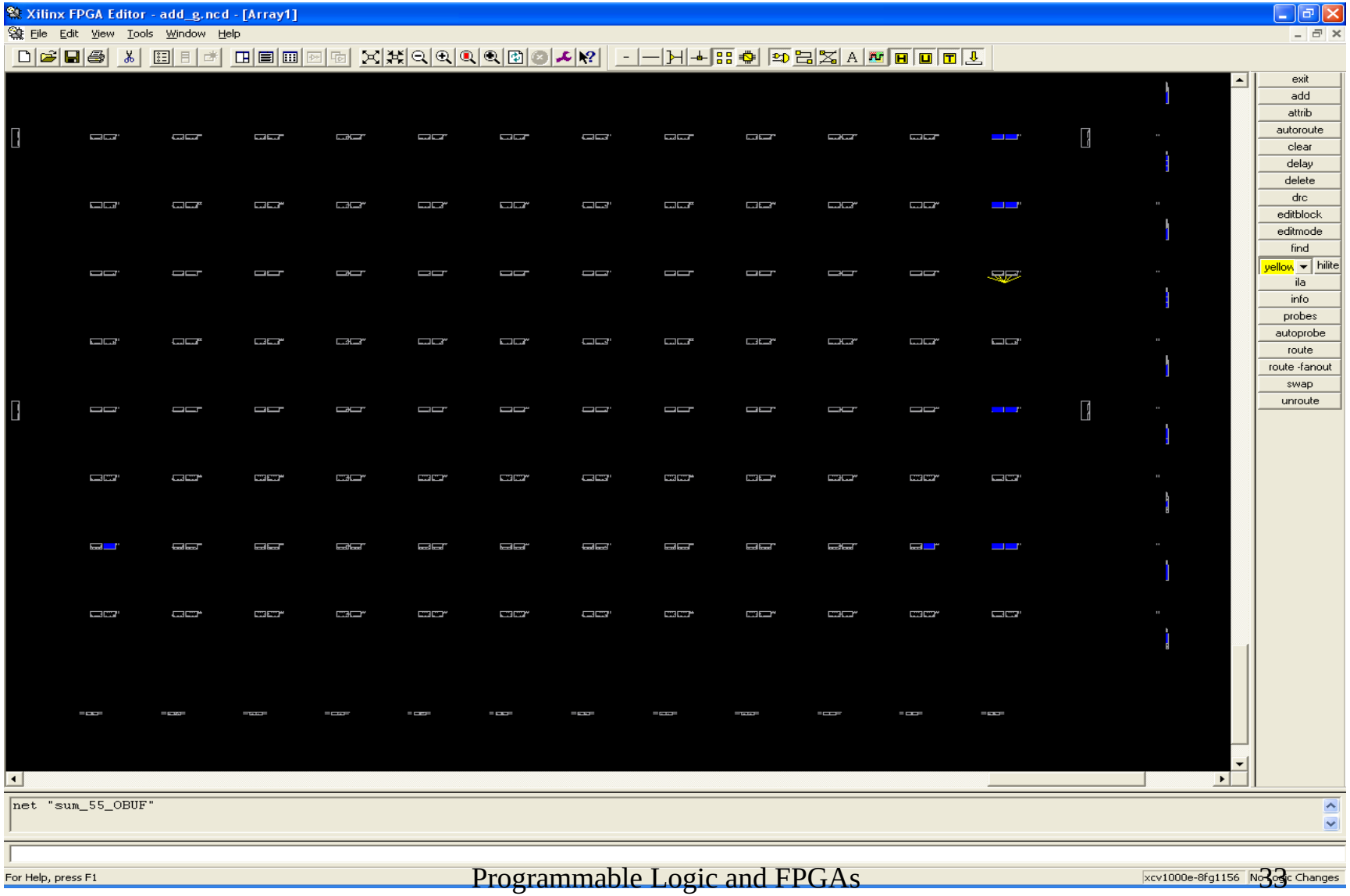
Synthesis



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.)

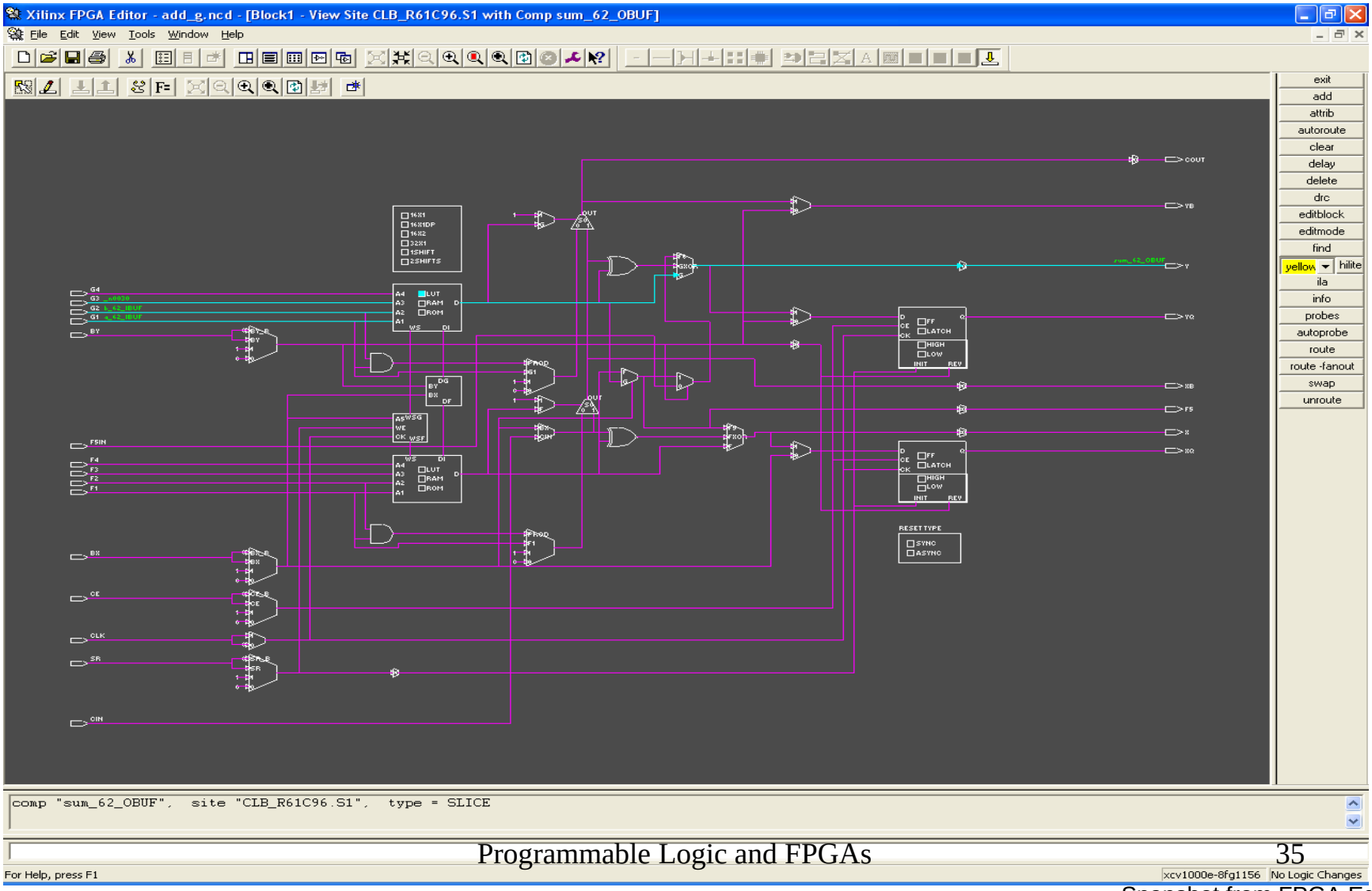


Snapshot from FPGA Editor

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.)

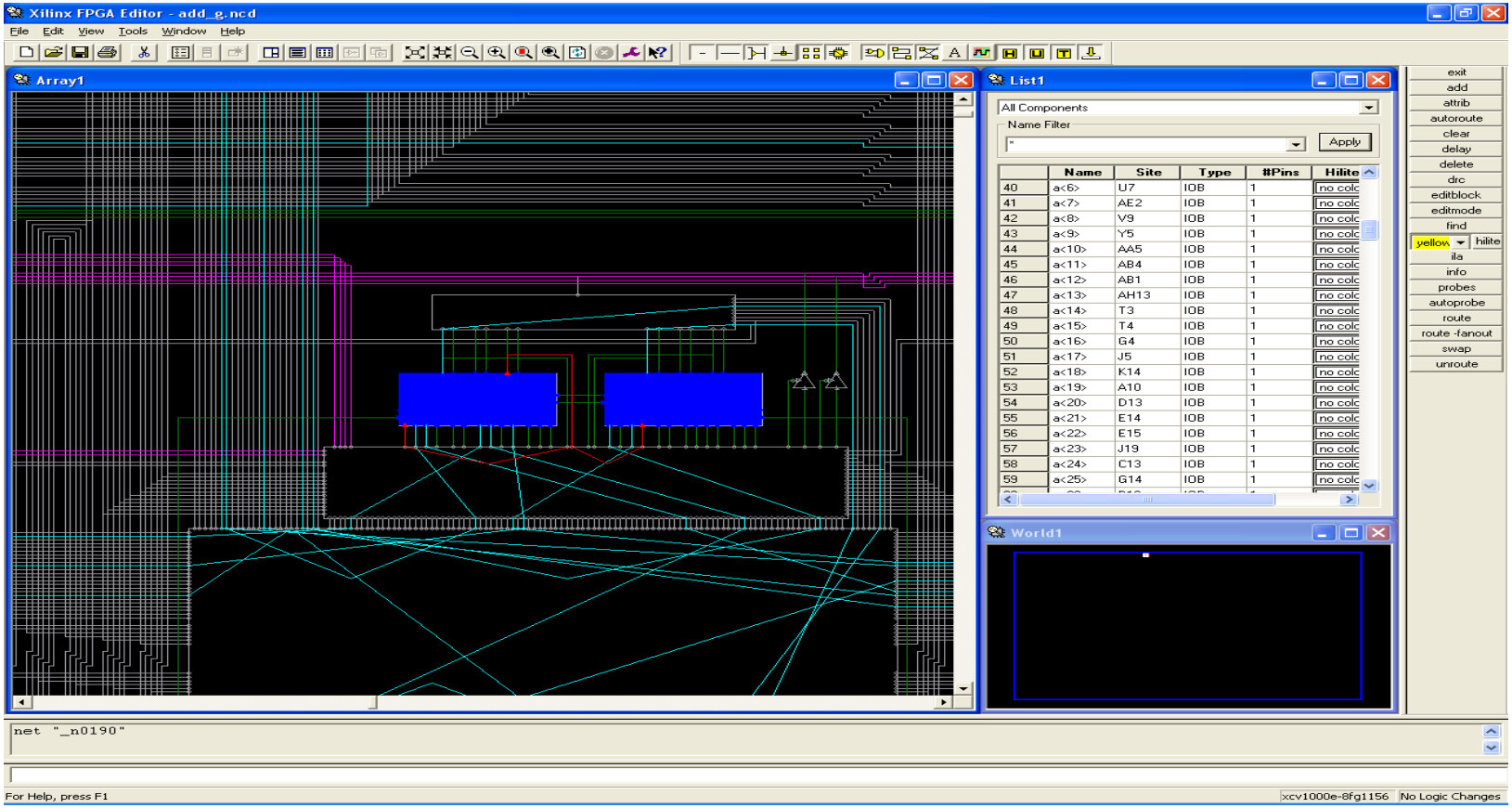


Snapshot from FPGA Editor

Routing the Design

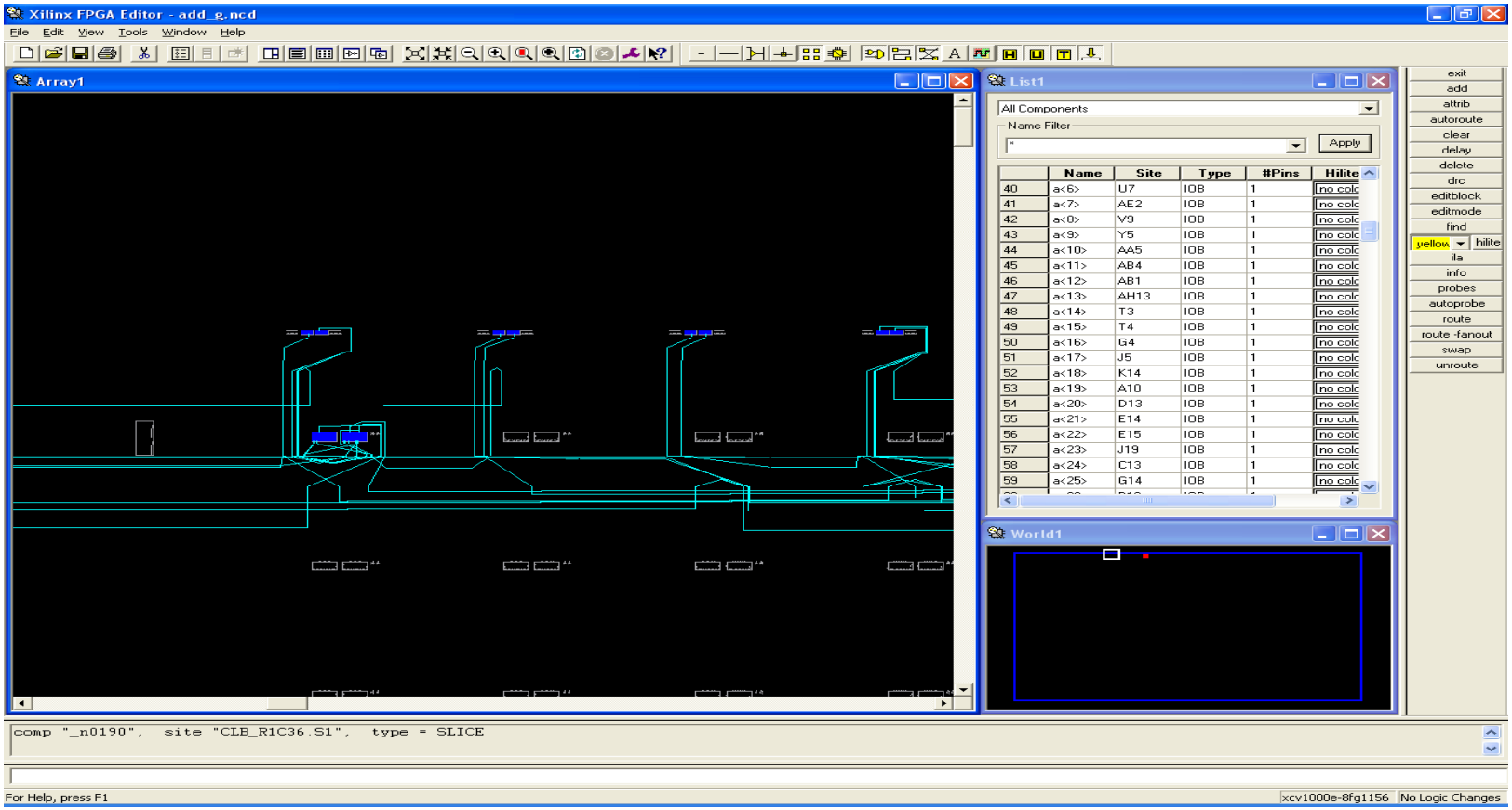
- Routing : The process of creating the desired interconnection of logic cells to make them perform the desired function.
- Routing follows after placement.
- Once logic resources have been assigned a location within the FPGA, we need to interconnect the logic resources using internal buses inside the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

Routing the Design (contd.)



Snapshot from FPGA Editor

Routing the Design (contd.)

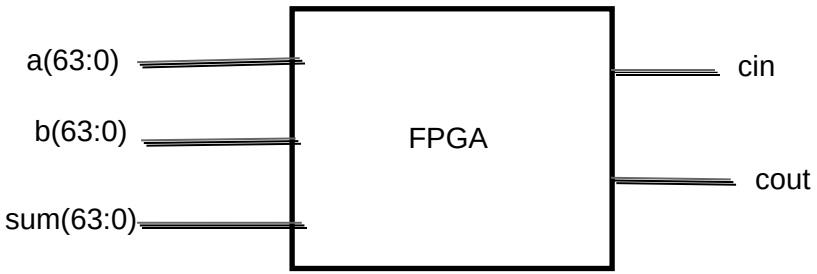


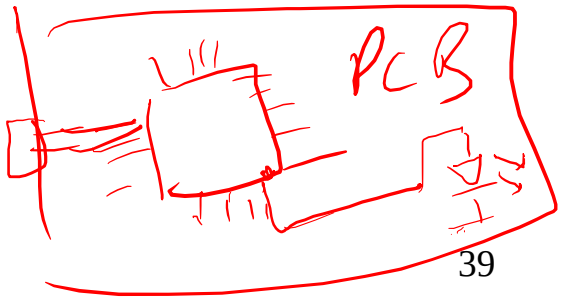
Snapshot from FPGA Editor

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.)

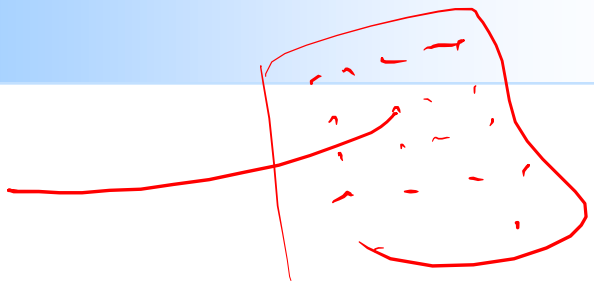
- A file called a UCF (User Constraint File) is used to define which pin will be connected to a particular input or output.

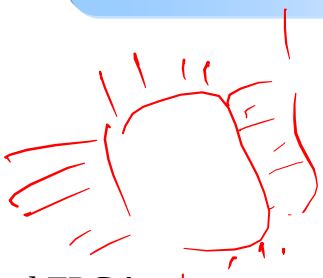
- Within Xilinx Project manager, the “assign package pin” function can be used to easily define input and output pin location.

