Verilog HDL and Synthesis

Noor Mahammad Sk

Assistant Professor

Dept. of Computer Science & Engineering

Indian Institute of Information Technology, Design and Manufacturing

Chennai – 48

Outline

- Introduction to CAD Tools
- Verilog HDL Constructs
- Synthesizable Verilog HDL Constructs
- High Performance Circuit Design techniques
- How to write Test benches
- Conclusions
- References and Resources

Evolution of CAD Tools

- Digital circuit design evolved over last three decades
- SSI – Small Scale Integration (Tens of transistors)
- MSI – Medium Scale Integration (Hundreds of transistors)
- LSI – Large Scale Integration – (Thousands of Transistors) - demanded automation of design process – CAD started evolving.

Evolution of CAD Tools

- VLSI – Very Large Scale Integration – Tens of Thousands of Transistors – CAD Tools are inevitable
- VLSI chip design forced
  - Automation of process
  - Automation of Simulation based verification - replacing breadboard techniques – HDL development
  - Modular and Hierarchical techniques of design – a natural object orientation approach

CAD Terminologies

- HDL – Hardware Description Language
  - Describing a circuit to the computer
  - A programming language by all means
  - Concurrency constructs to simulate circuit behavior
  - Verilog and VHDL
  - Simulation for verification and Synthesis
  - Synthesizable constructs - RTL

CAD Terminologies

- RTL – Register Transfer Level
  - Specifying how the data flows between registers and how the design processes data
  - Registers store intermediate results
  - Logic between any two registers in a data flow determines the speed of the circuit
- Synthesis – Converting RTL to a set of gates and wires connecting them – Ambit of Cadence, Design Compiler of Synopsys, *Precision* of Mentor, Blast Fusion from Magma are some of the commercially available synthesis tools.

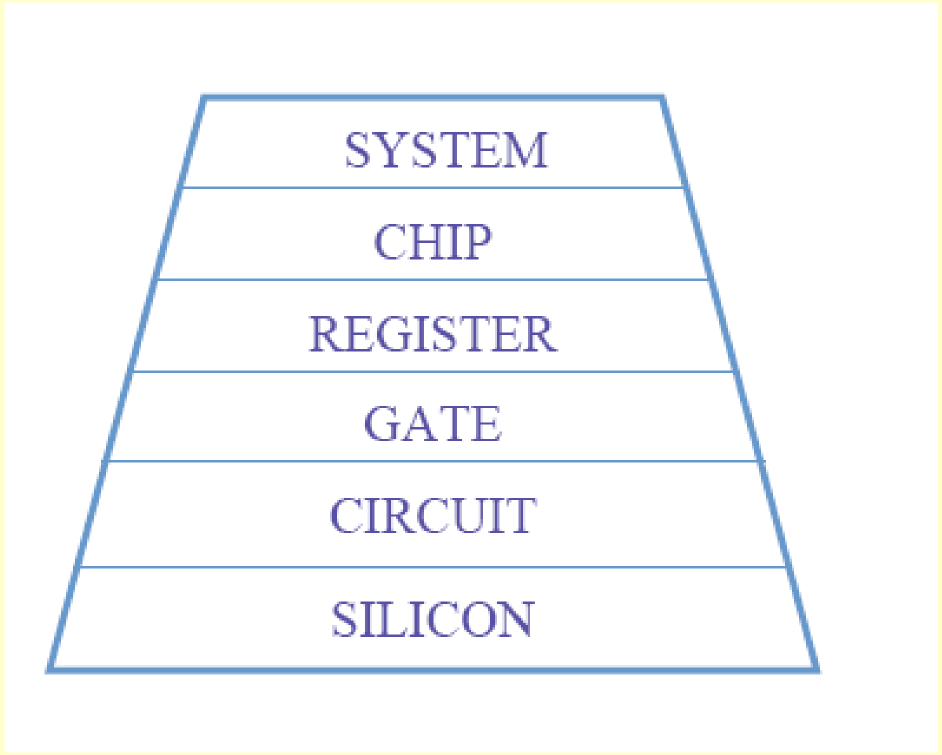
Design Flow

- The process of converting an “idea”to a “chip”is called the VLSI Design Process.
- VLSI Design Process involves a sequence of steps –Flow.
- Tools that enable the design process are called CAD (Computer Aided Design) tools for VLSI.