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;
```

```
NET "cin"  LOC = "T9";
NET "cout" LOC = "M13" ;
```

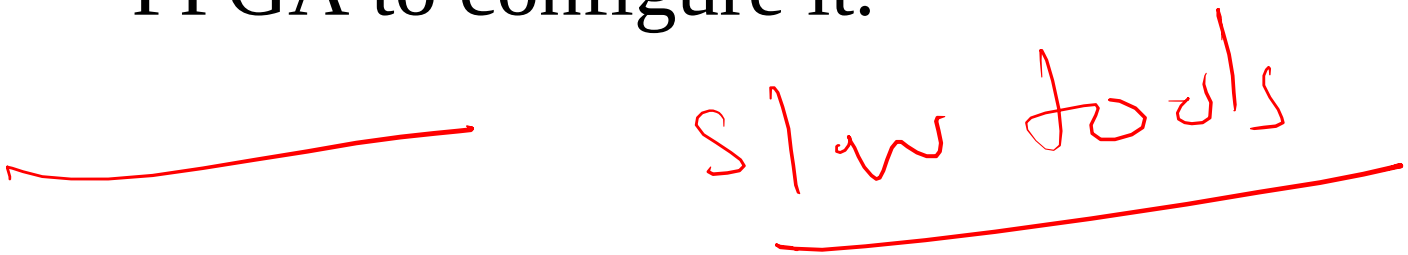




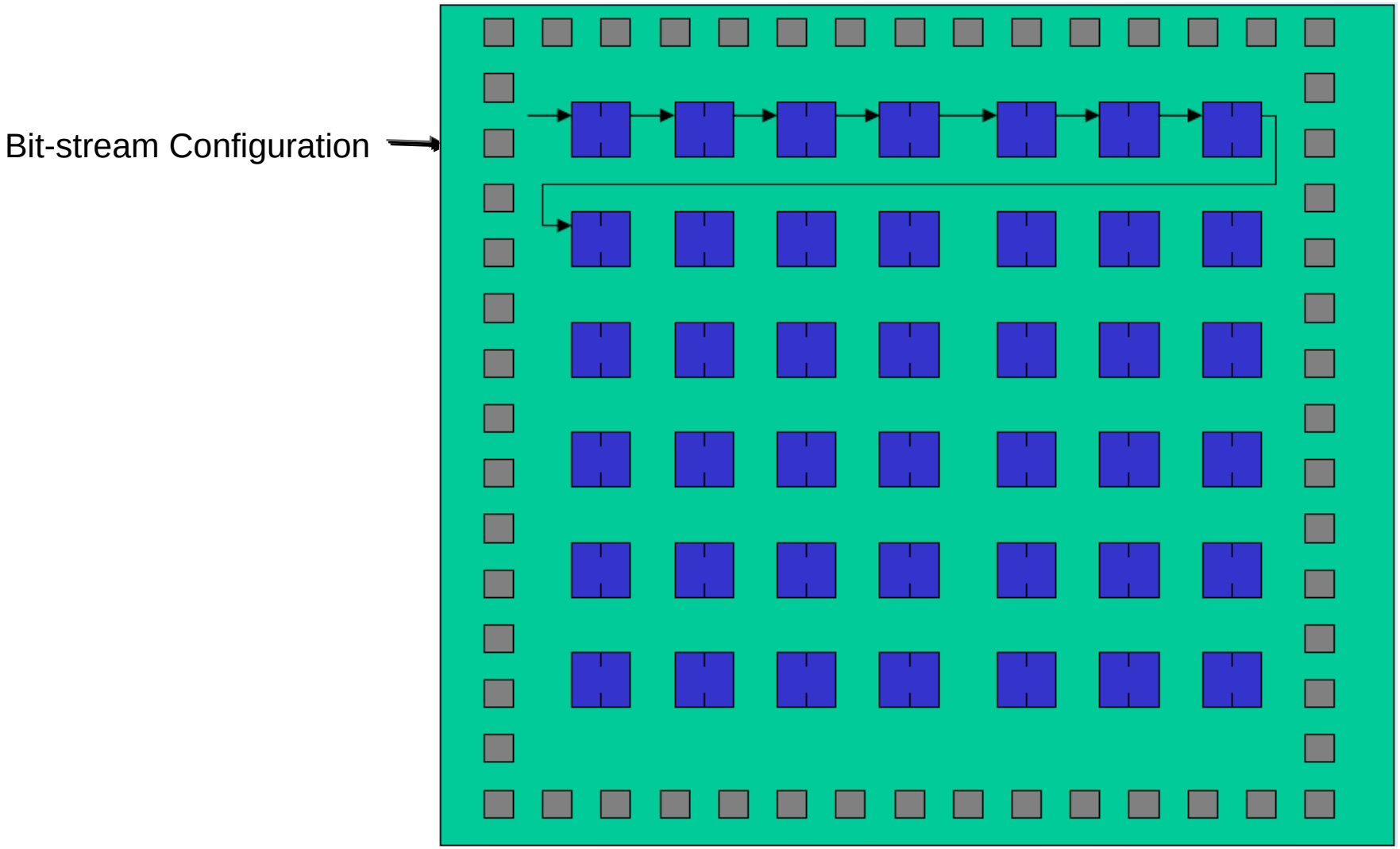
Convert Design into Bit-stream

- Always done using automated tools.

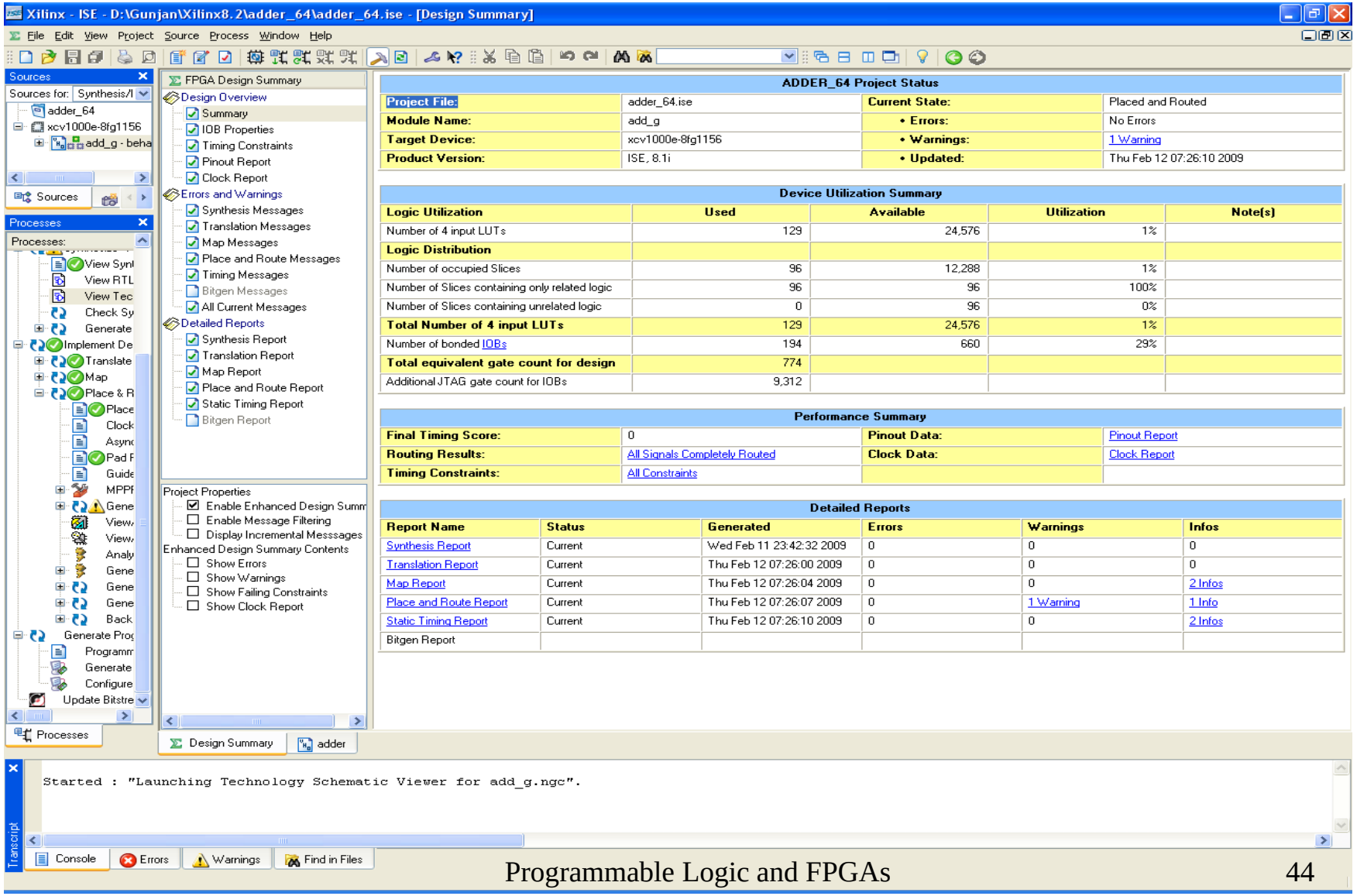
- The placed and routed design is converted into a bit-stream that is downloaded into the FPGA to configure it.



Design to Bit-stream (contd.)

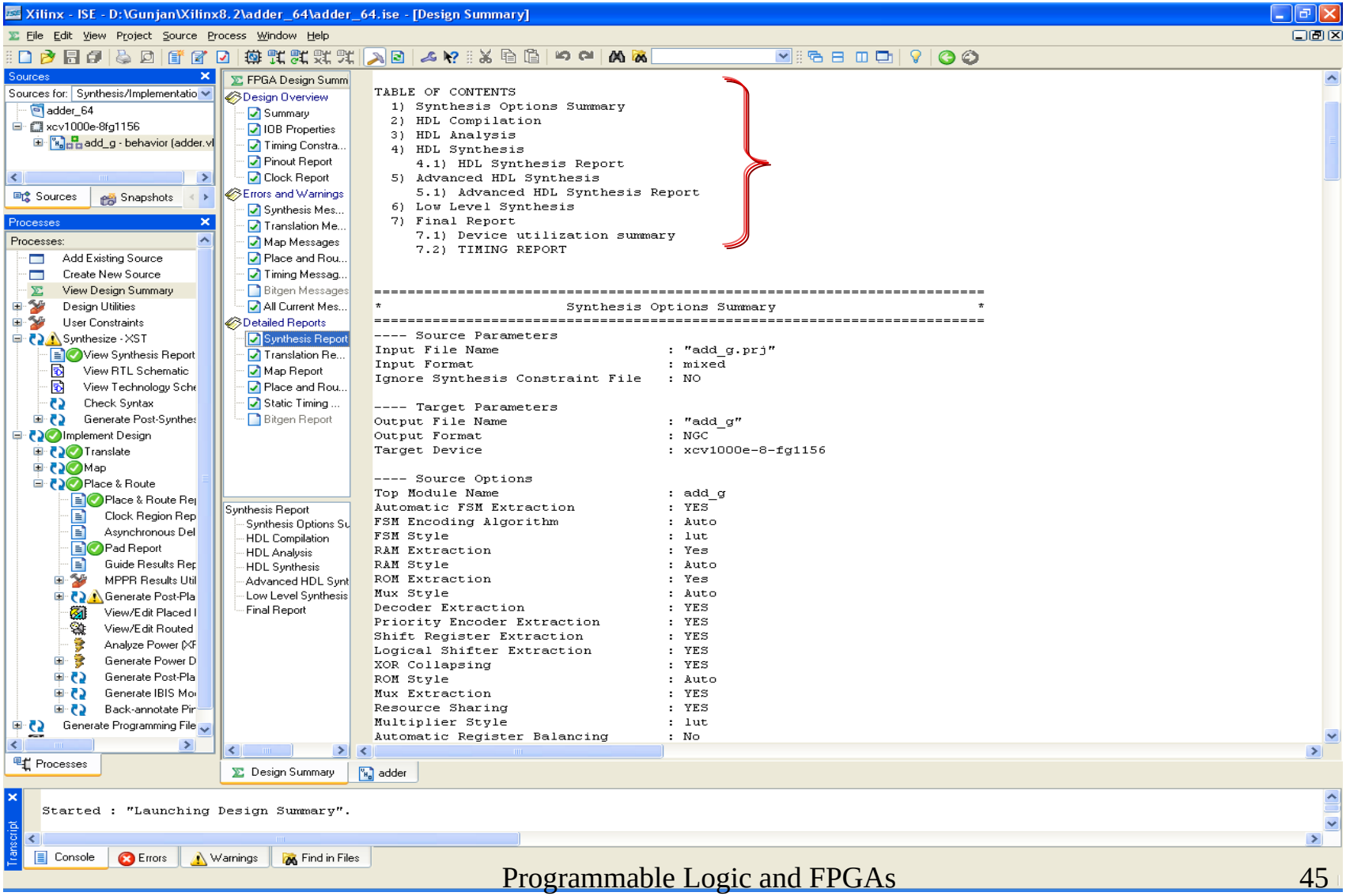


Design Summary



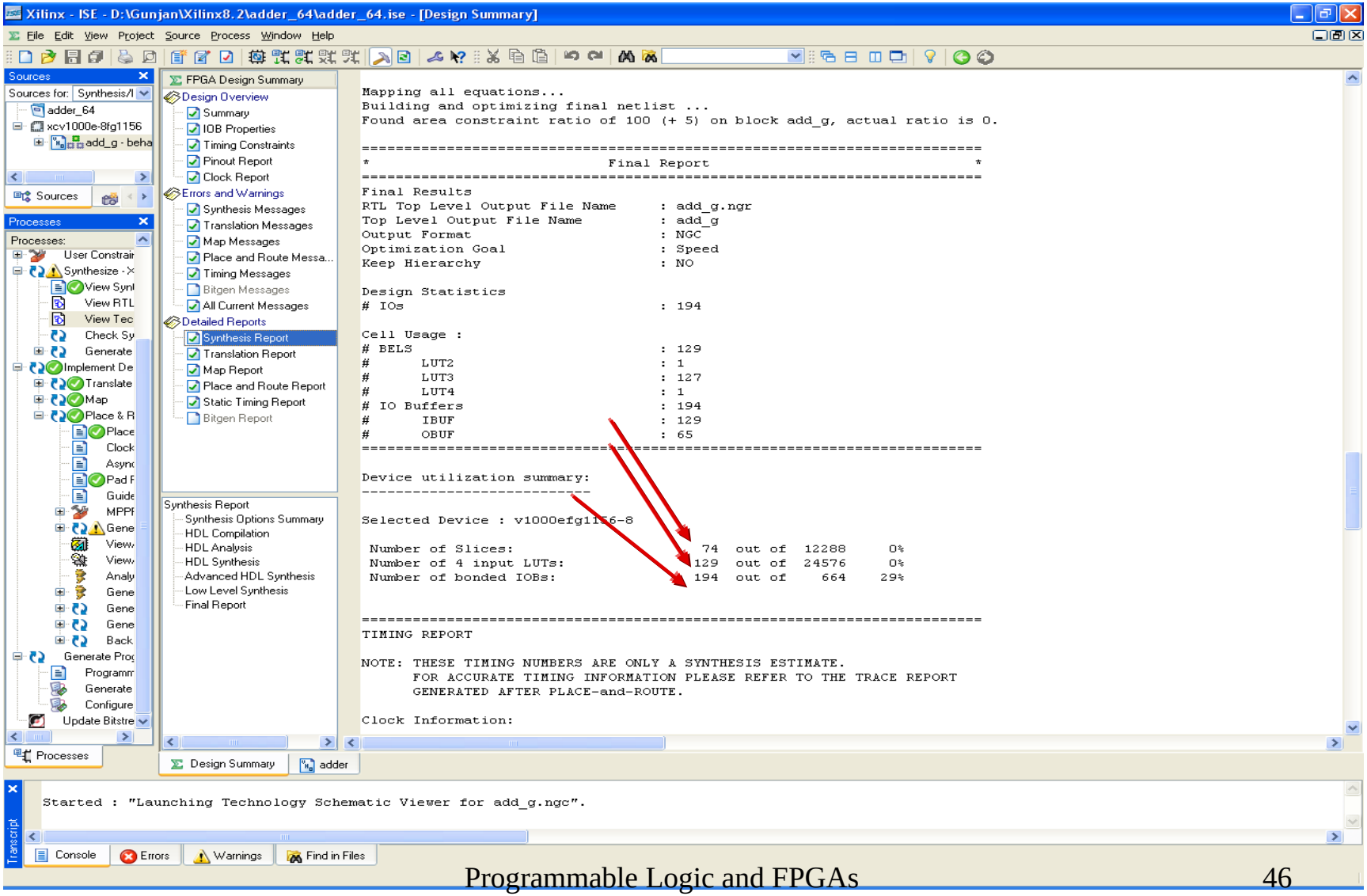
Snapshot from Xilinx ISE

Synthesis Report



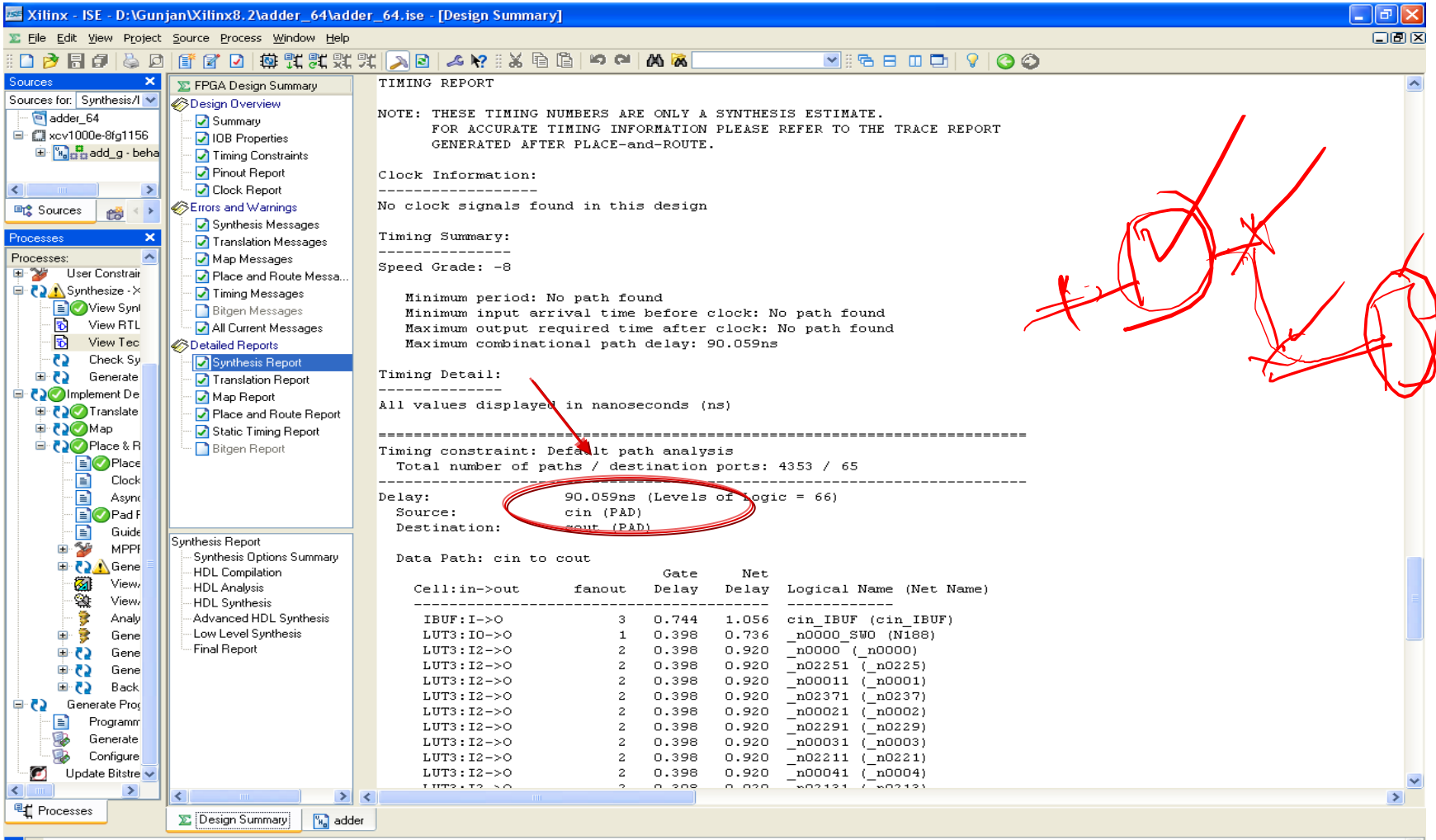
Snapshot from Xilinx ISE

Synthesis Report – Device Utilization Summary



Snapshot from Xilinx ISE

Synthesis Report – Timing Report

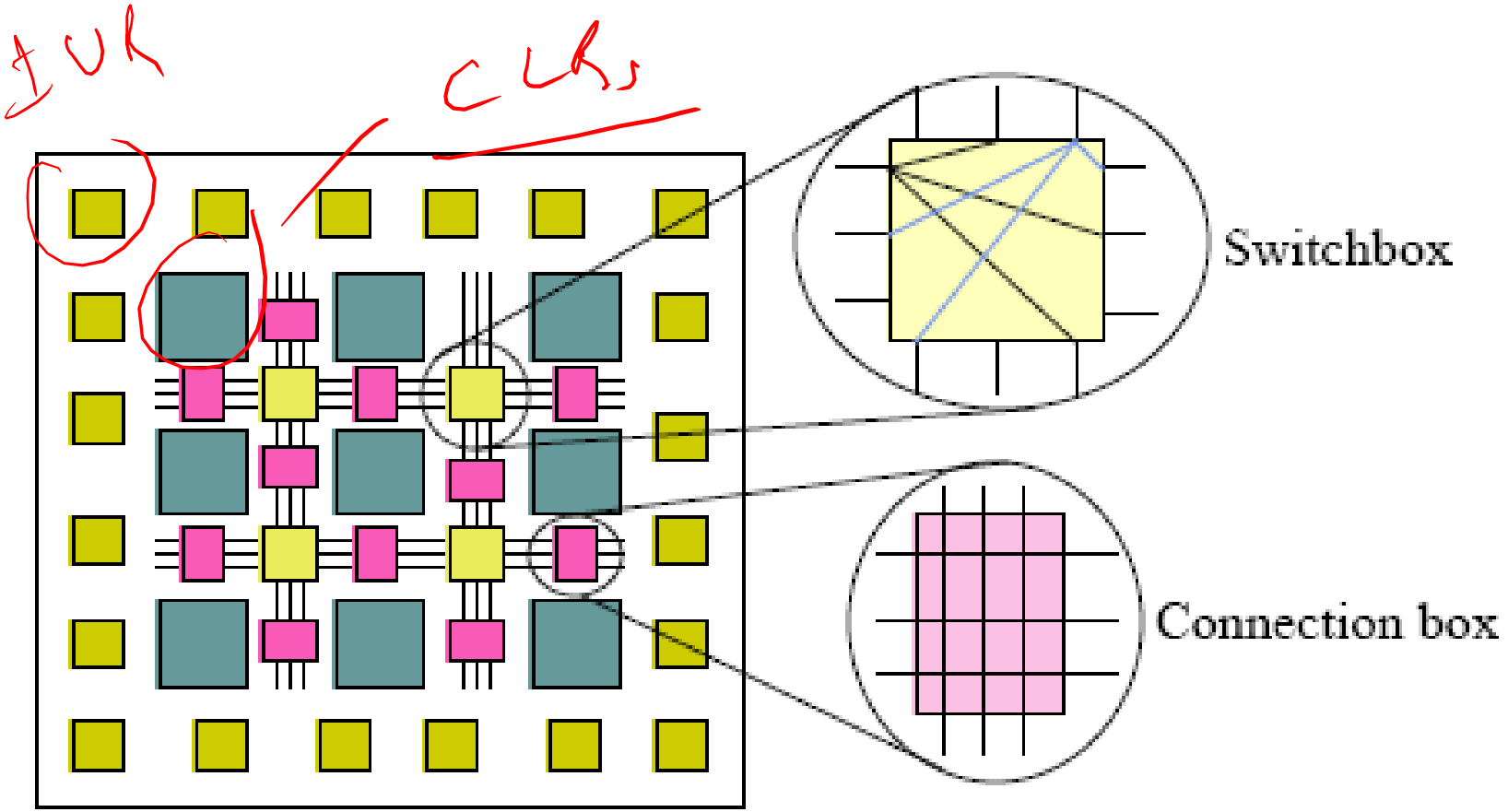


Snapshot from Xilinx ISE

Lecture 34: FPGA Architecture

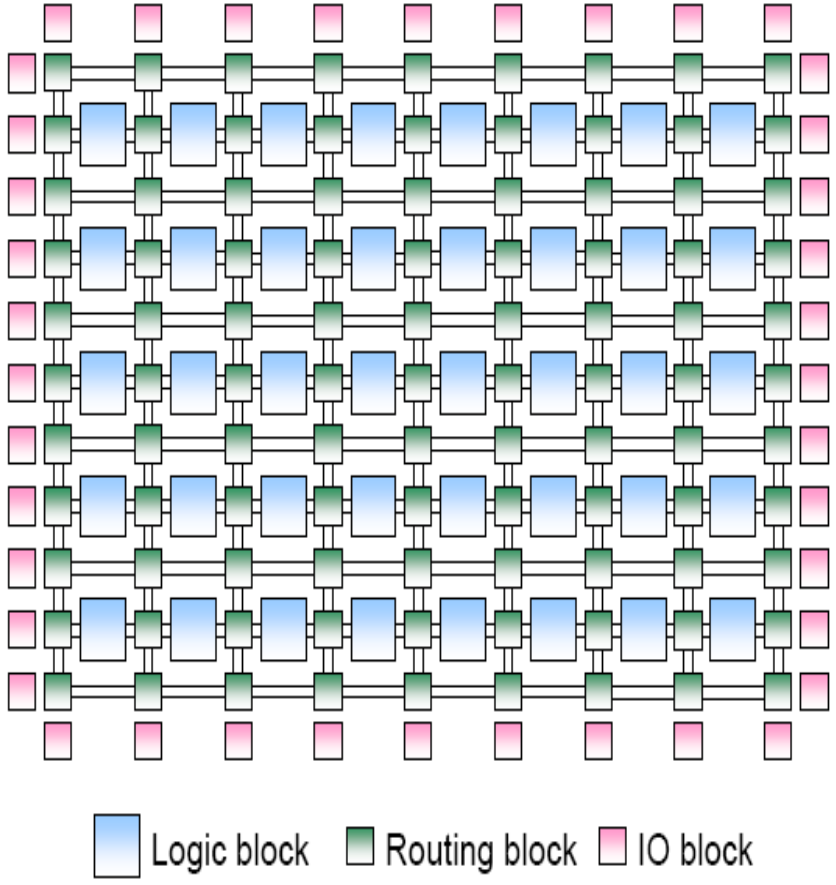
M. Balakrishnan

Generic 2D FPGA

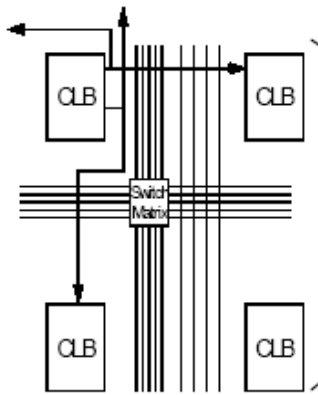


FPGA Architecture

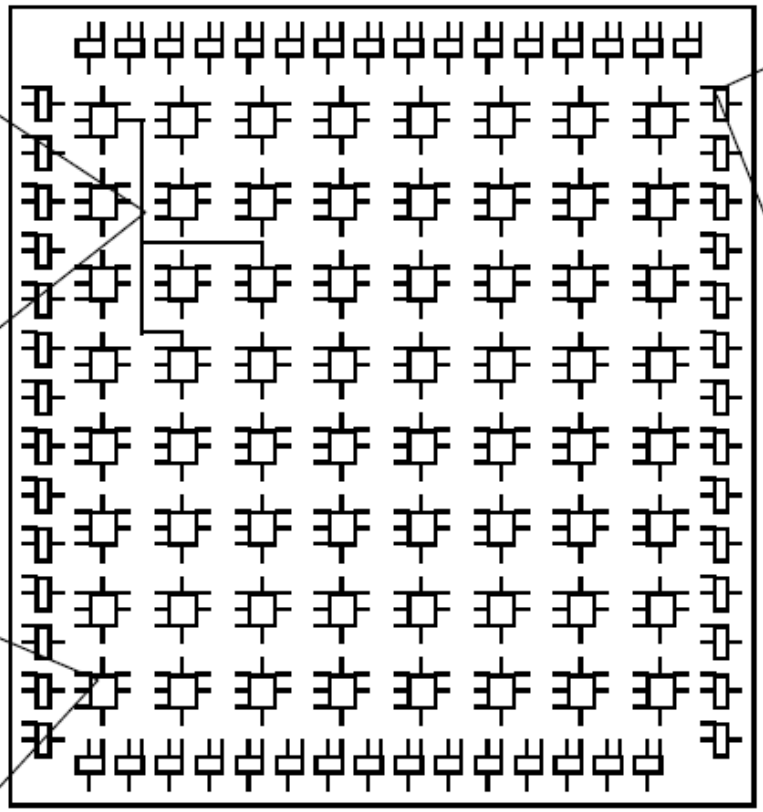