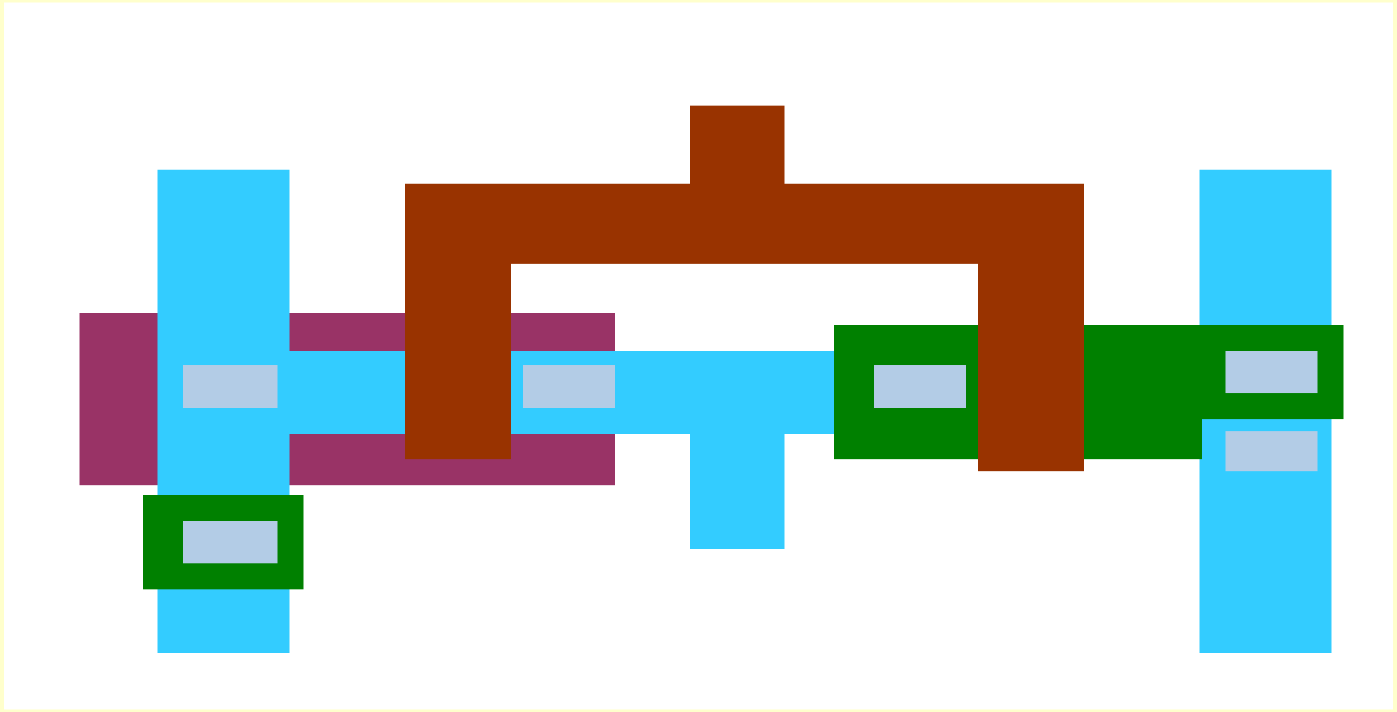
Abstraction Hierarchy

- Designers use different abstraction domains for VLSI design.
- Structural Domain
  - –Set of primitive components.
  - –Primitive components are interconnected to form larger components.
- Behavioral Domain
  - –Components are defined by their input/output response.
  - –The components can themselves be implemented in many ways.

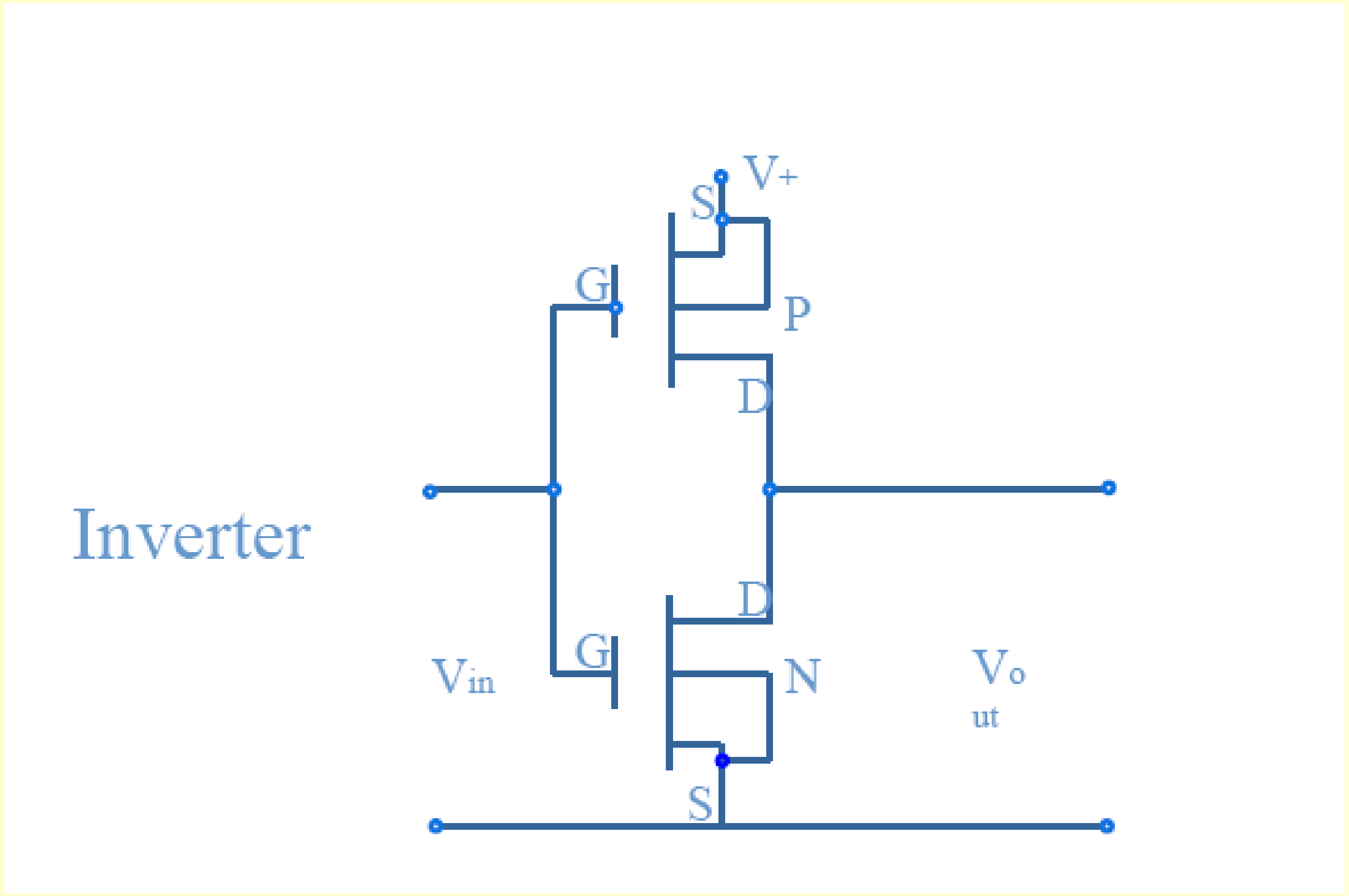
Abstraction Levels



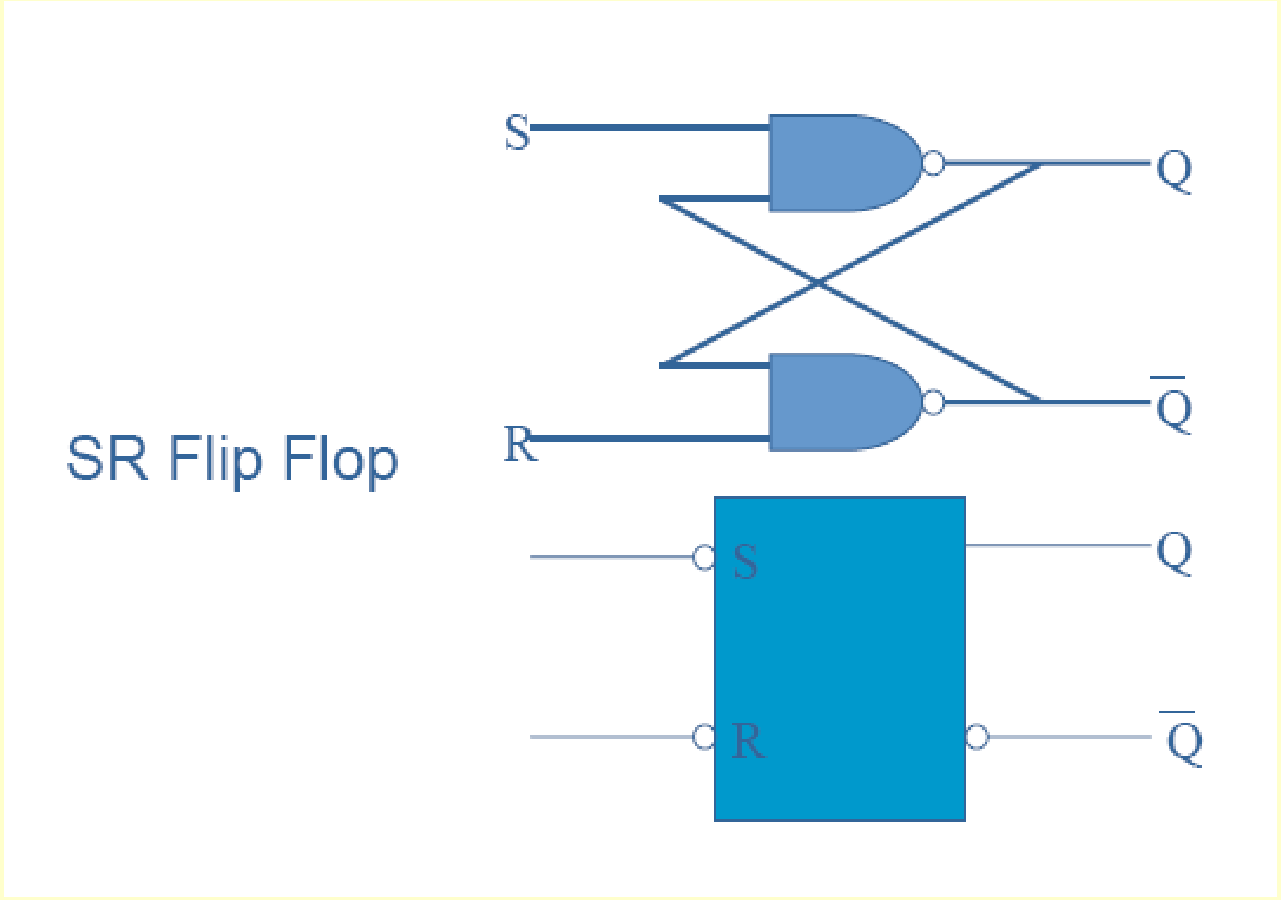
Silicon Level



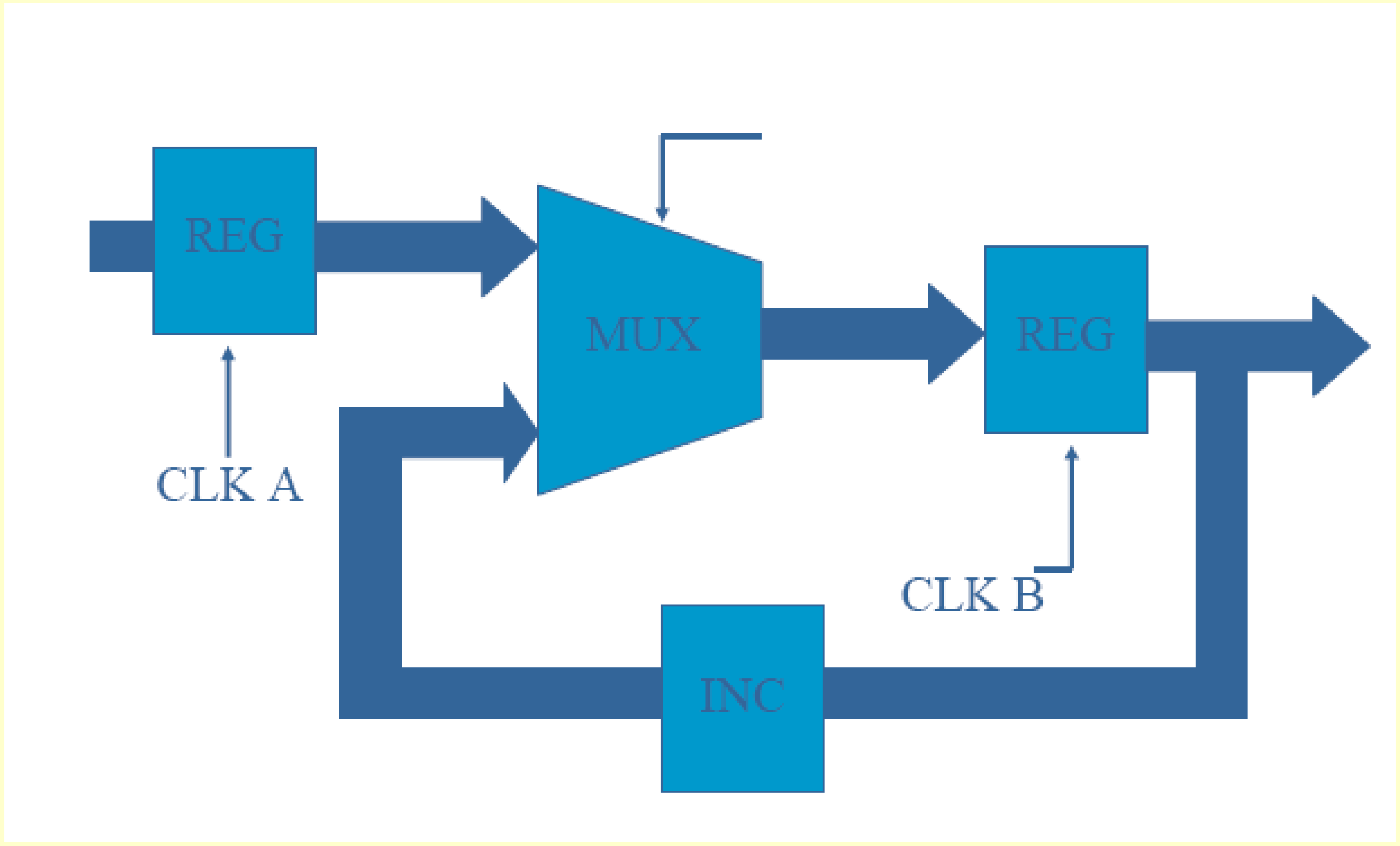
Circuit Level