- An FPGA is an array of programmable logic elements that can be connected to inputs or outputs.



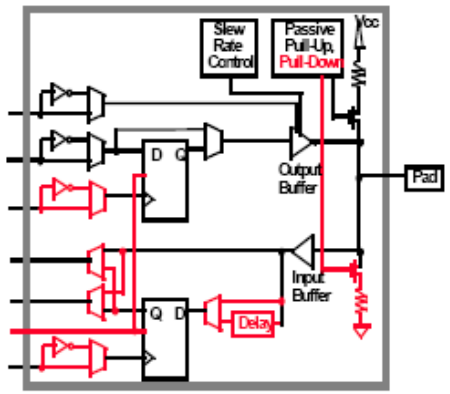
SRAM Based FPGA - XILINX



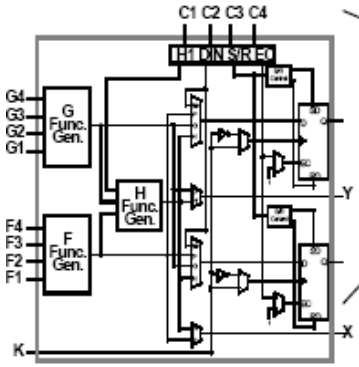
Programmable Interconnect



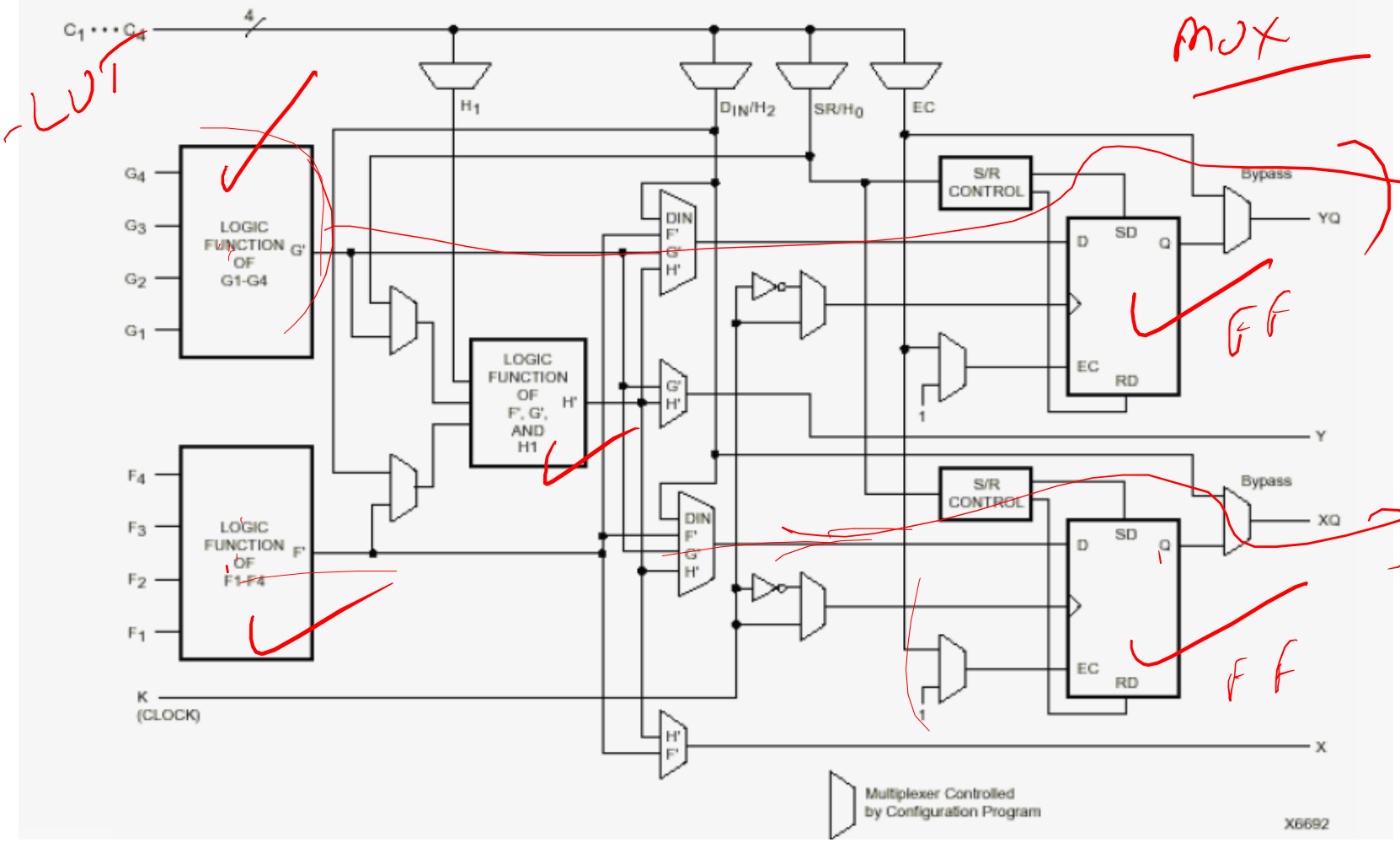
Configurable Logic Blocks (CLBs)

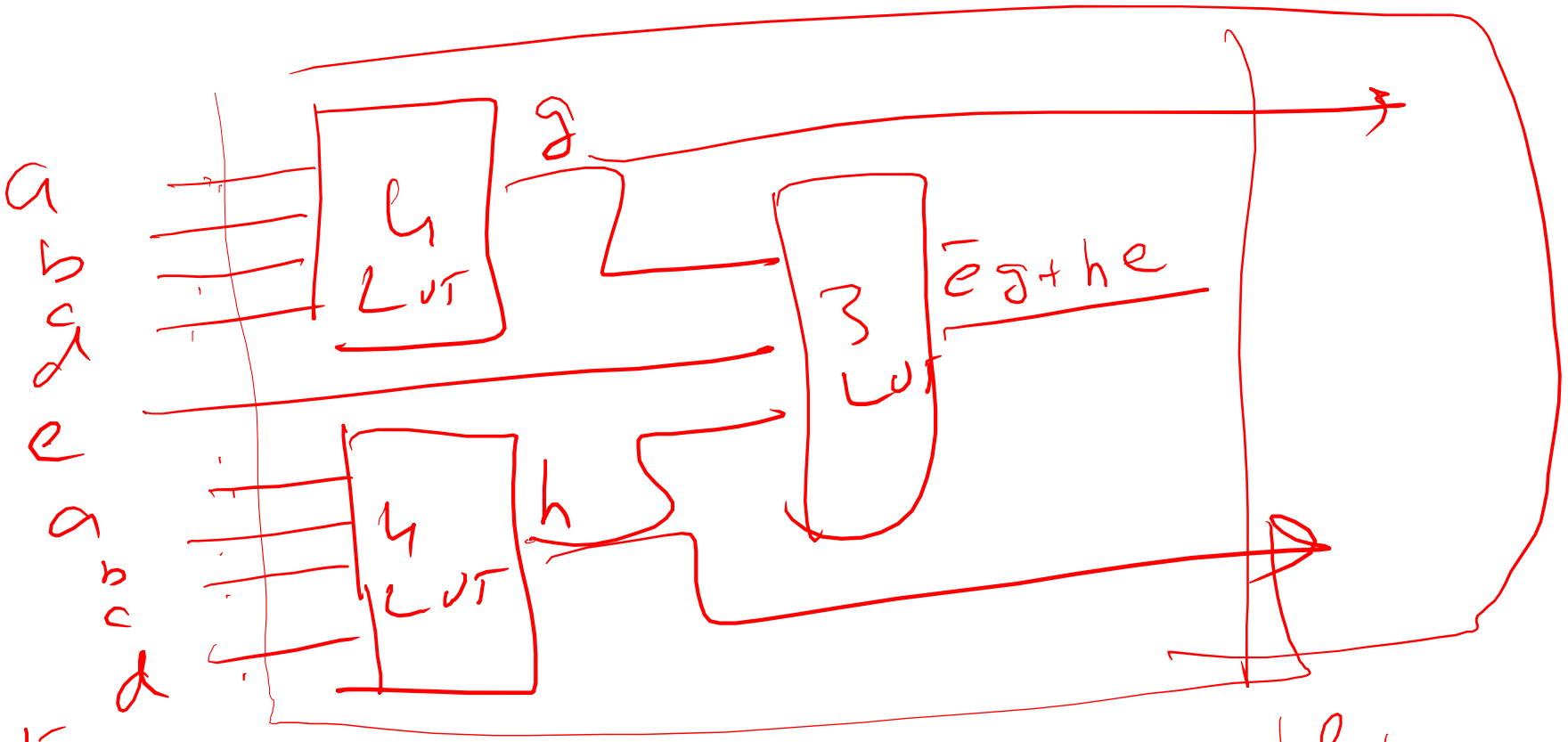


I/O Blocks (IOBs)



Xilinx 4000 CLB





9 inputs

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

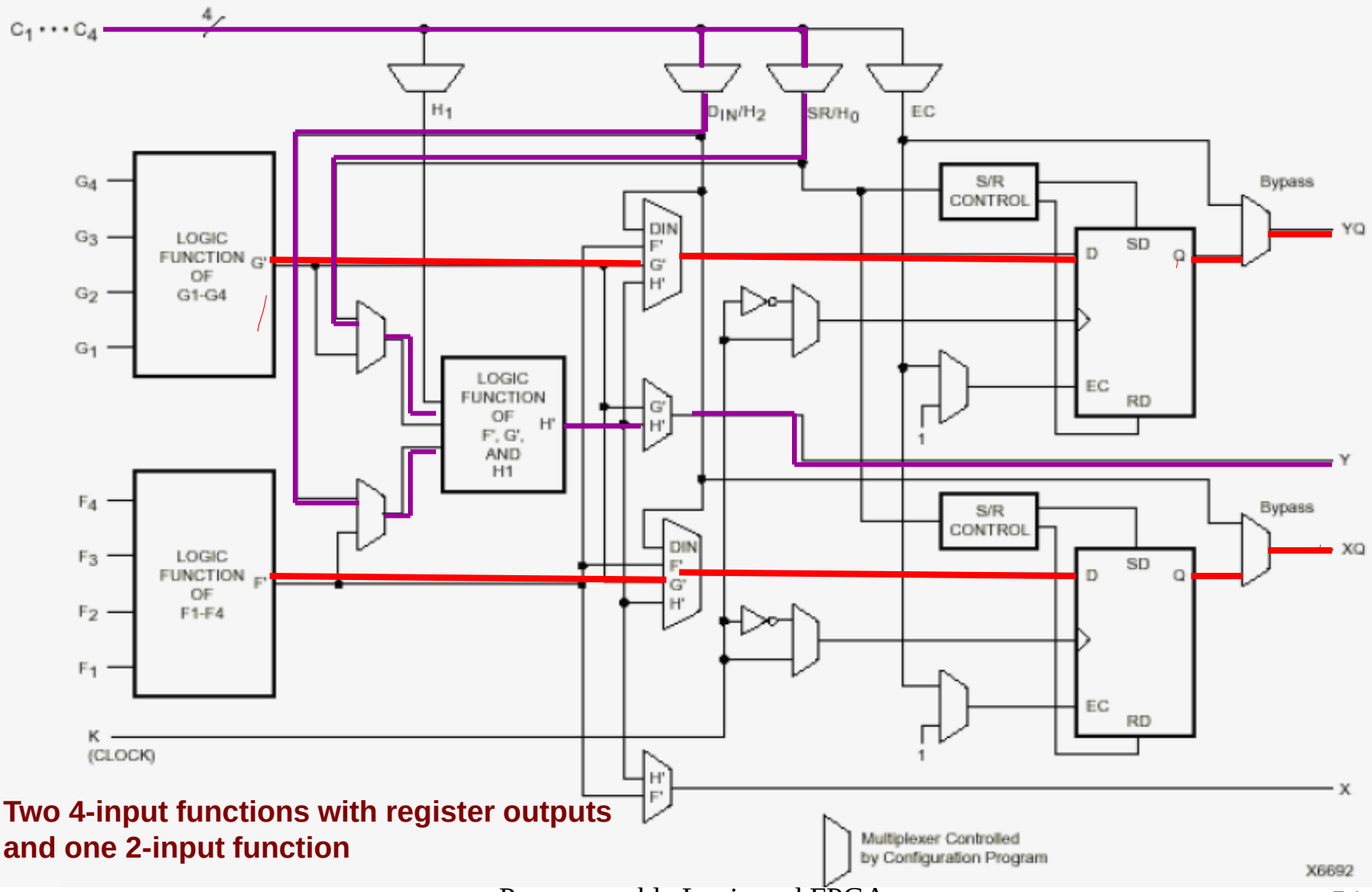
5 - variables

4 - variables

Answer

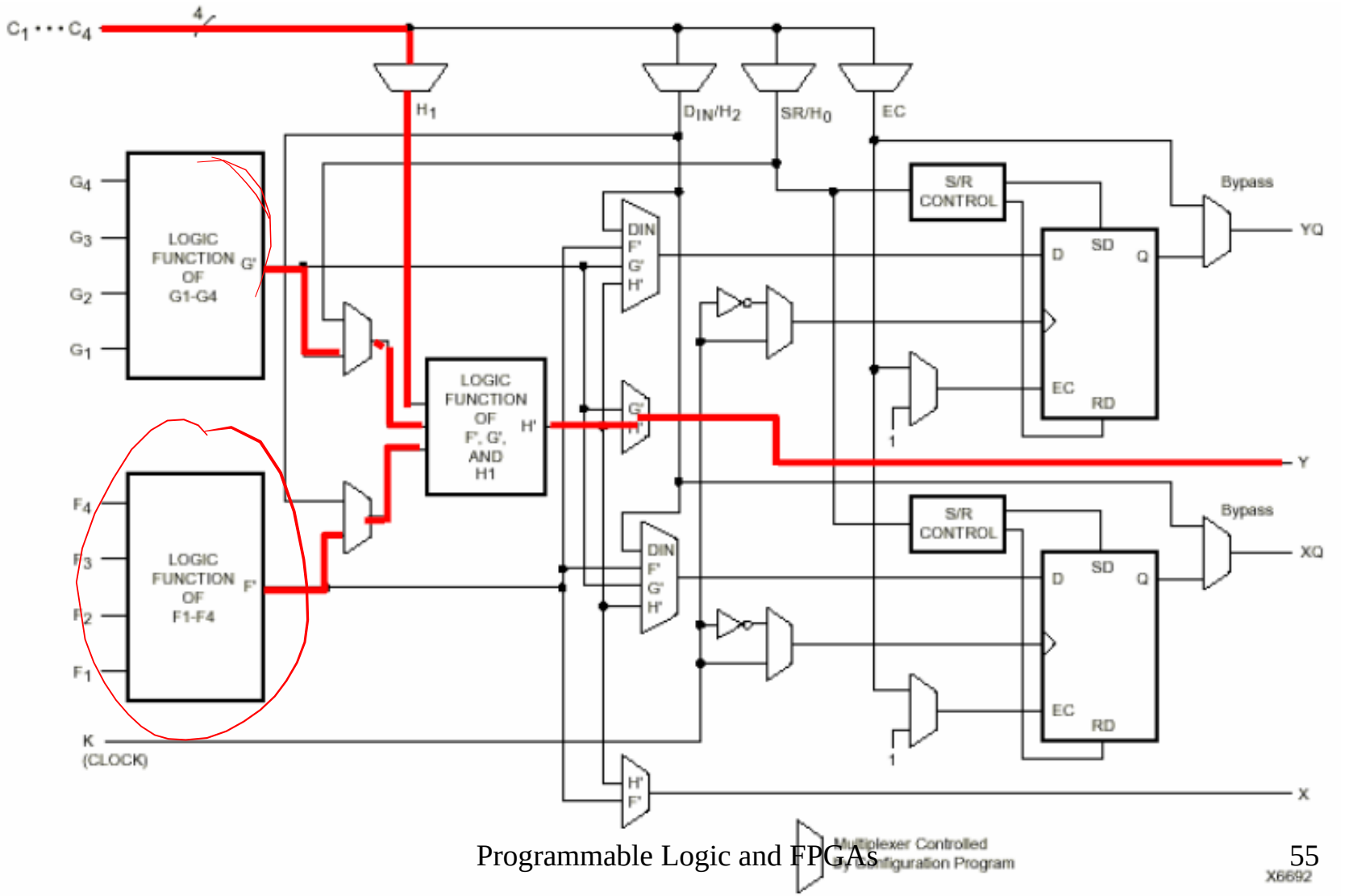
$f(a,b,c,d,e) = g(a,b,c,d)\bar{e} + h(a,b,c,d)e$

Implementing Combinational Logic

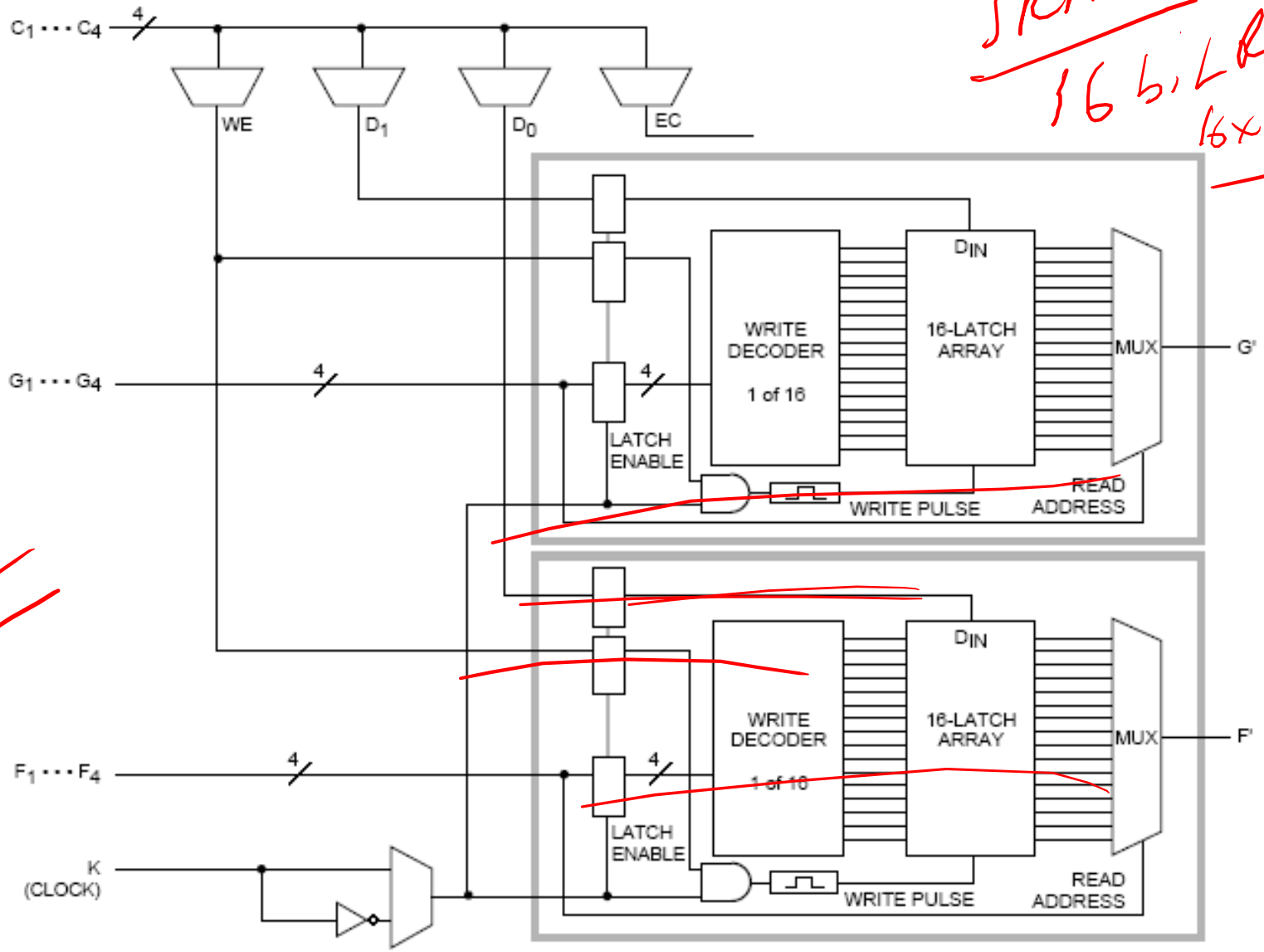


Two 4-input functions with register outputs and one 2-input function

5 input Function

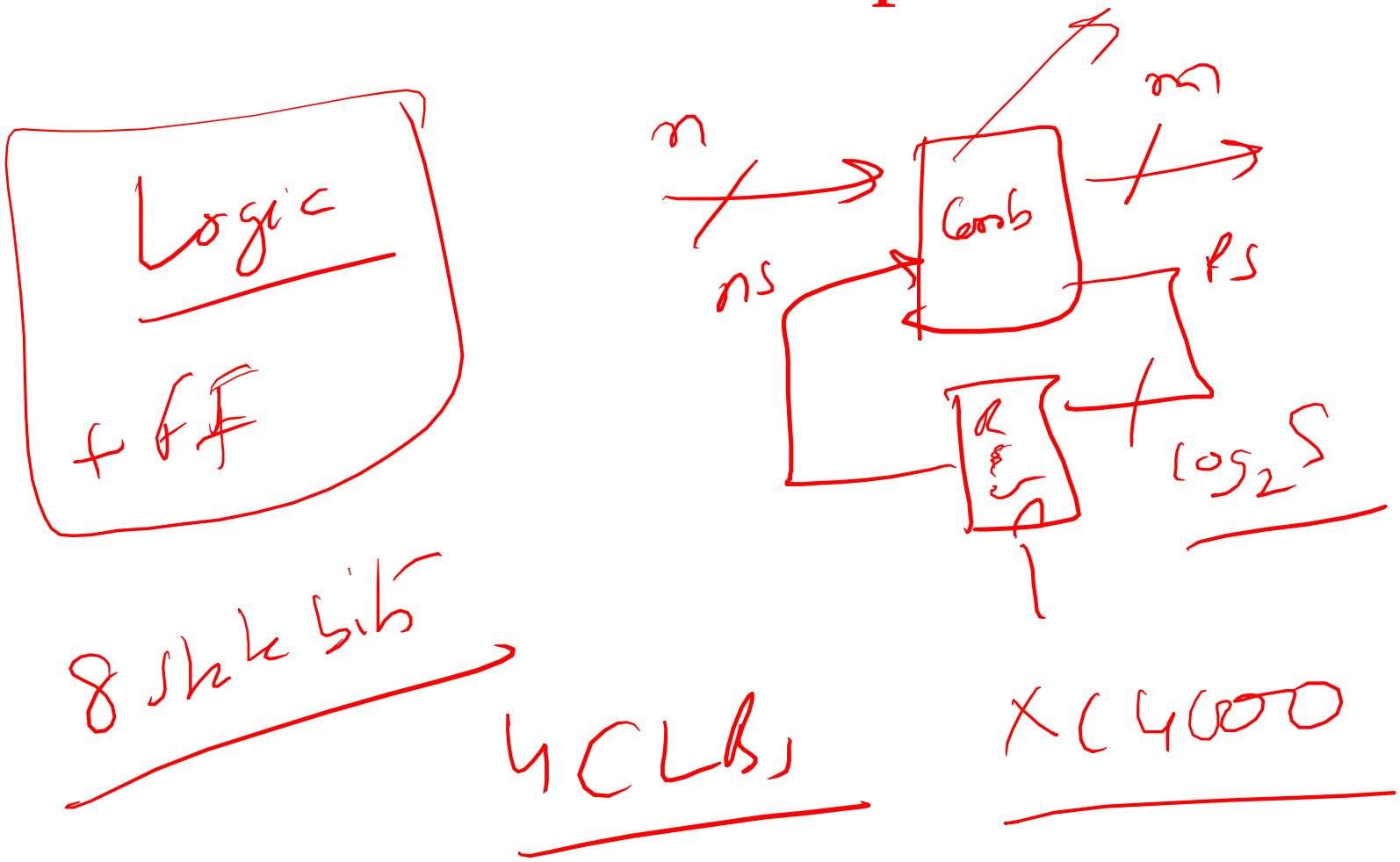


Single Port RAM

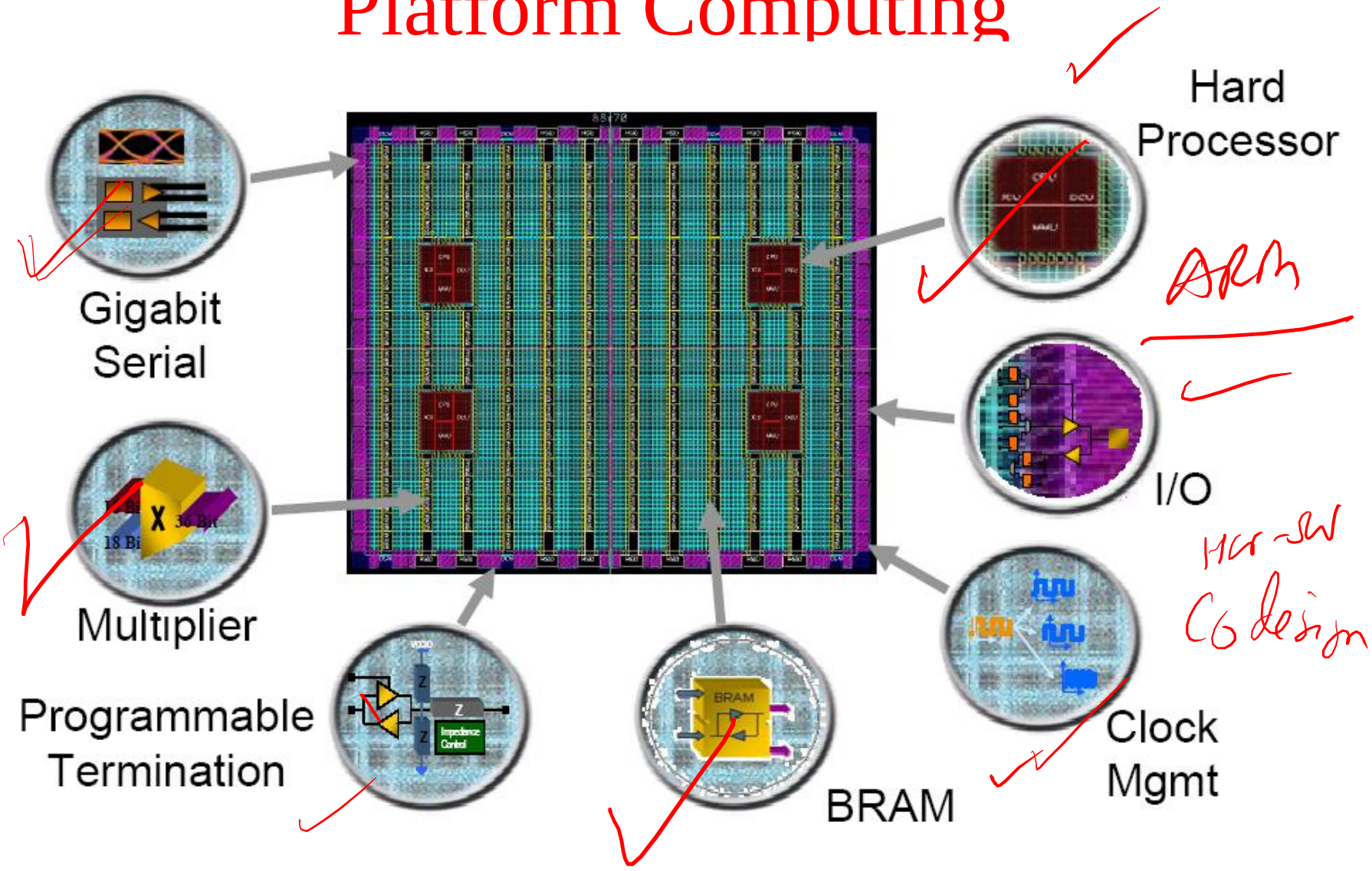


http://www.xilinx.com/bvdocs/publications/4000.pdf

Function Decomposition



Platform Computing

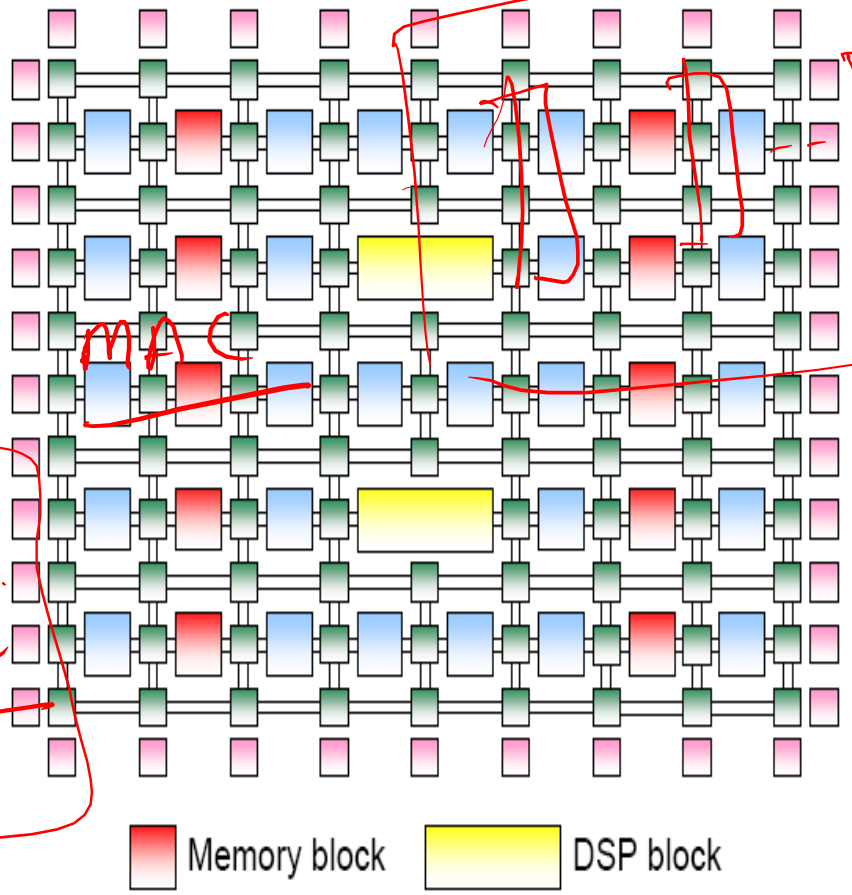


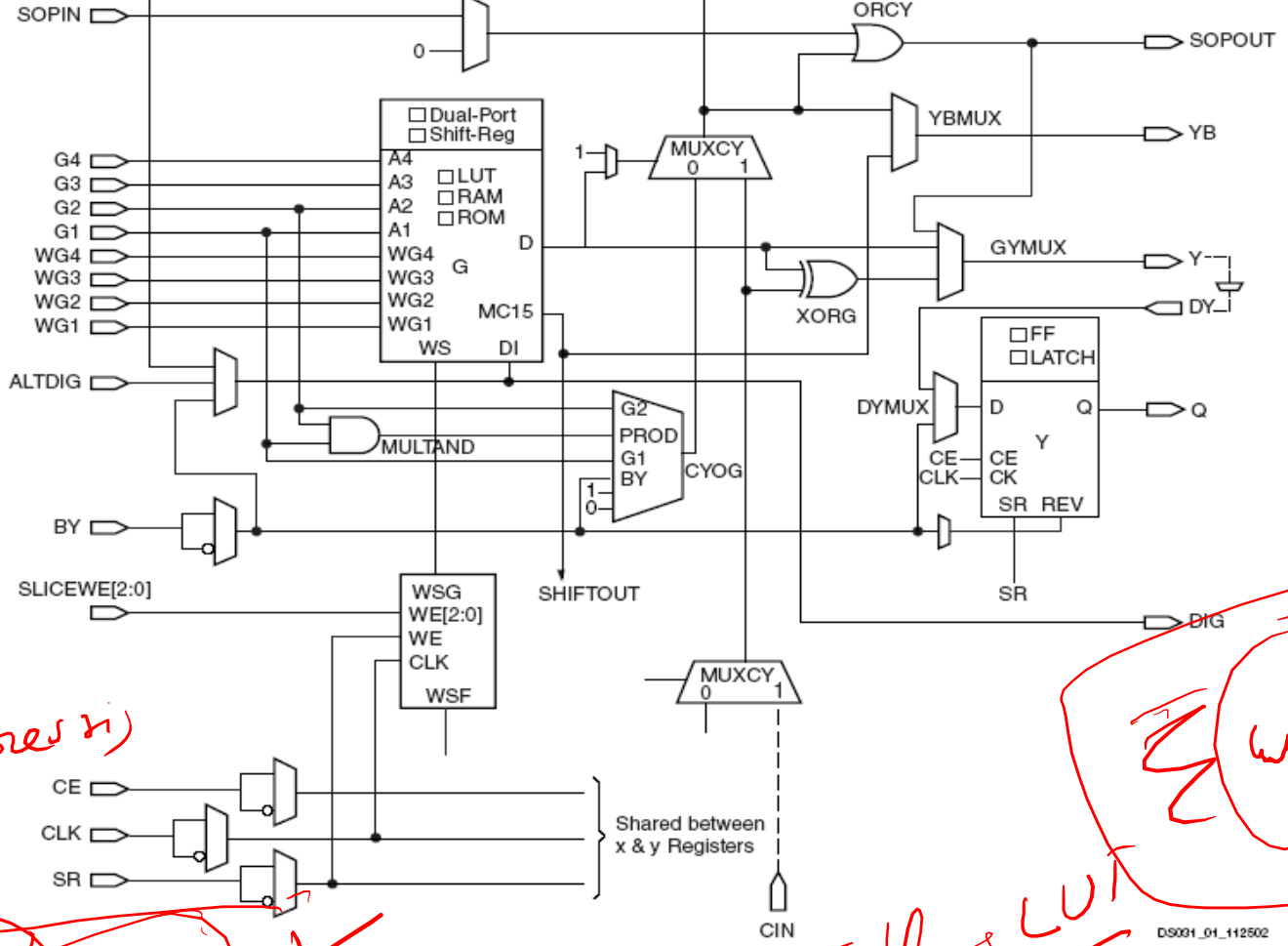