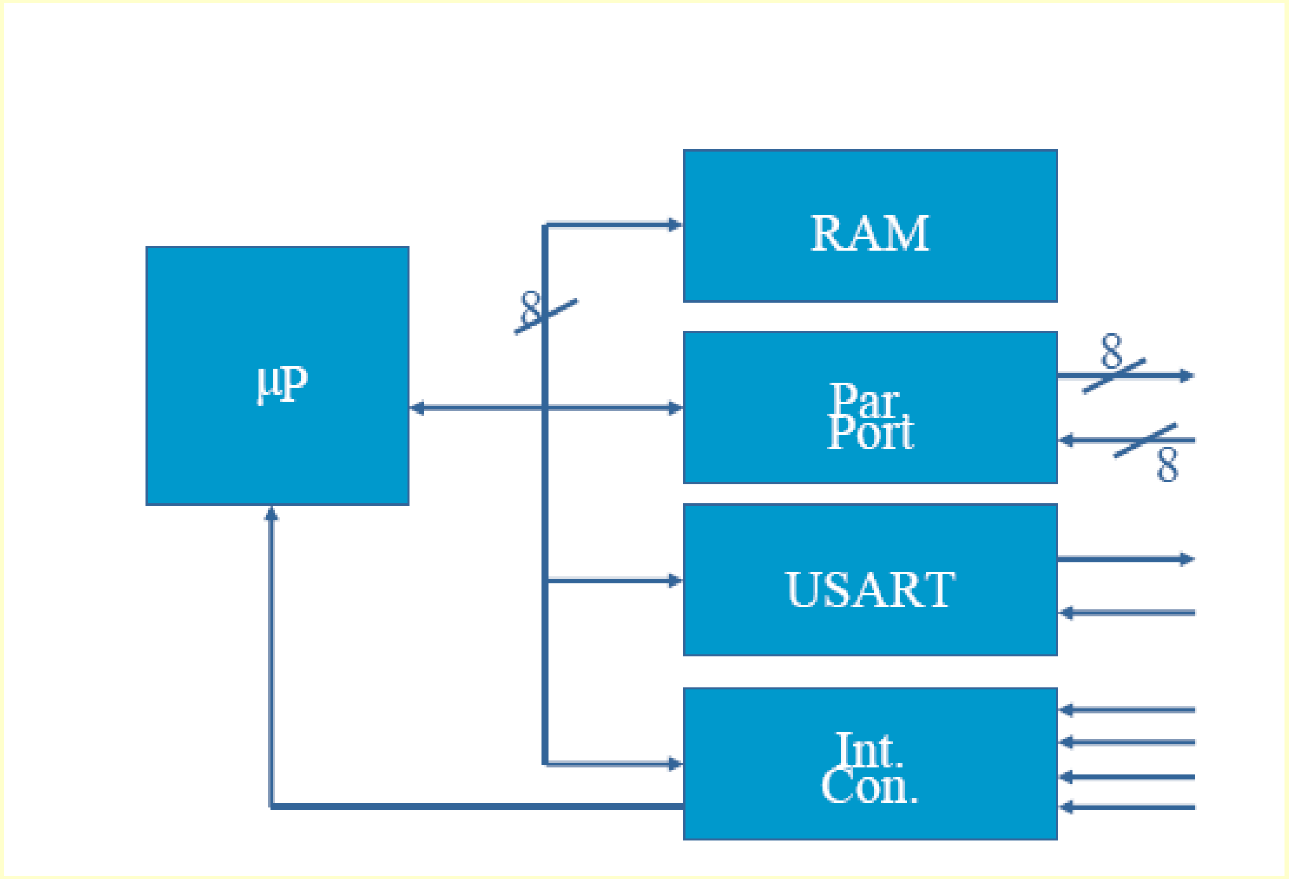
Gate Level



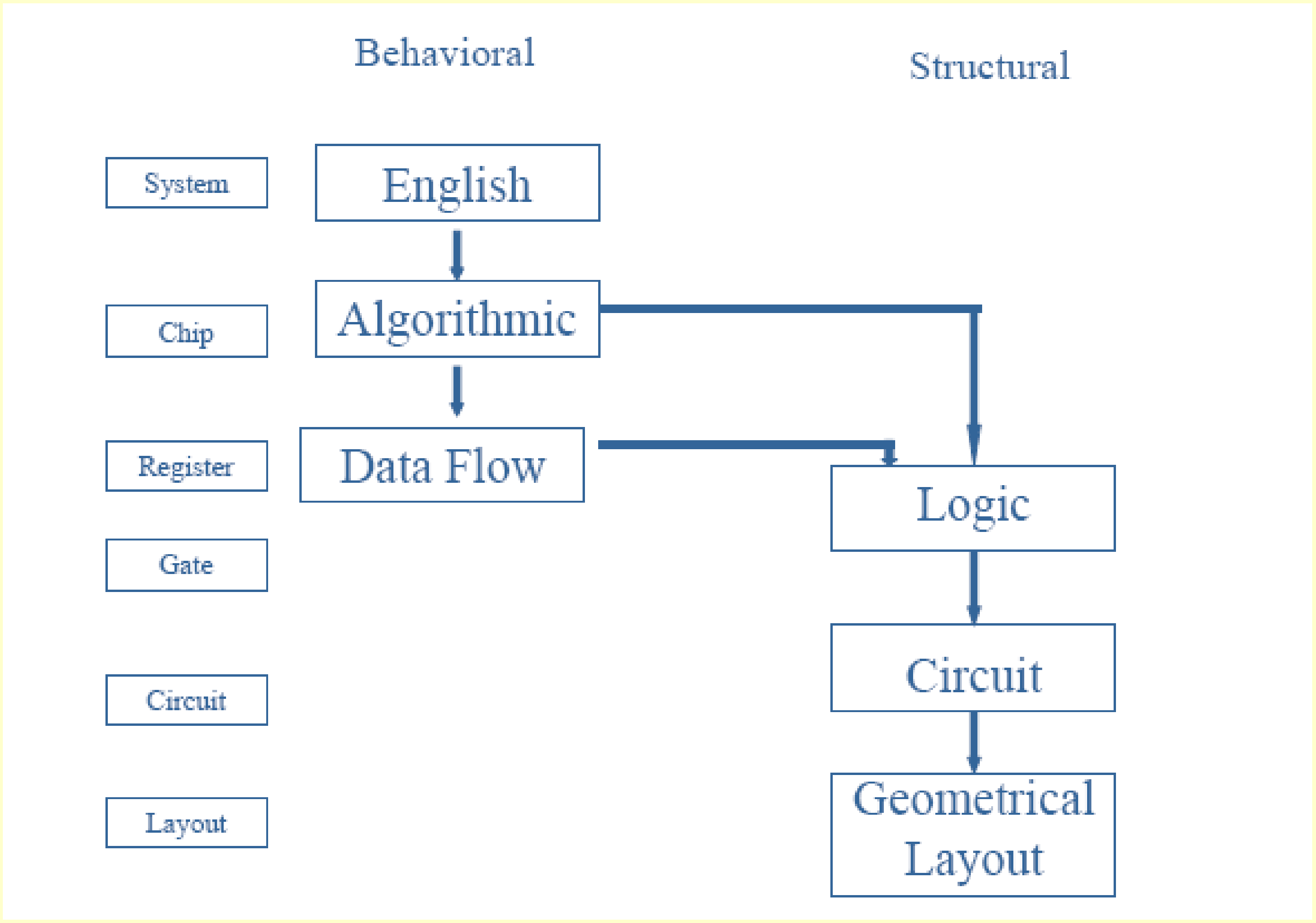
Register Level



Chip Level



Typical Design Track

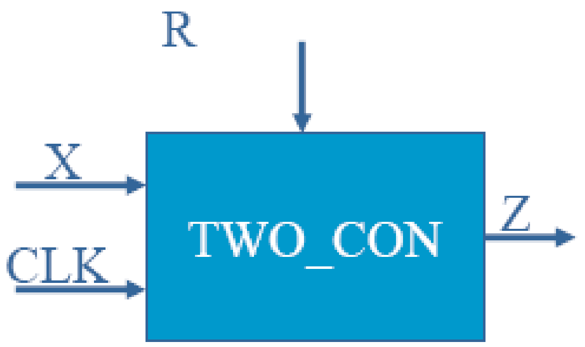


Design Representation

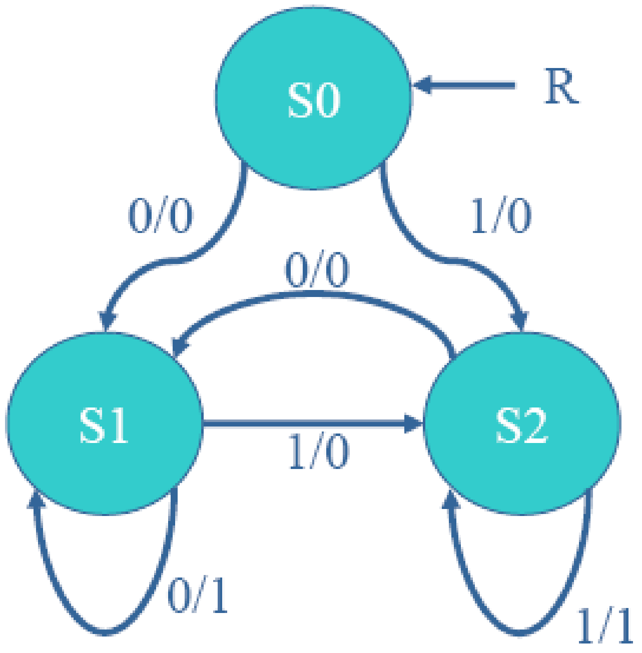
- Done in many ways
- Pictures
- Text
- *Is picture worth a thousand words?*