Courtesy of David B. Parlour, ISSCC 2004 Tutorial,
"The Reality and Promise of Reconfigurable Computing in Digital Signal Processing"

Modern FPGA Fabric

Programmable fabric

- Memory blocks
- DSP blocks





Signal Processing

$\sum w_i x_i$

LUT

RAM

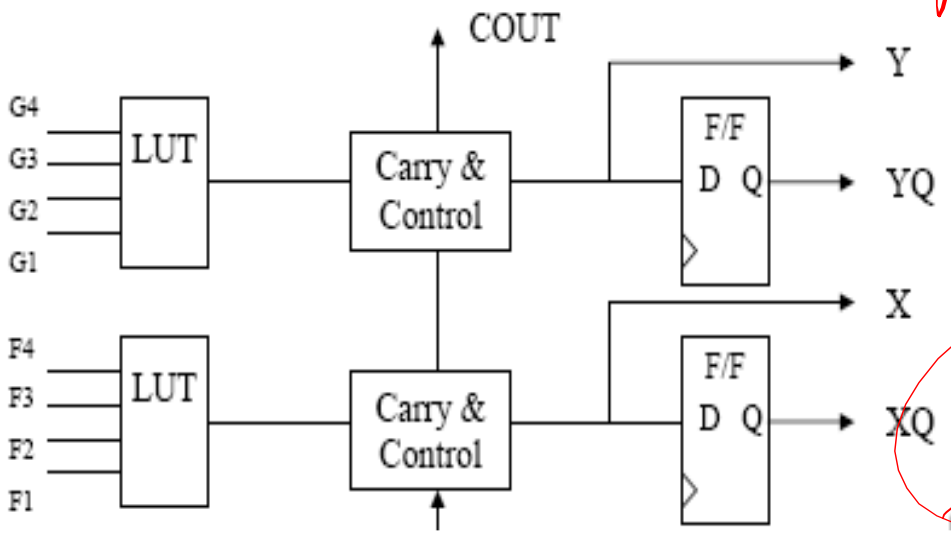
ROM

Truth Table → LUT

Configuration bit-file

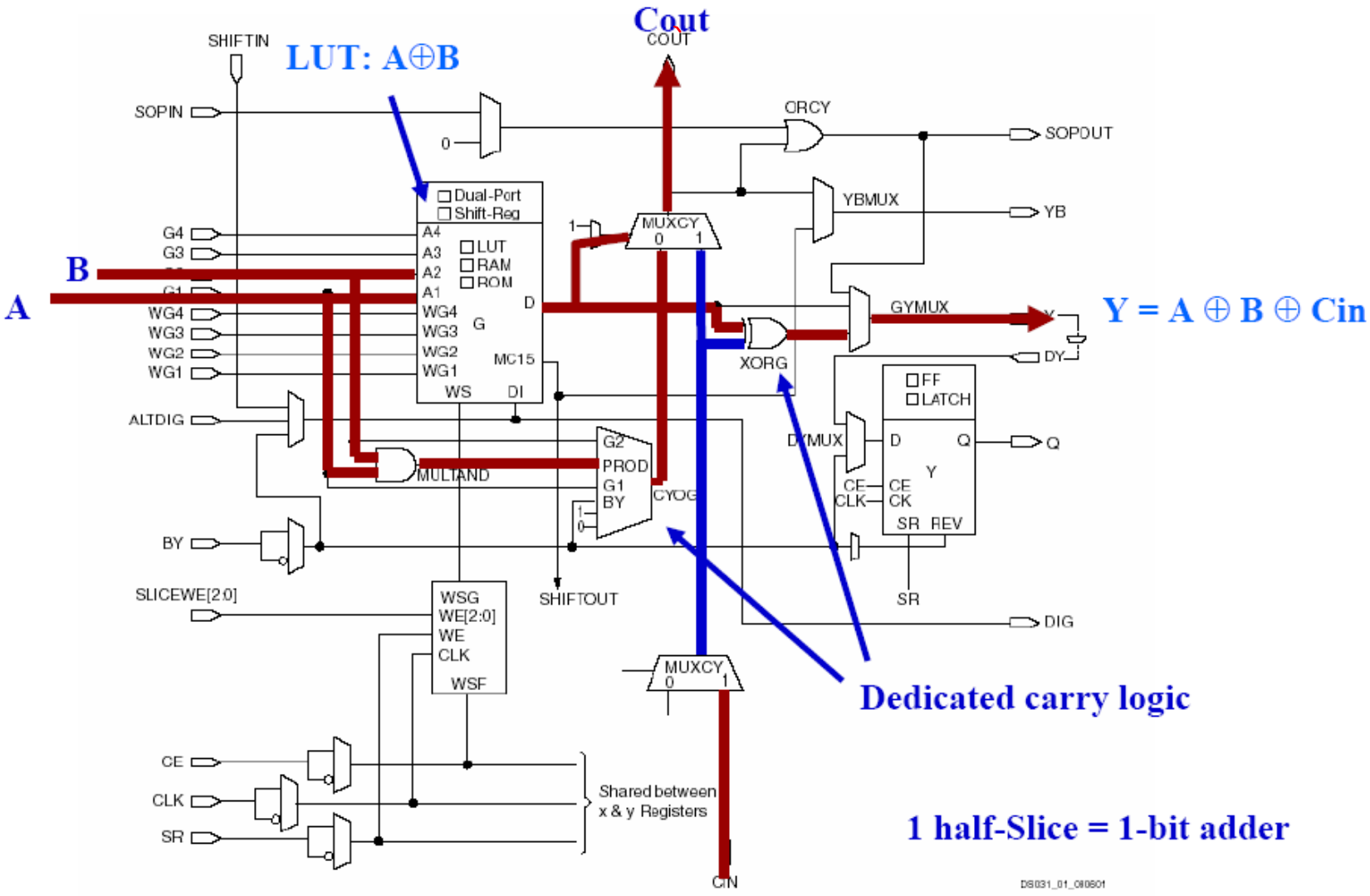
FPGA Performance Issues

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

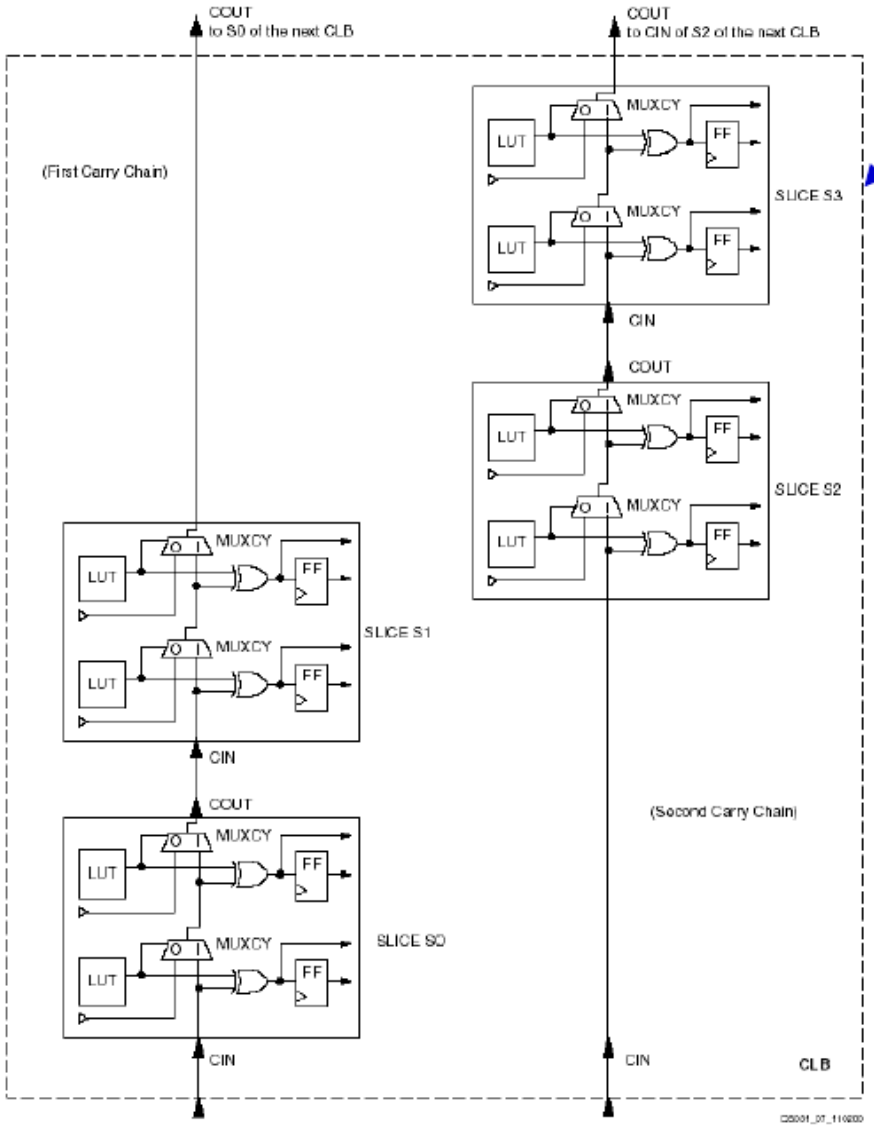


Source: xilinx.com

Adder Implementation

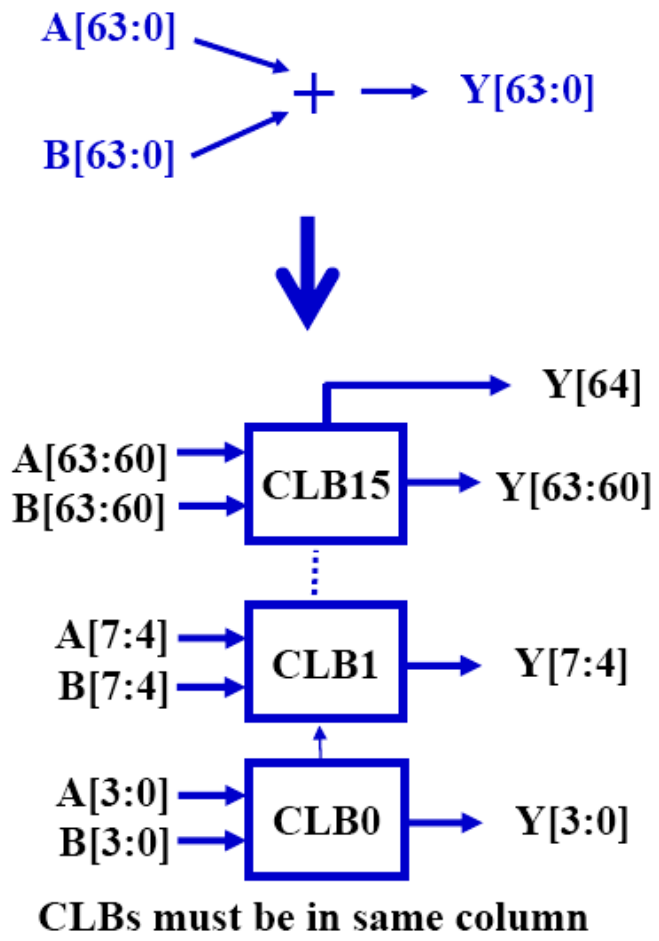


Carry Chain Implementation



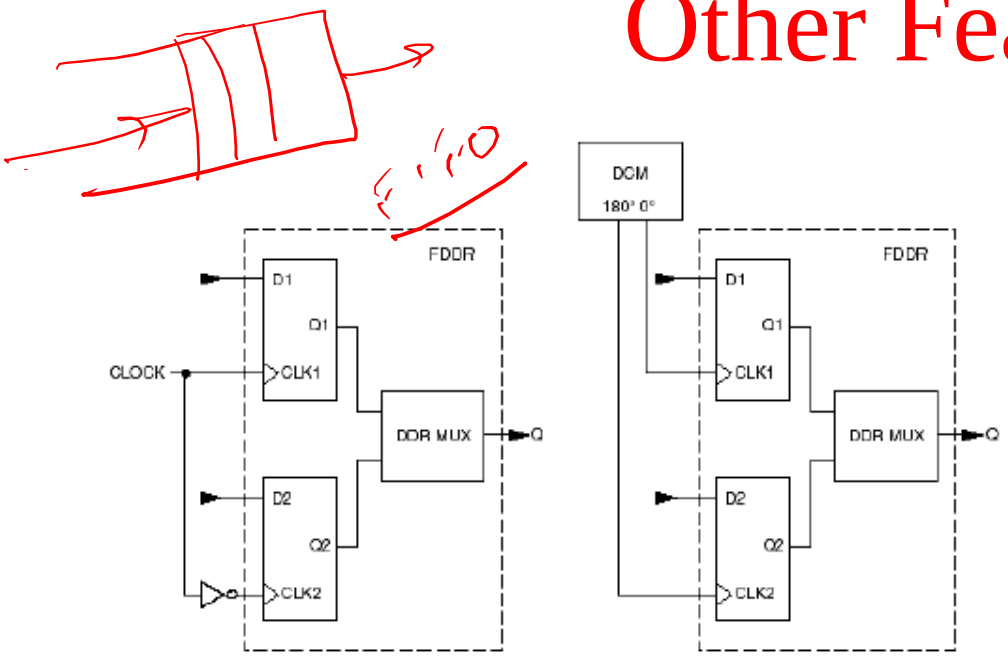
1 CLB = 4 Slices = 2, 4-bit adders

64-bit Adder: 16 CLBs

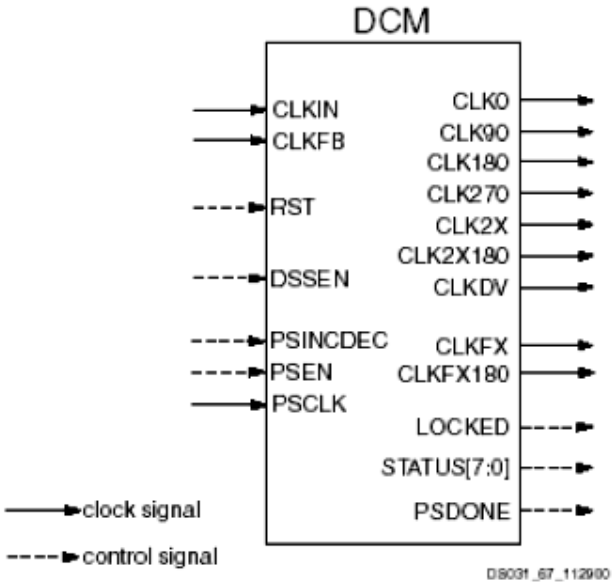


CLBs must be in same column

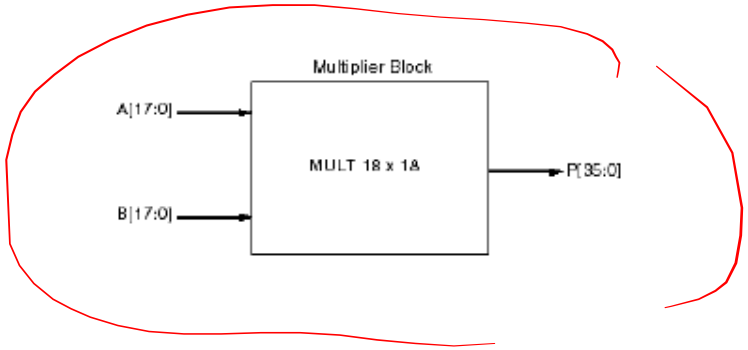
Other Features



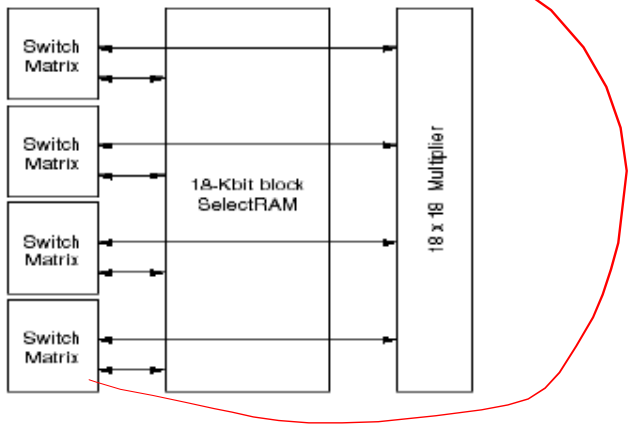
Double Data Rate registers



Digital Clock Manager



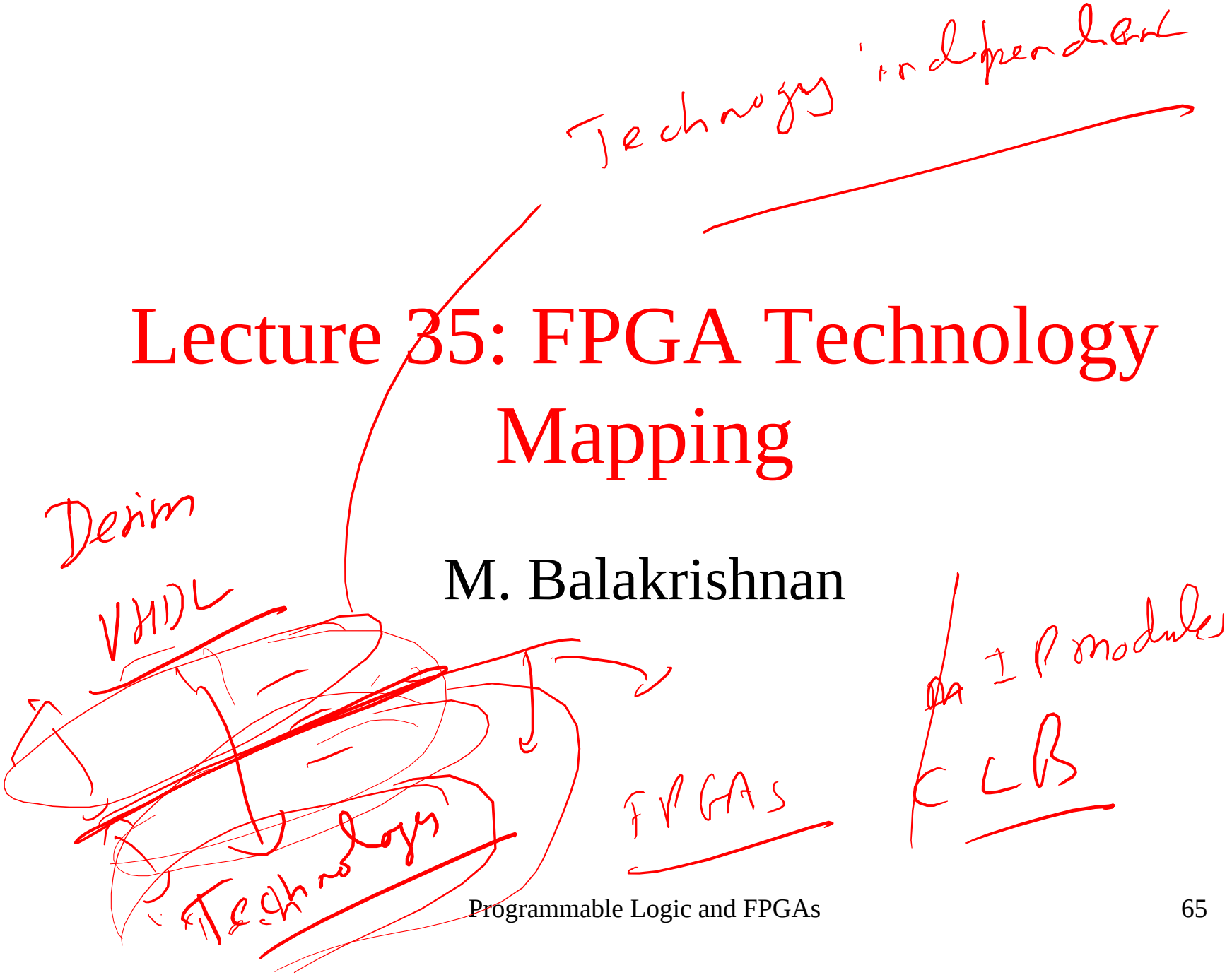
Embedded Multiplier



Block SelectRAM

Lecture 35: FPGA Technology Mapping