Design Representation Using Pictures

Specification: Detect inputs that are identical and in sequence

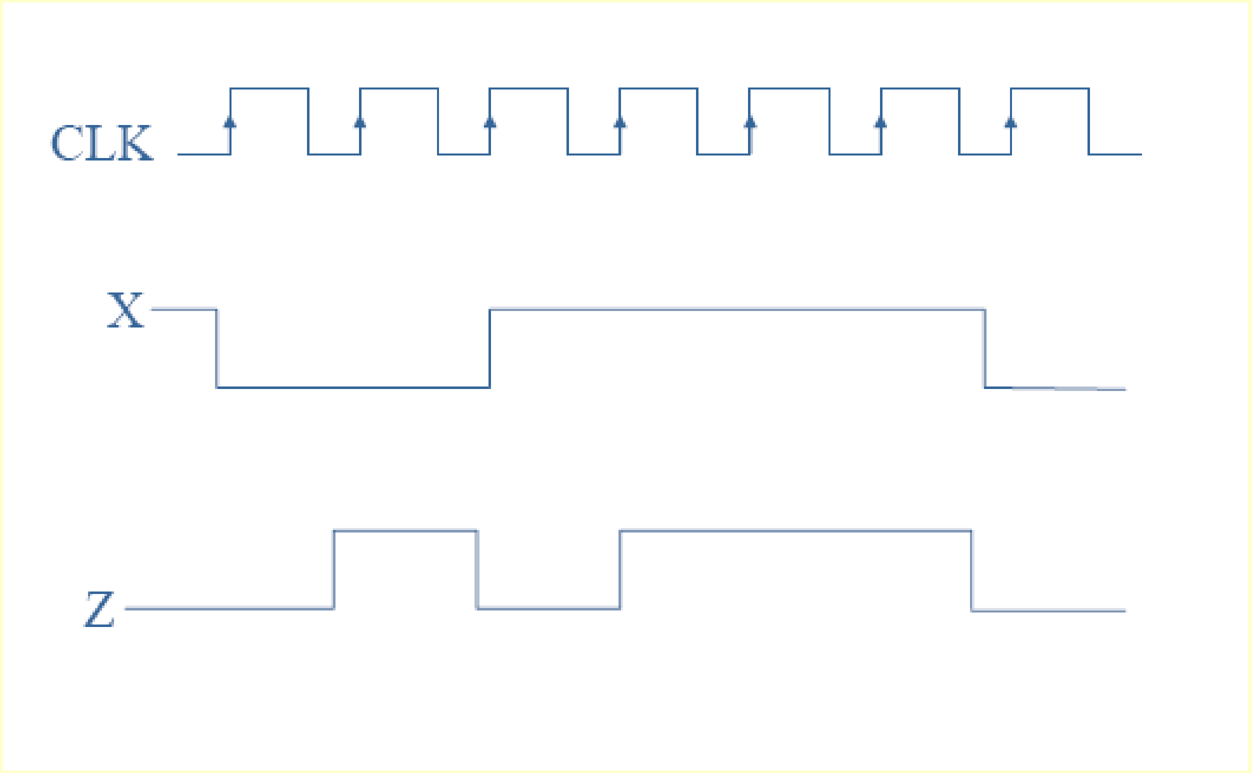


Block diagram

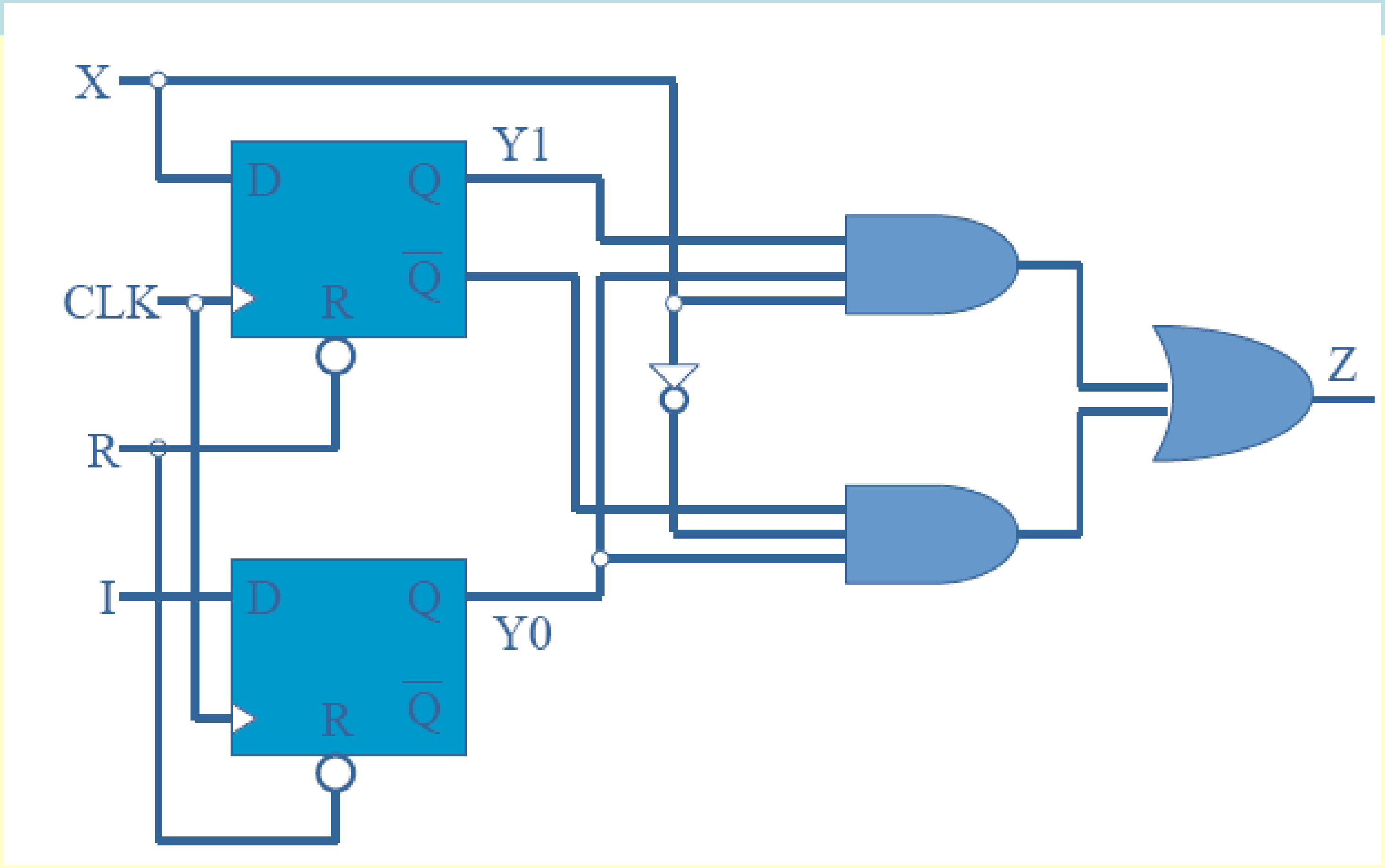


State diagram

As a Timing Diagram



As a Circuit

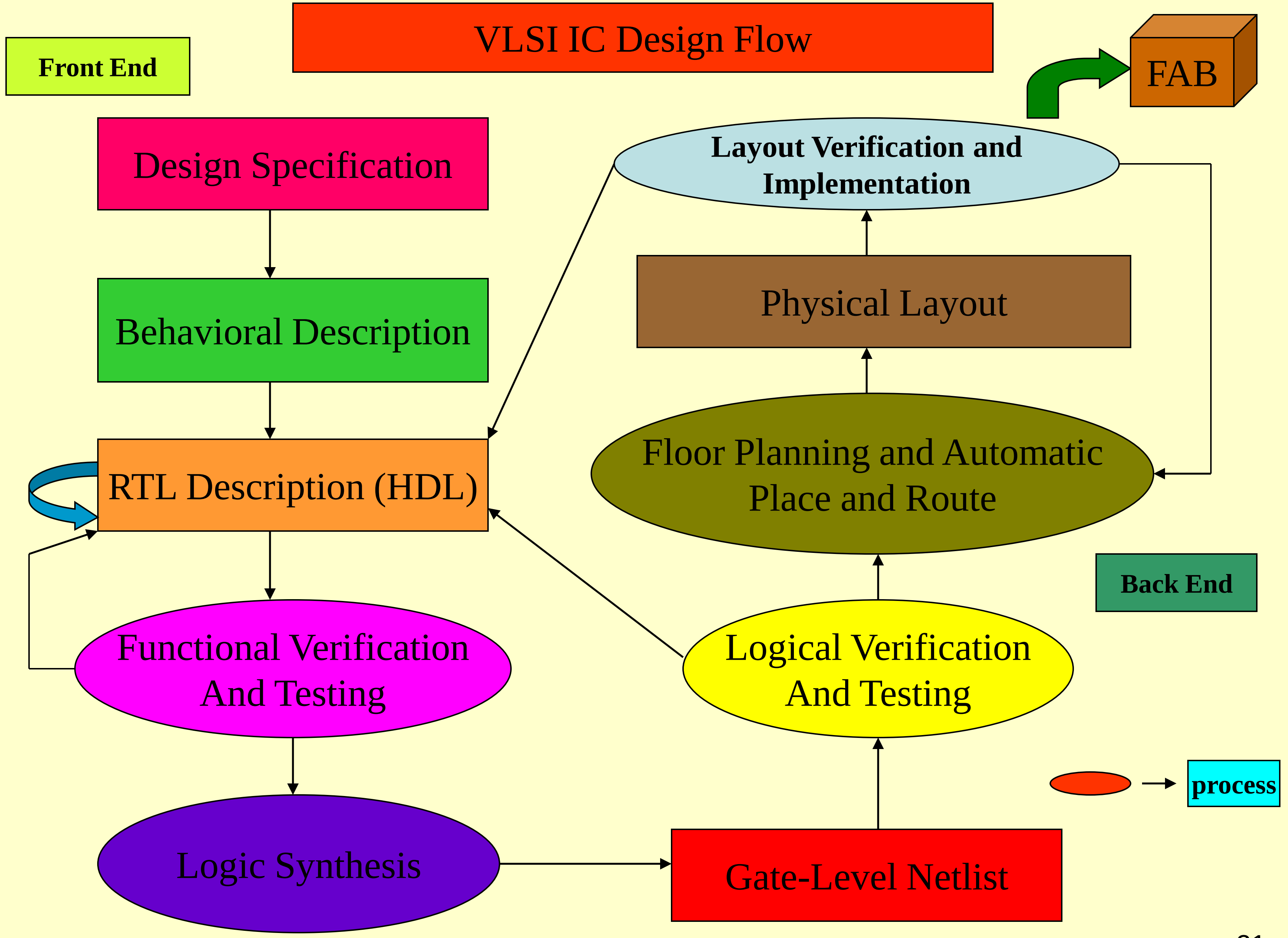


And in Verilog

```
module detector (Xin, clk, R, I, Zout);
input Xin, clk, R, I;
output Zout;
reg Y1, Y0;
    initial
    begin
            Y1 = 1'b0; Y0 = 1'b0;
    end
    always@(posedge clk or negedge R) begin
            if (R== 1'b0) begin
                        Y1 = 1'b0; Y0 = 1'b0;
            end
            else begin
                        Y1 = Xin; Y0 = I;
            end
    end

assign Zout =  Y0 & ((!Y1 & !Xin) | (Y1 & Xin));
endmodule
```

VLSI IC Design Flow


Front End
FAB
Design Specification
Layout Verification and Implementation
Behavioral Description
Physical Layout
RTL Description (HDL)
Floor Planning and Automatic Place and Route
Back End
Functional Verification And Testing
Logical Verification And Testing
process
Logic Synthesis
Gate-Level Netlist

HDL

- HDL stands for **Hardware Description Language**
- Definition : A high level programming language used to model hardware.
- Hardware Description Languages
  - have special hardware related constructs.
  - currently model digital systems, and in future can model analog systems also.
  - can be used to build models for simulation, synthesis and test.
  - have been extended to the system design level.

Why Use HDLs

- Allows textual representation of a design.
- High level language similar to C,C++.
- Can be used for Modeling at the
  - Gate Level
  - Register Level
  - Chip Level
- Can be used for many applications at the
  - Systems Level
  - Circuit Level
  - Switch Level
- Design decomposition is simple with HDLs and hence can manage complexity
- Early validation of designs.