M. Balakrishnan



Definition

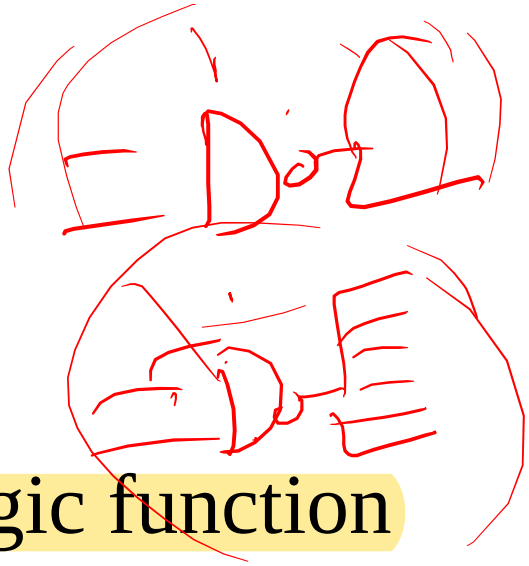
Technology mapping is also referred to as *library binding*.

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.

Input & Library

- 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

Typical Library



A typical simple library cell :-

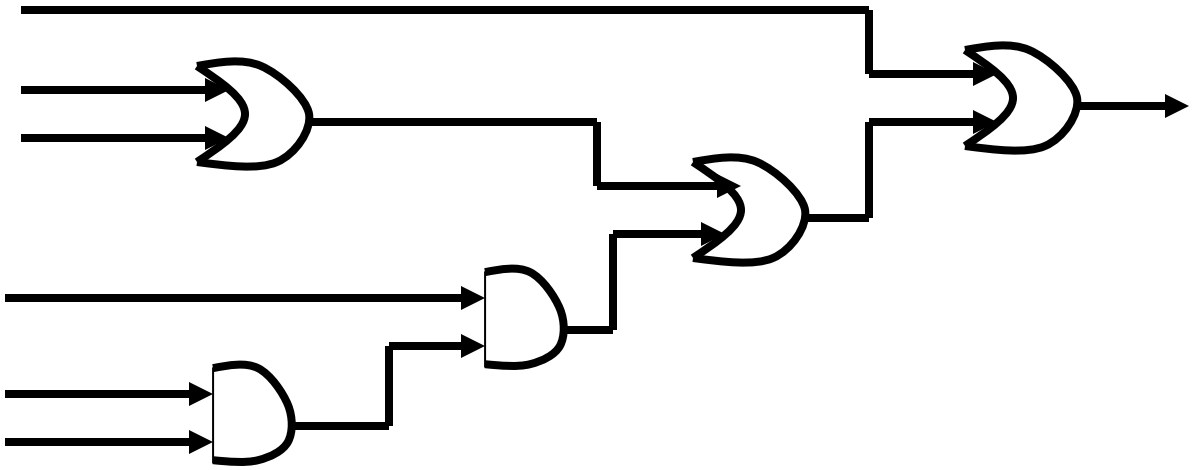
- a single output combinational logic function
- cost in terms of area
- delay in terms of propagation delays for each input/output pair and as a function of load and/or fanout. Sometimes only the worst case values are stored.
- power in terms of average current

13/11 -- Continue here

Network Covering

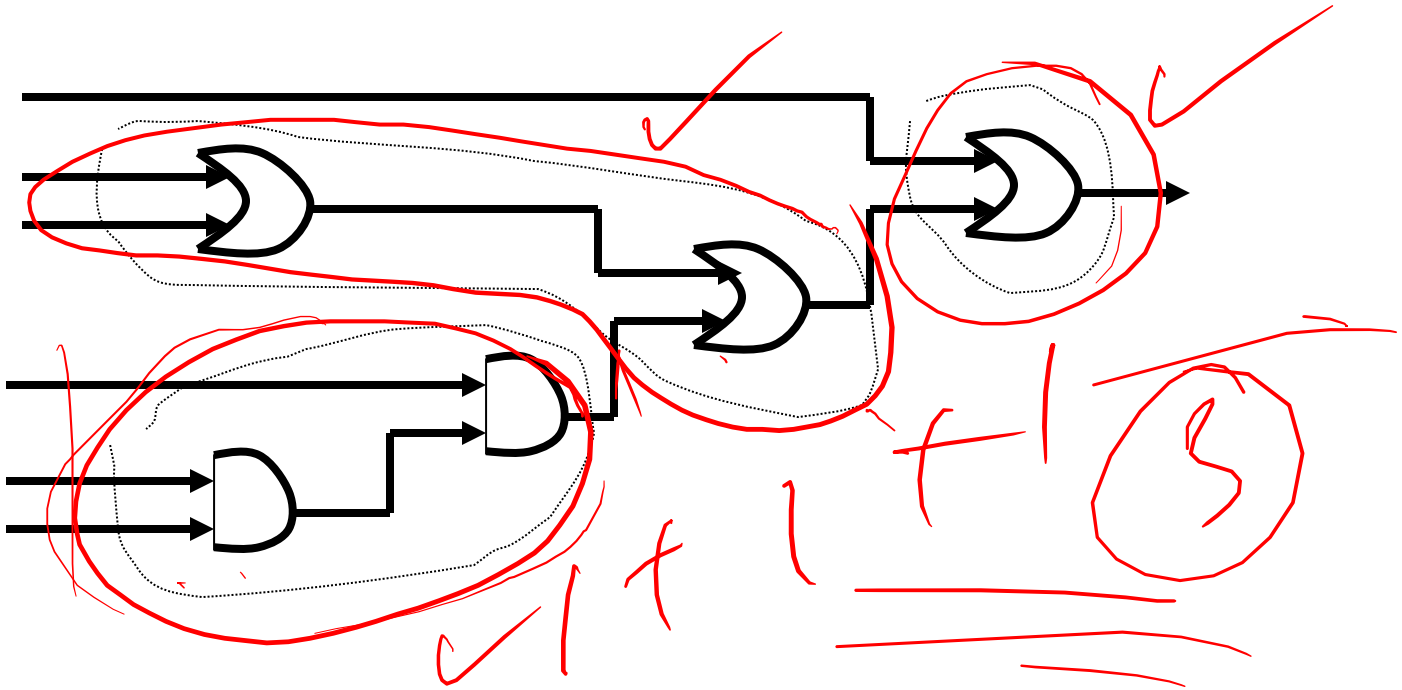
Network covering implies replacement of the sub-networks of the original network with cell library instances. Covering entails recognizing the equivalence of library cell to the identified sub-network and selecting adequate number of them to cover the whole network.

Example 1



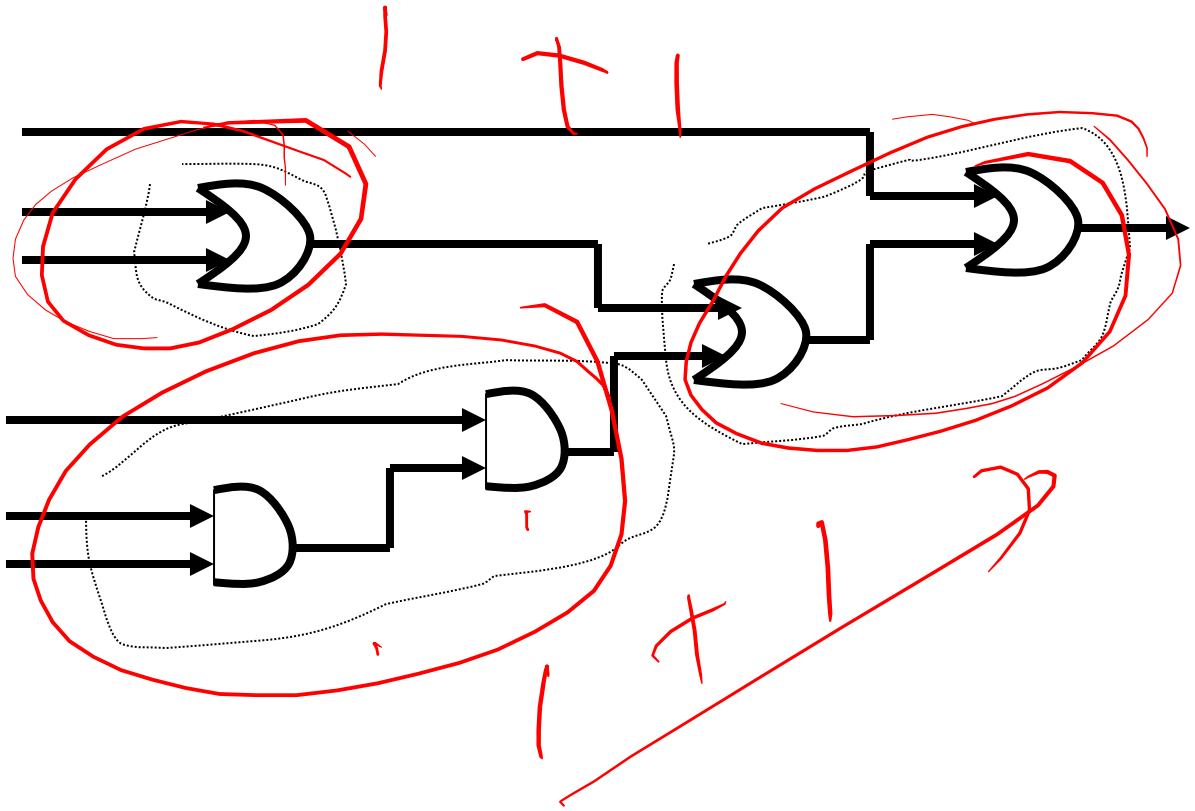
Cell library consists of two and three input gates

Example: First Mapping



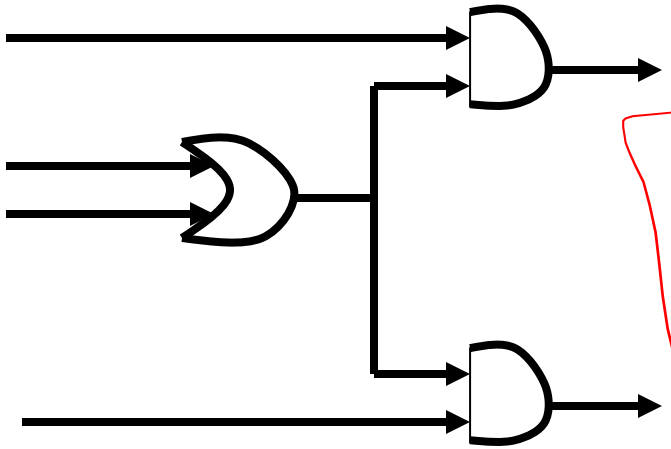
Cell library consists of two and three input gates

Example: Second Mapping



Cell library consists of two and three input gates

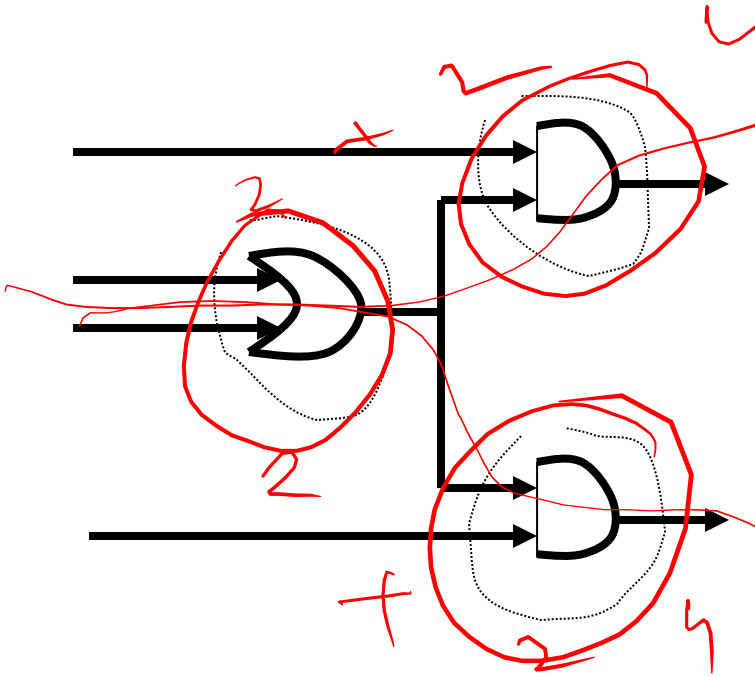
Example 2



Cell library consists of

Component	Area	Delay
AND2	3	2
OR2	3	2
OA21	5	3

Example: First Mapping

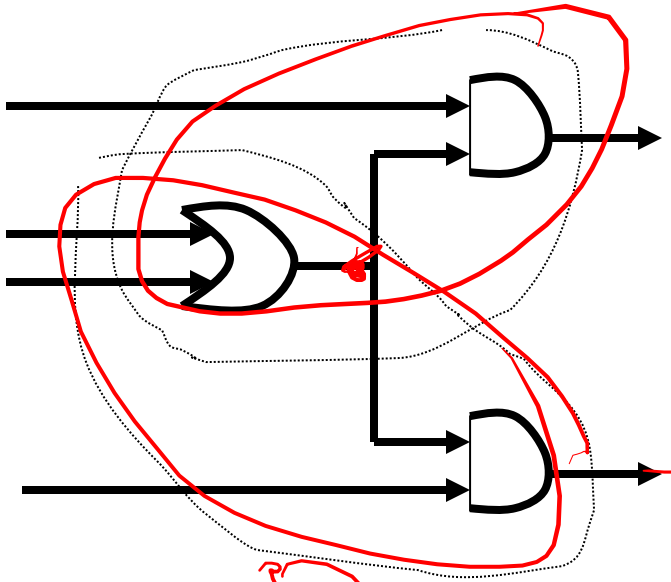


Cell library consists of

Component	Area	Delay
AND2	3	2
OR2	3	2
OA21	5	3

Area = 9, Delay = 4

Example: Second Mapping

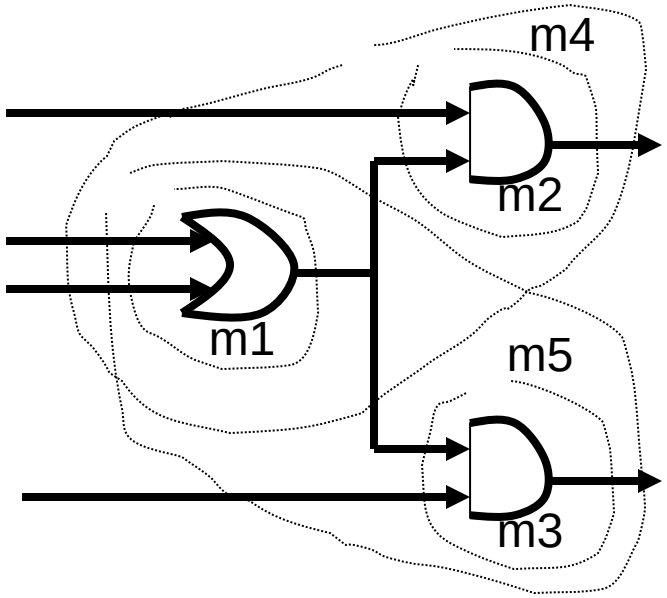


Cell library consists of

Component	Area	Delay
AND2	3	2
OR2	3	2
~~OA21~~	5	3

Area = 10, Delay = 3

Example



Cell library consists of

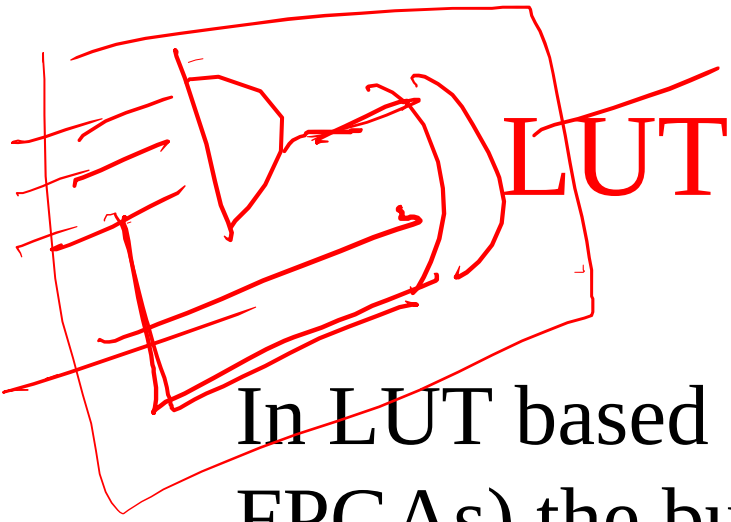
Component	Area	Delay
AND2	3	2
OR2	3	2
OA21	5	3

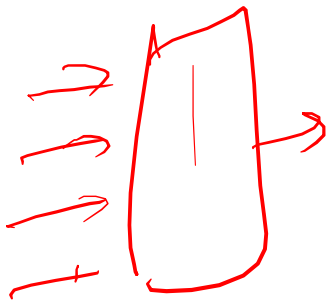
(m1 + m4 + m5)(m2 + m4)(m3 + m5)(m2’ + m1)(m3’ + m1) = 1

??

FPGA Structures & Mapping

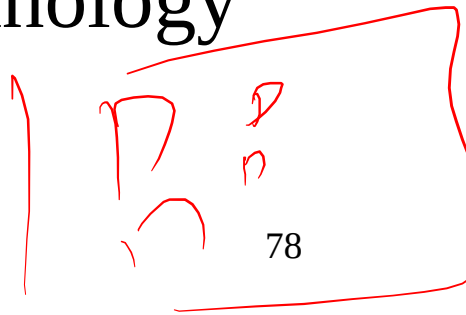
LUT Based FPGAs



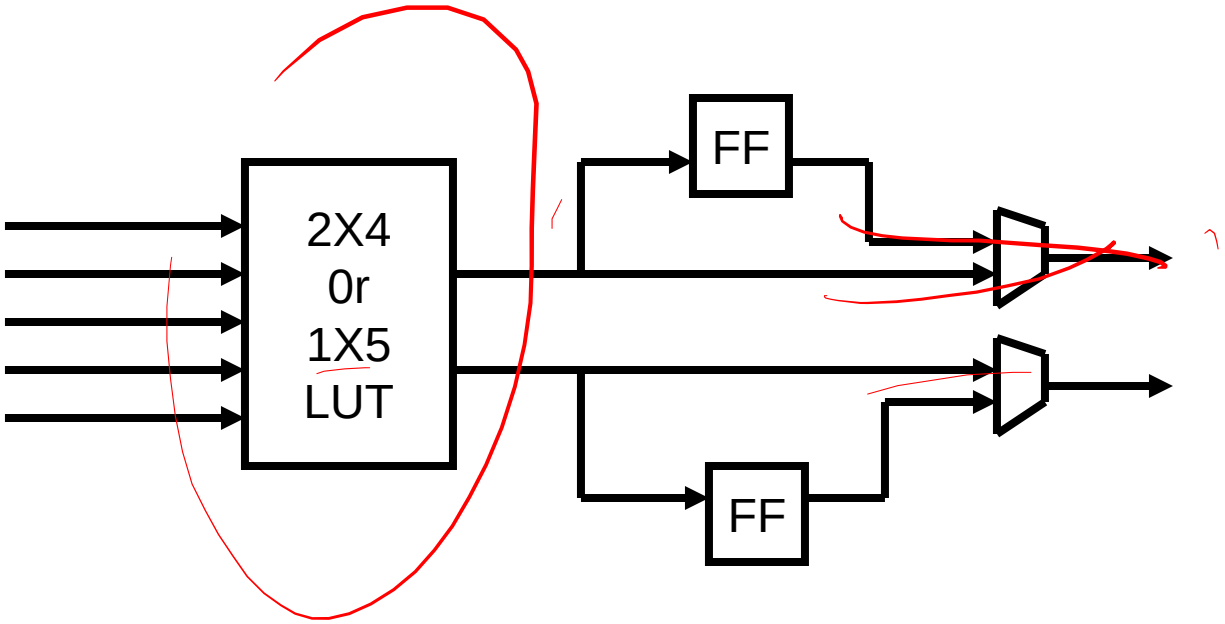


In LUT based FPGAs (example XILINX FPGAs) the building blocks are LUTs and Flip-Flops. A n-input LUT can implement all functions of n-variables.

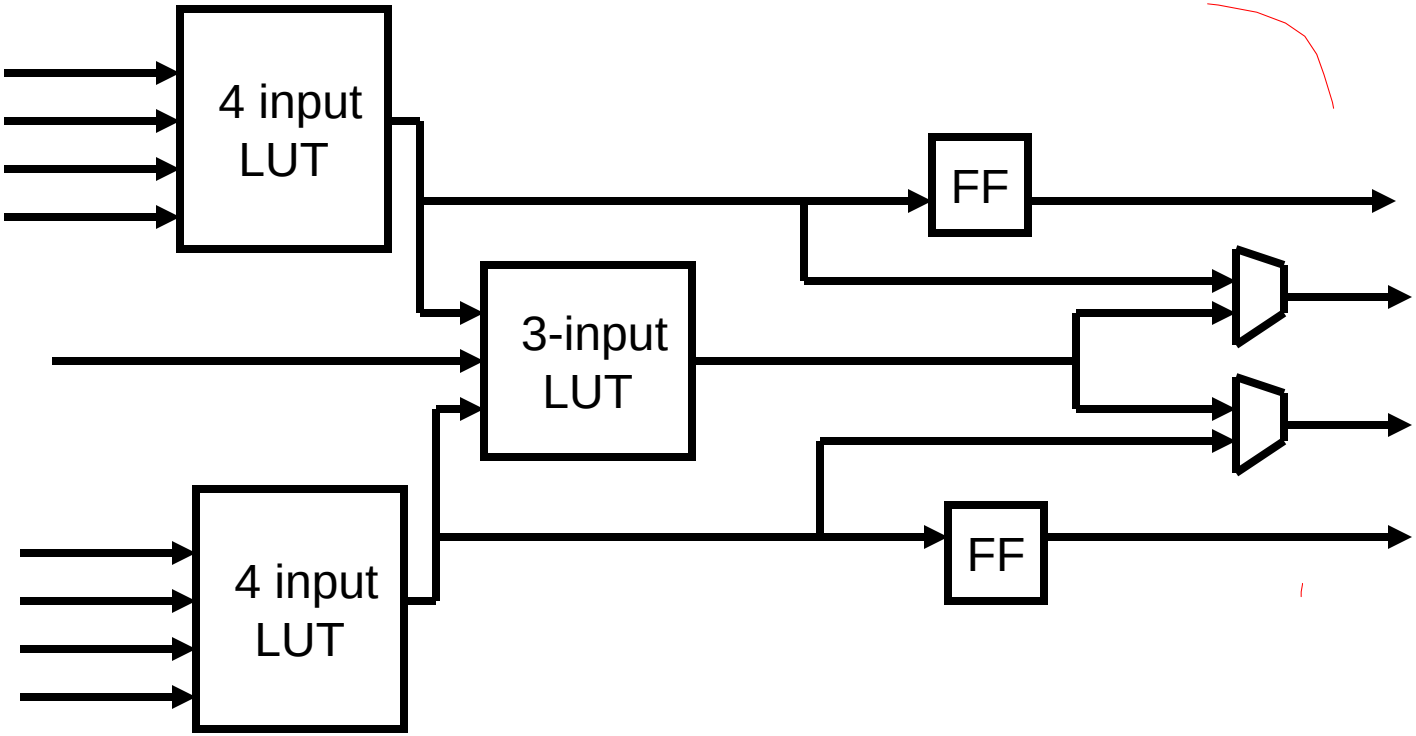
The FPGA itself is composed of CLB's with each CLB containing multiple LUT's and flip-flops which makes the technology mapping problem more complex.



XC3000 CLB

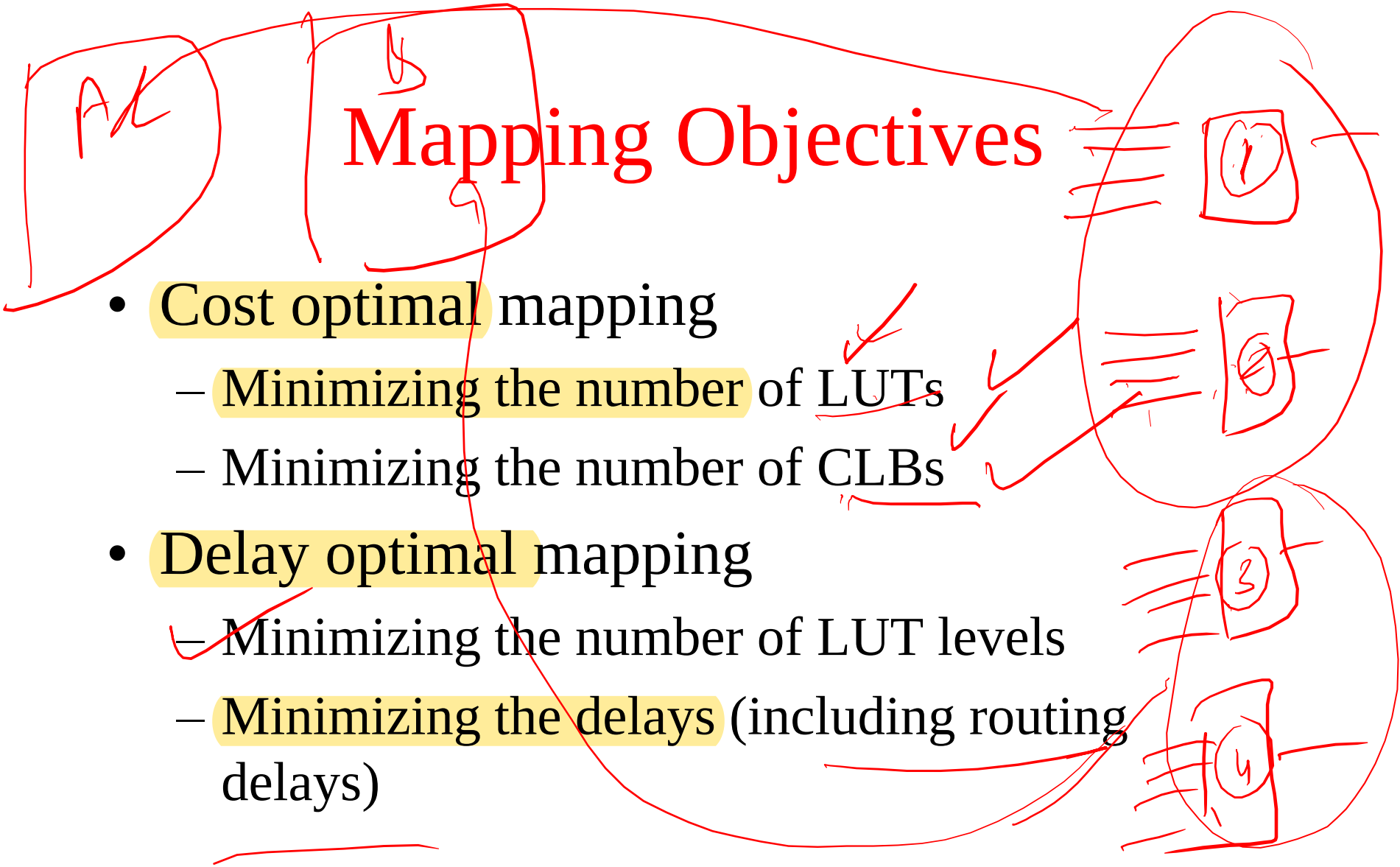


XC4000 CLB

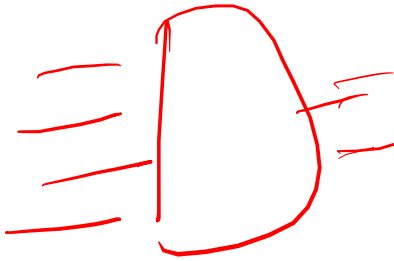


Mapping Objectives

- 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

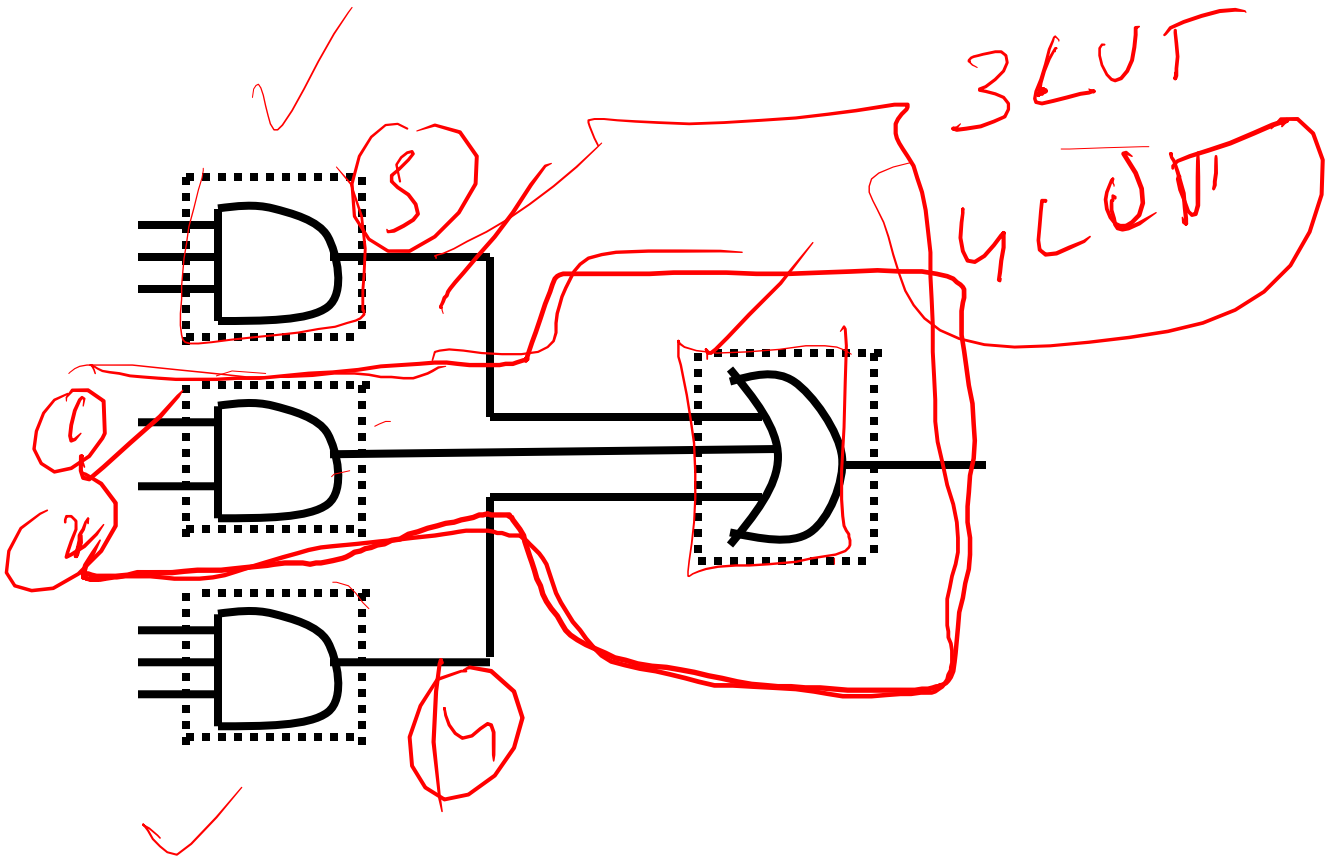


The problem of k-input LUT maps can be mapped to the problem of bin packing. We have to minimize the number of bins each with a capacity of k.

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.

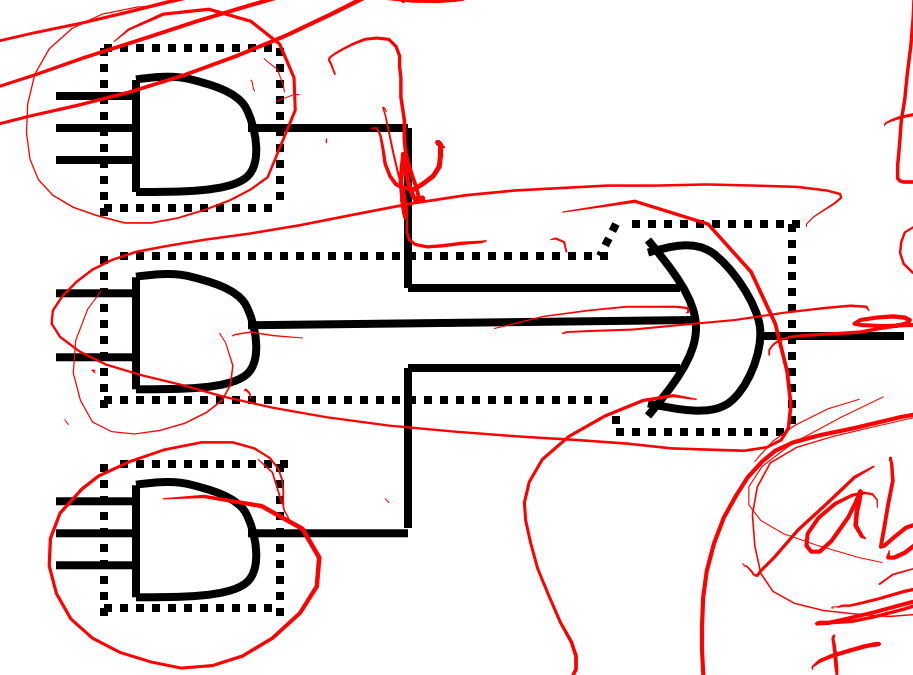
Example: Simple Mapping

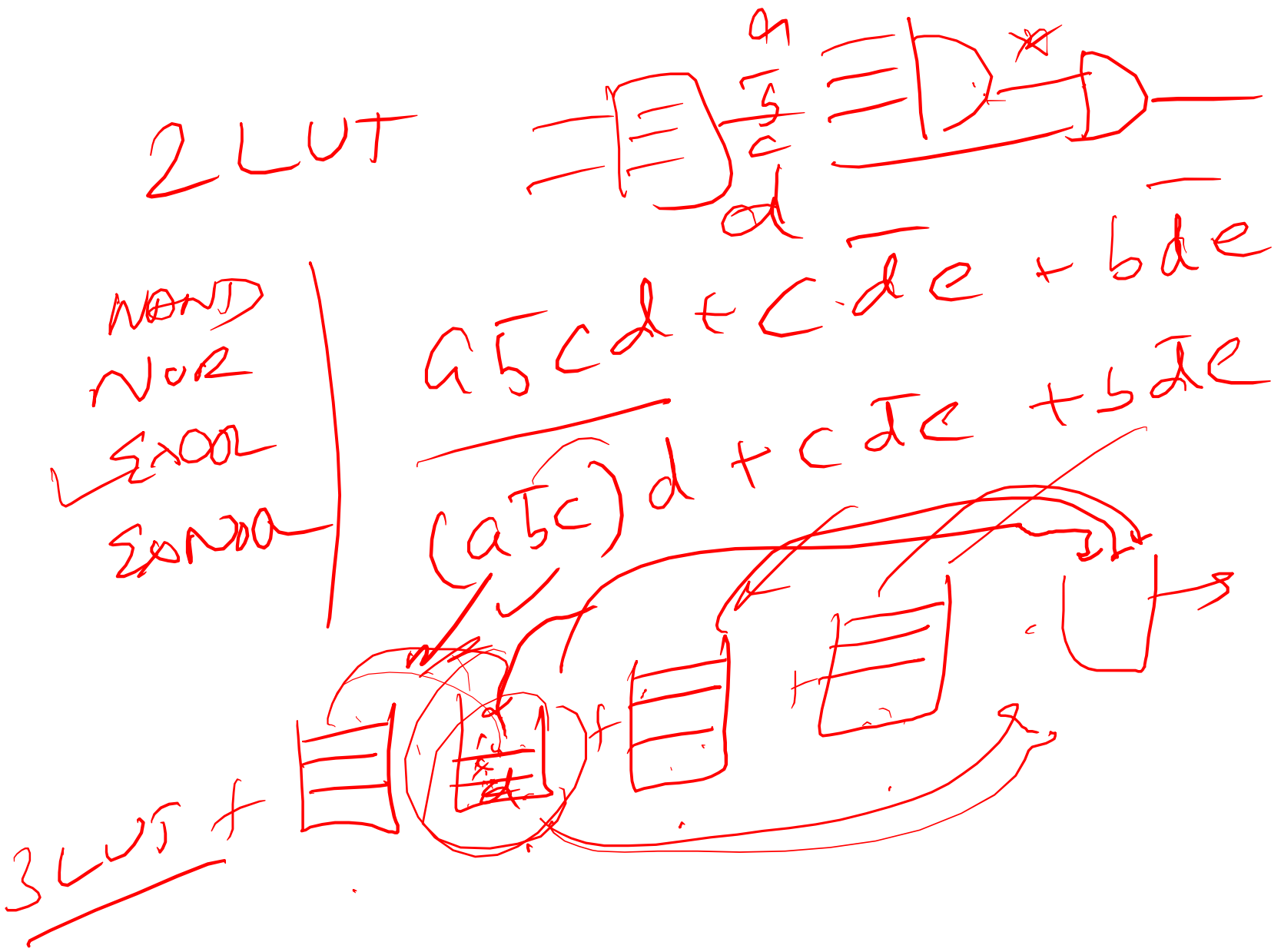


Sum of Products: Bin Packing

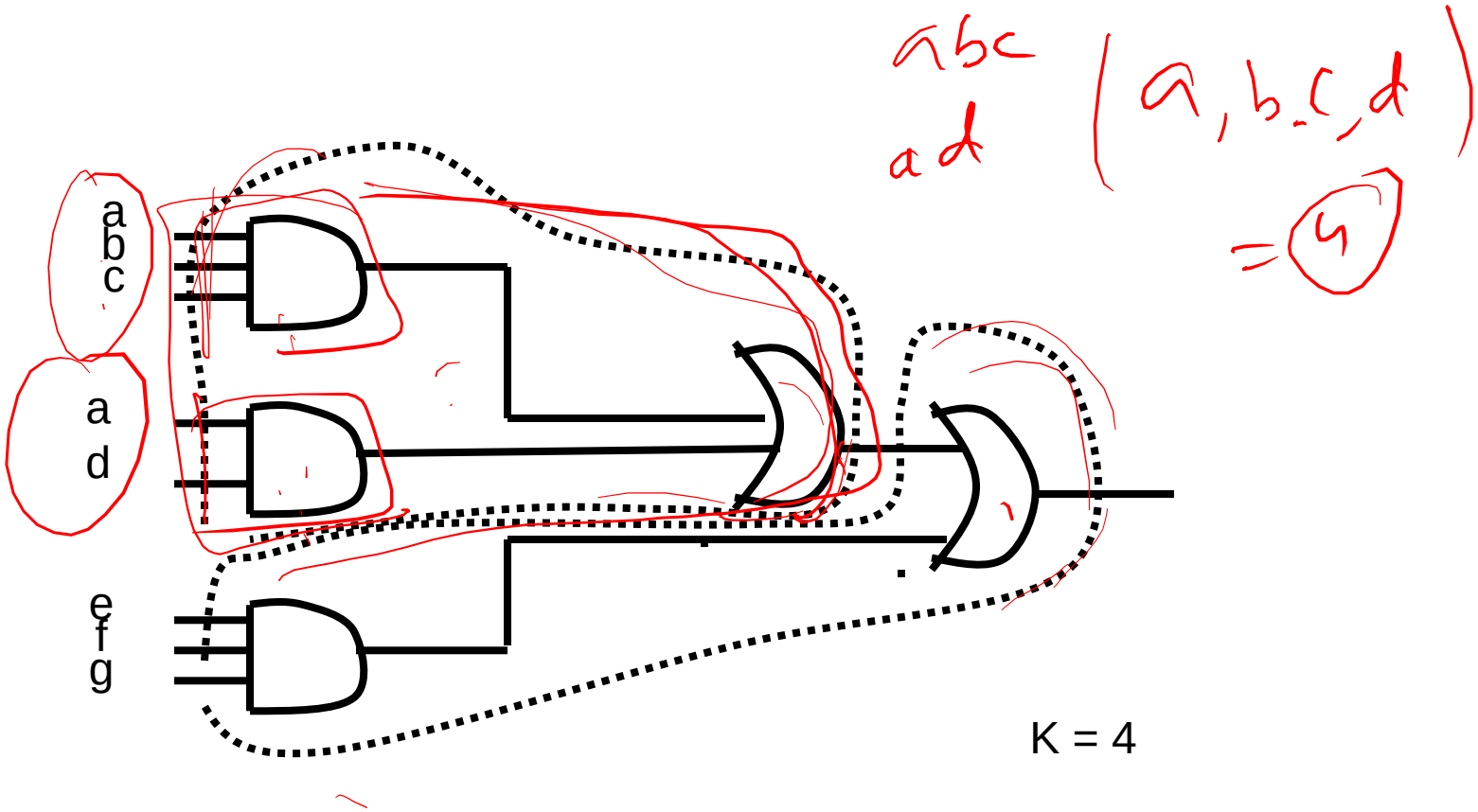
- Select the product term with the most number of variables and fit it into any table where it fits and if it doesn't fit anywhere add a new table
- The table with the fewest number of unused inputs is declared as final
- Associate this output with the first table that can accept it

Example: 4-input LUT

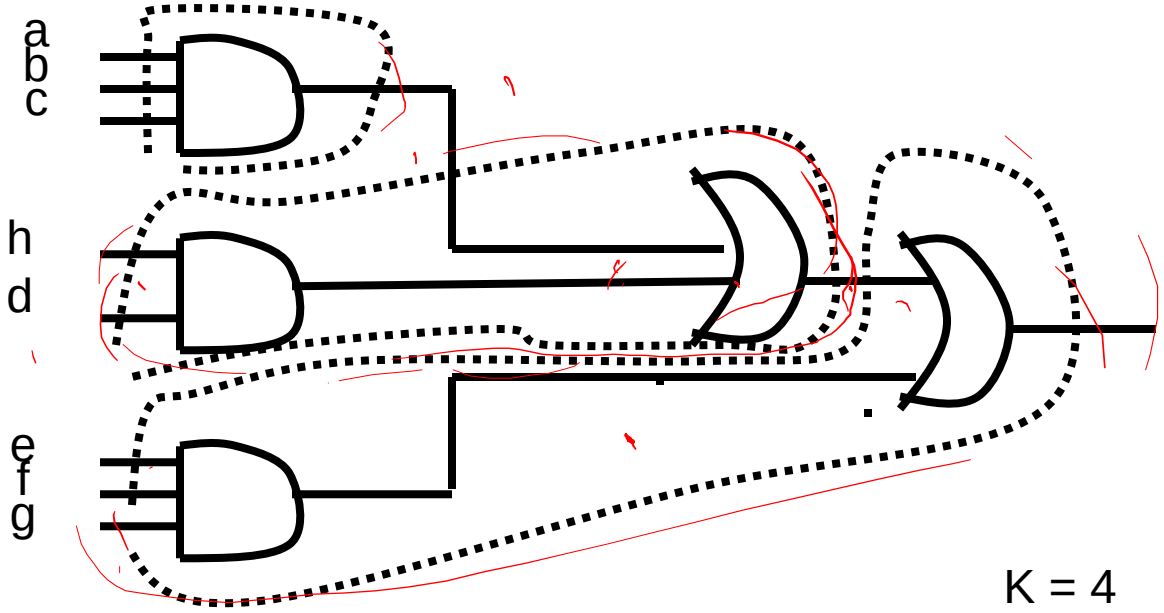



2 LUT
NAND
NOR
EXOR
EXNOR
3 LUT

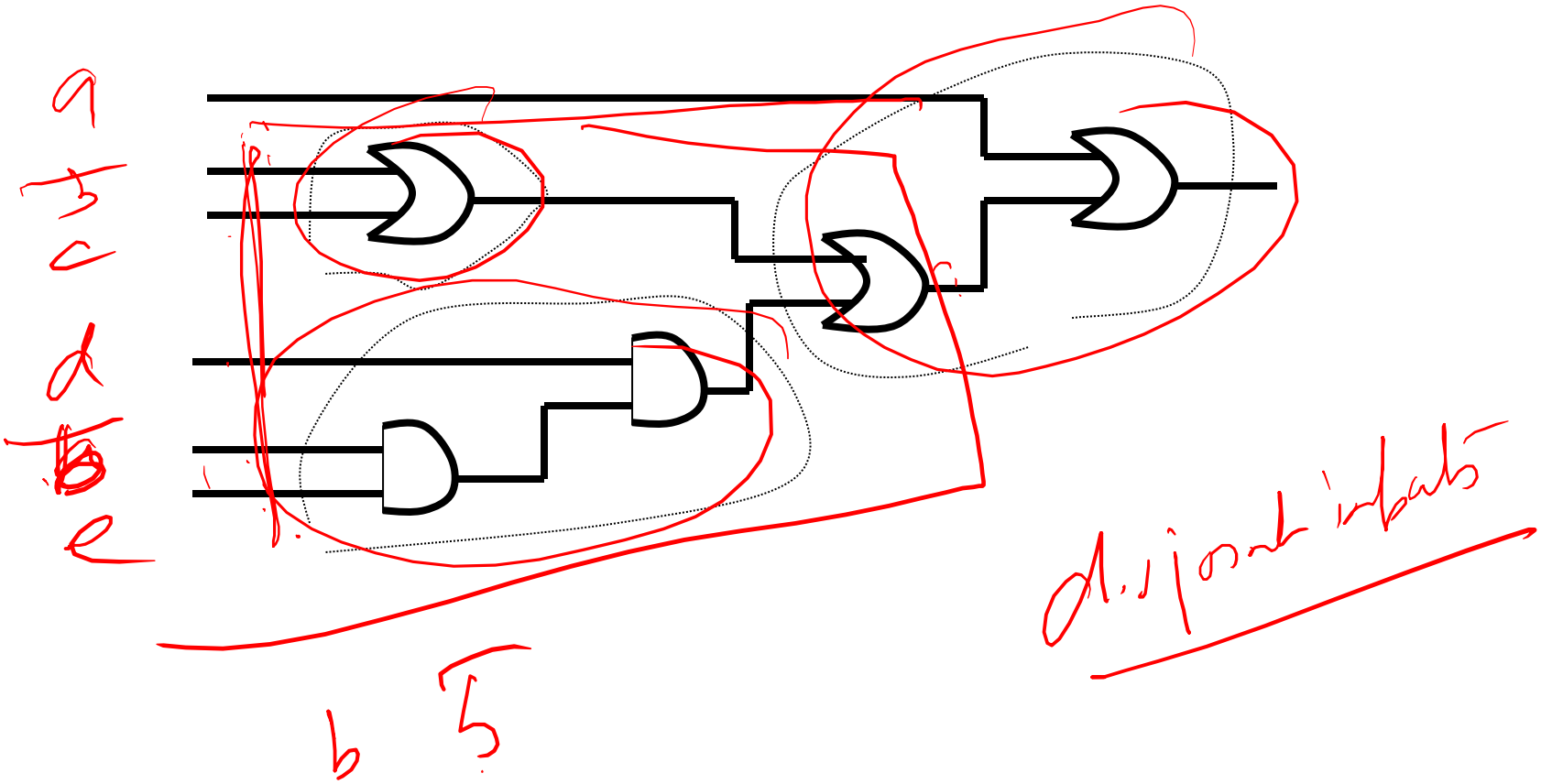
Example: Overlapping Inputs



Example: Decomposition



Example: 3 input LUT



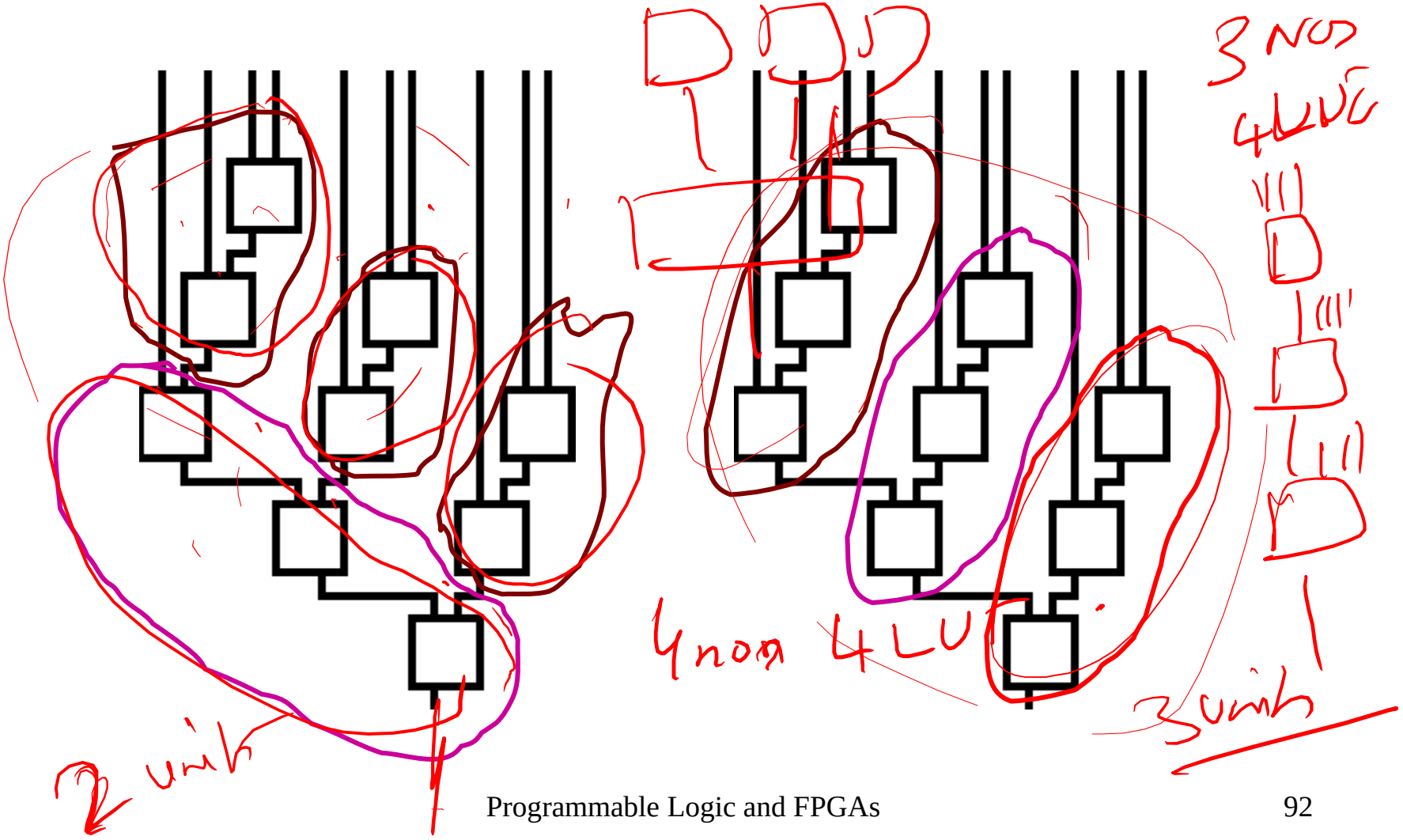
FPGA Technology Mapping: Issues

LUT Mapping

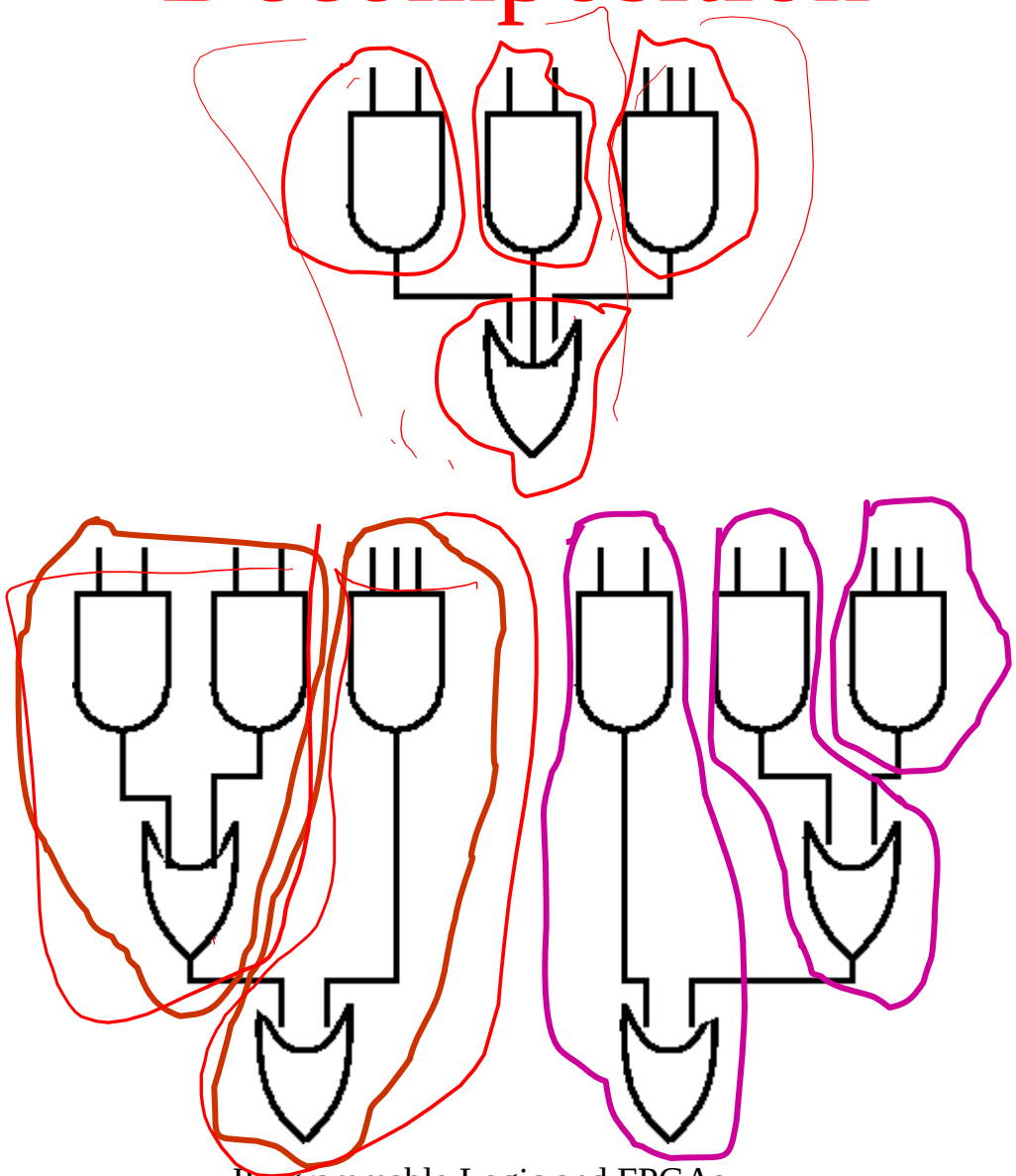
Starting from a technology independent optimized circuit, produce a minimal LUT cover for the circuit. The complexities are due to the following reasons.

- Fanout nodes
- Reconvergence
- Node decomposition and packing

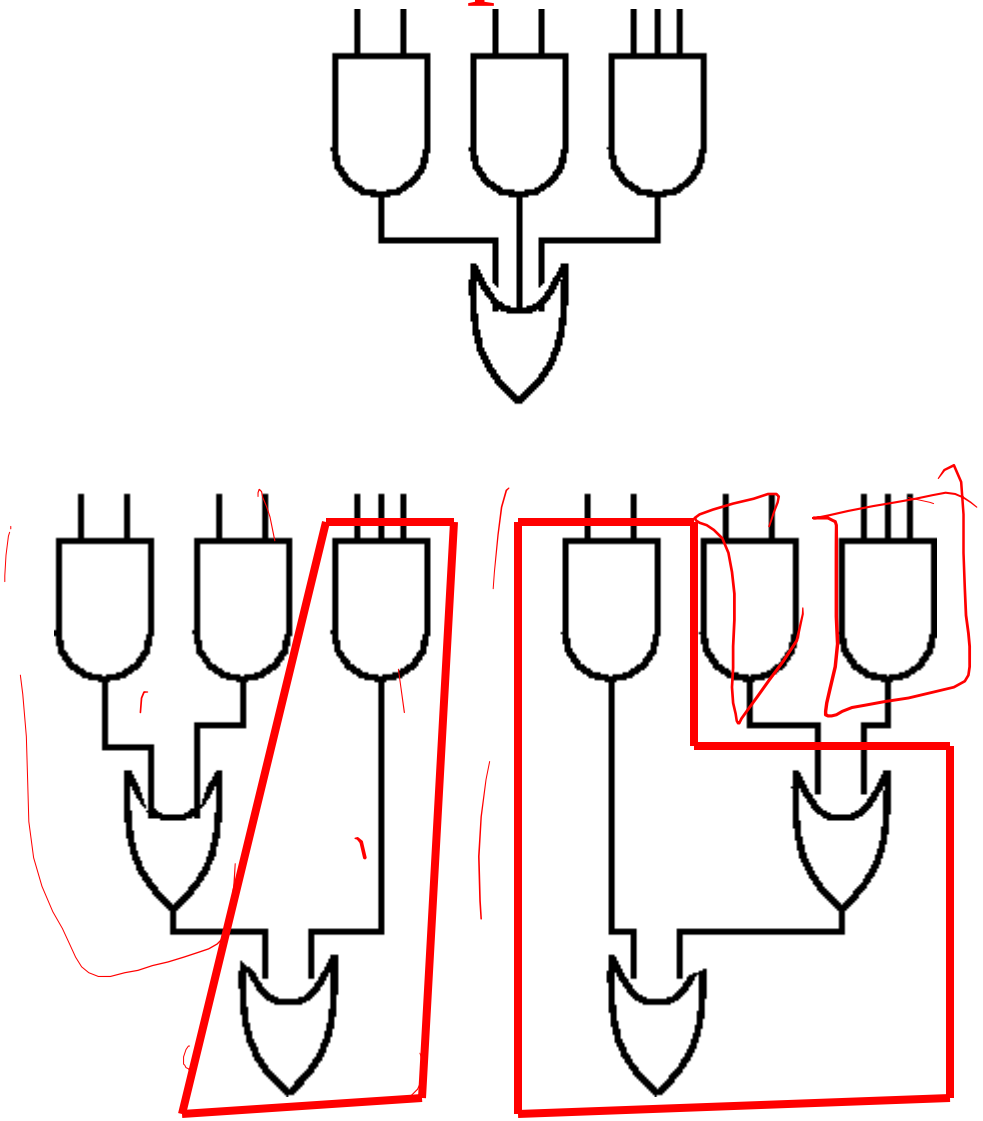
Area vs. Delay



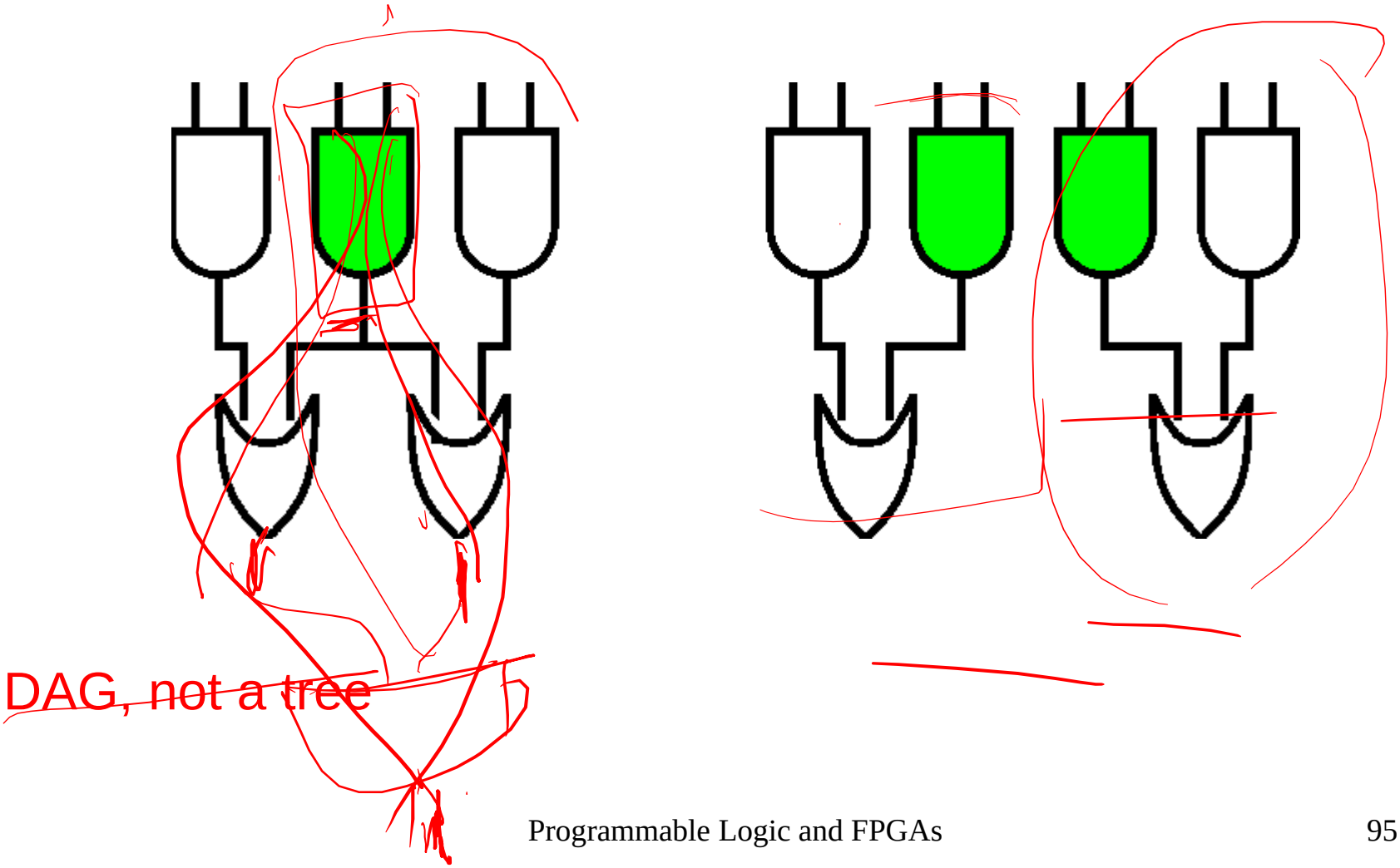
Decomposition



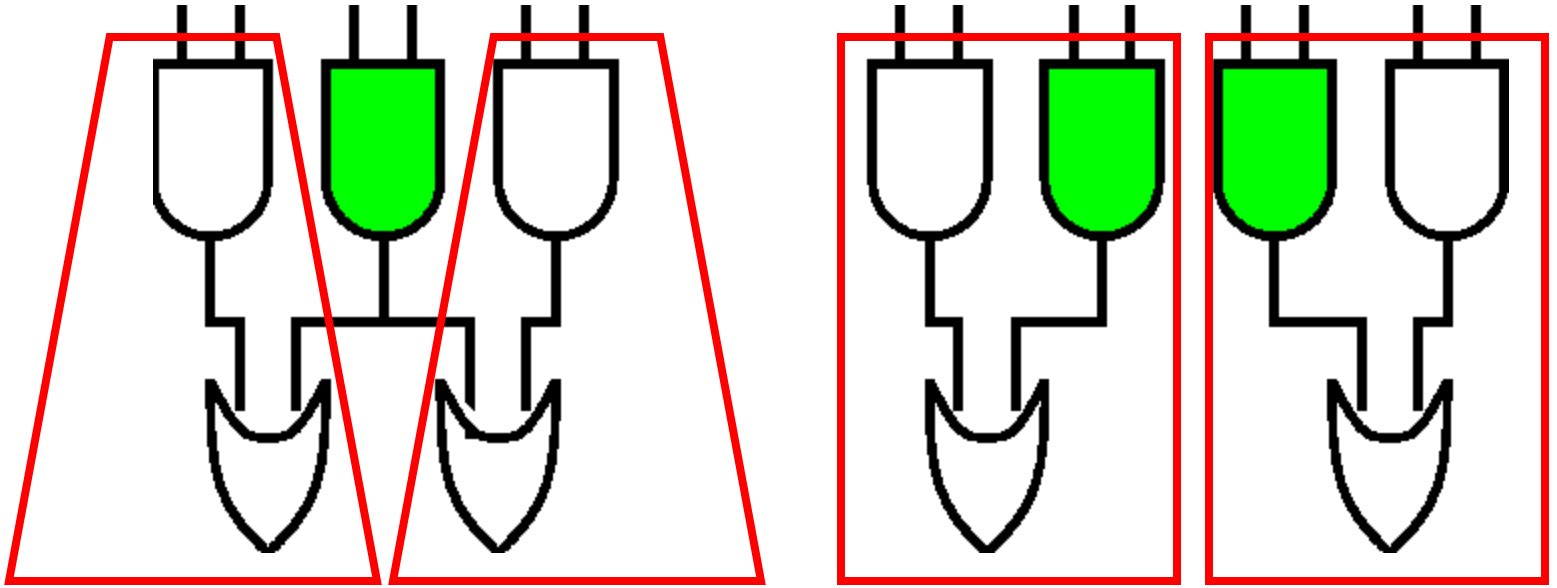
Decomposition



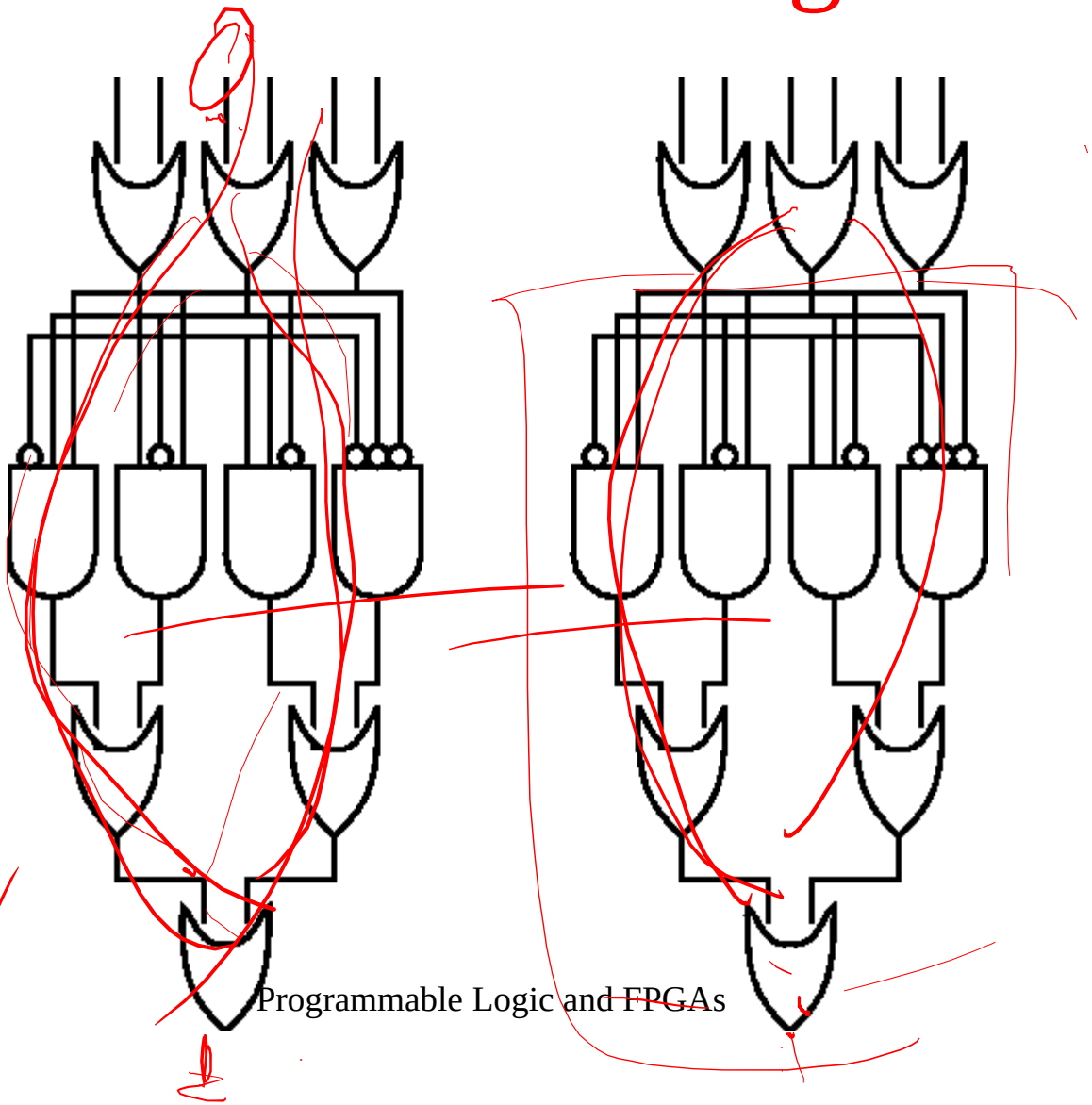
Fanout: Replication



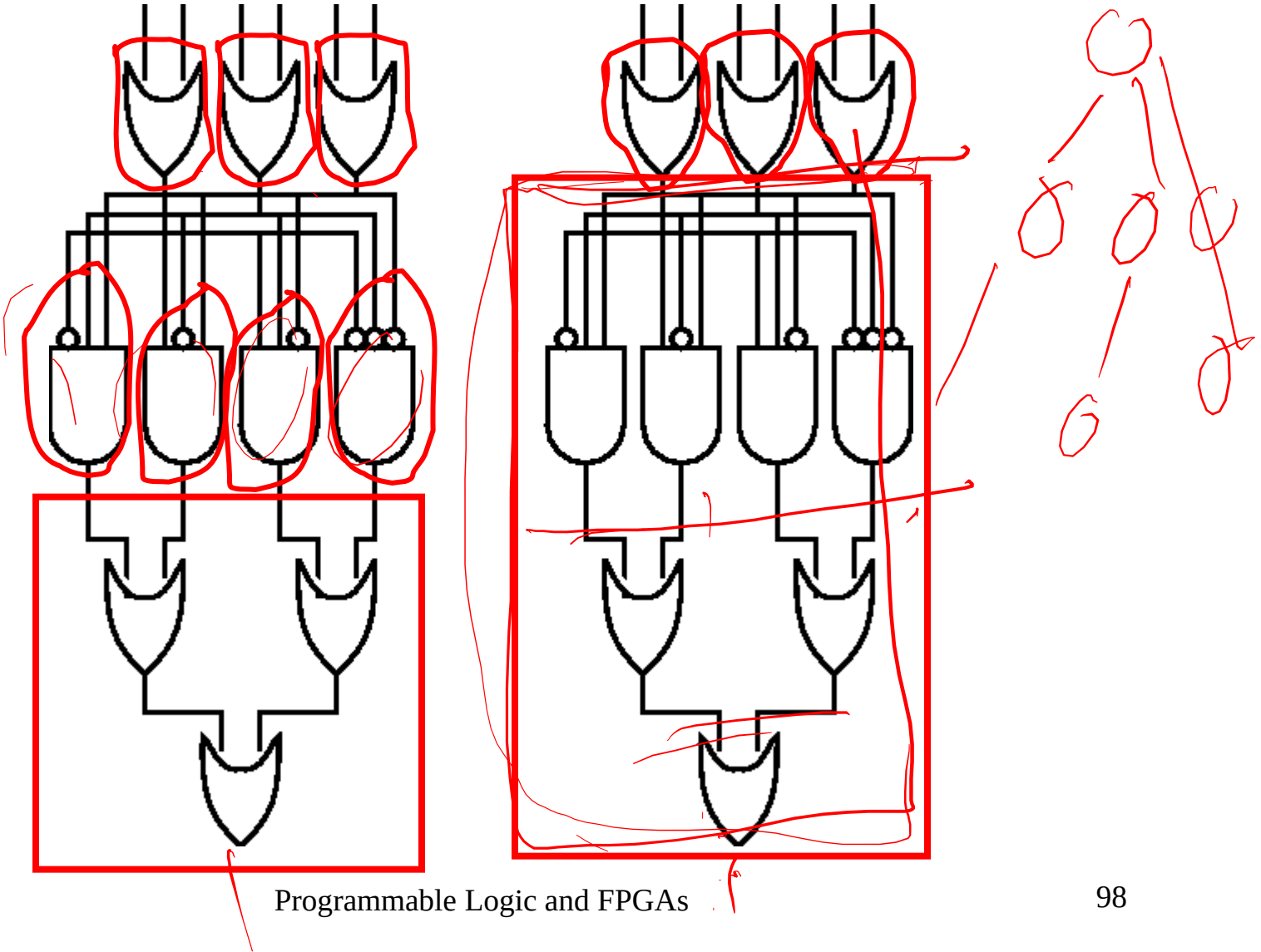
Fanout: Replication



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.

Thank You