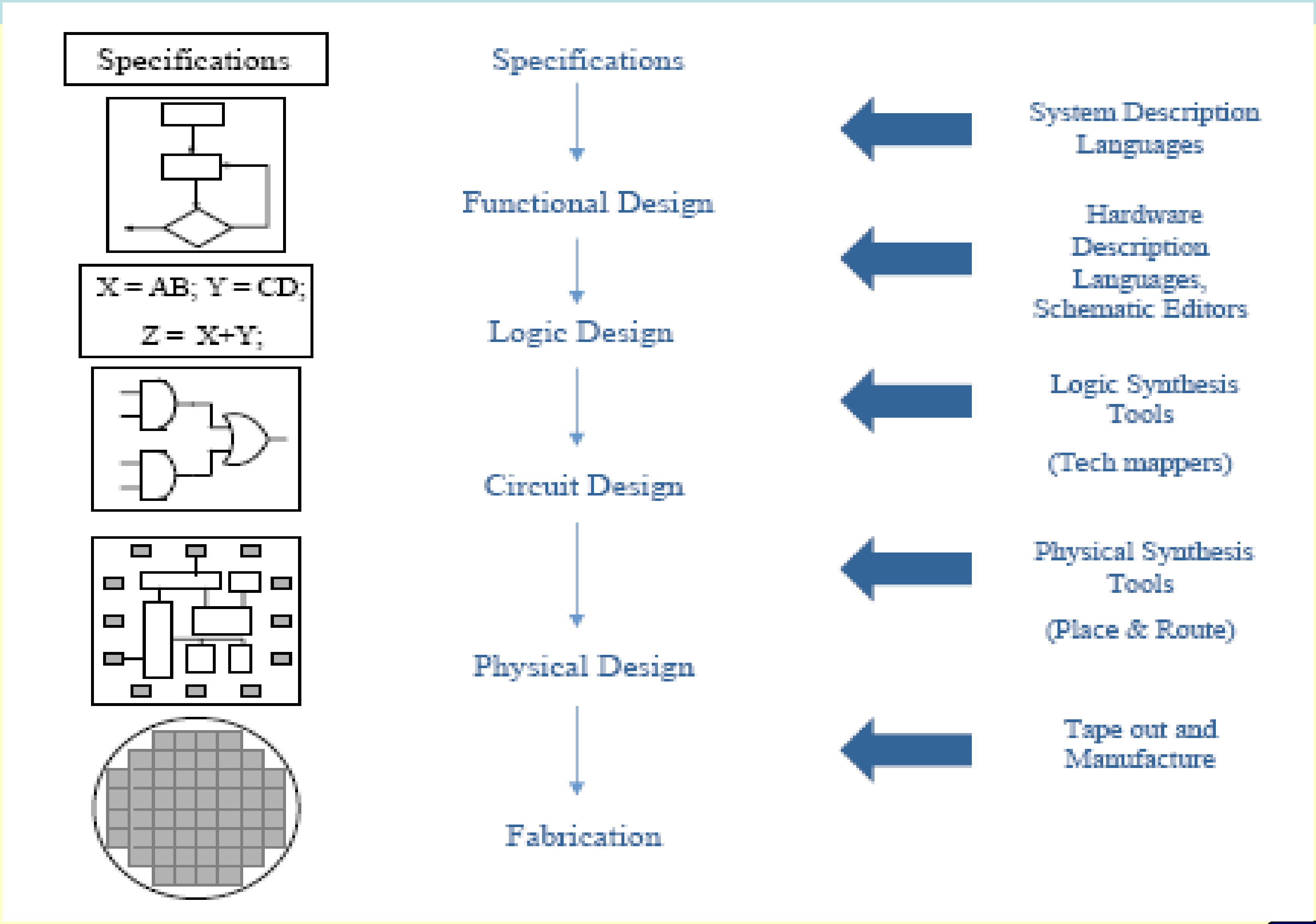
Need for Design Tools

- Current systems are very complex.
- Design abstraction and decomposition is done to manage complexity.
- Tools automate the process of converting your design from one abstraction level to another.
- Design Automation Tools improve productivity.
- Different tools are required in different steps.

Classification of CAD Tools

- Editors
  - Allows specification of the design either textually or graphically.
- Simulators
  - Models the response of a system to input stimuli.
- Analyzers
  - Used at different levels to check for correctness and compliance to rules.
- Synthesis
  - Transformation of representation between different abstraction levels.

Flow and Tools



CAD Tools -1 Design entry

- Graphical
  - Silicon Level – To create layouts
    - e.g. Magic
- Other Levels
  - e.g. ViewLogic, Protel
- Text
  - Natural language specification at system level.
  - Hardware Description Languages at Chip, Register and Gate levels.
    - e.g. VHDL, Verilog
- Circuit Level
  - e.g. SPICE

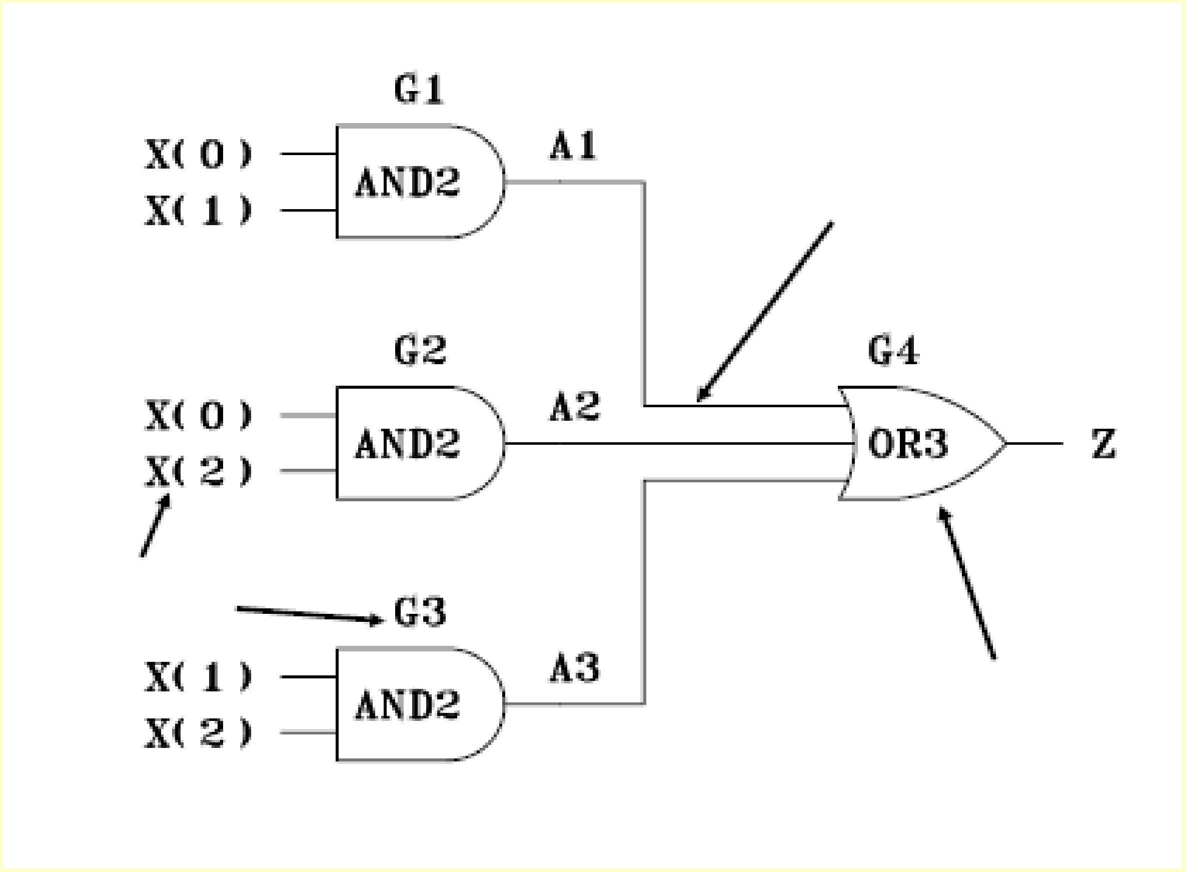
Graphical Editors

- Silicon Level editors are called Layout editors.
  - Draw rectangles describing metal, poly, diffusion etc
  - Library components are also at the same level.
  - Usually has online Design Rule Checking (DRC).
- Graphical Editors at other levels are usually called Schematic editors.
  - Used to create block diagrams and schematics.
  - The process is usually called *Schematic Capture*.

Schematic Editors

- Can create and display graphical components called “tokens”
- Can “interconnect” these tokens.
- Advantage :
  - Gives a structural representation called “netlist” describing the components used and their interconnections.
  - Also provides a simulation model to find the system’s response for different stimuli.

Example of Schematic Entry



Text based Design Entry

- Choose a specific HDL.
- Use text editors to describe the design.
  - e.g. vi, emacs, notepad etc.
  - Some tools have built-in editors
- Enter your design conforming to the language lexicon, syntax and semantics.
- Check for errors.
- “Compile” to get a simulation model.

What makes HDLs Different ?

- Hardware systems are concurrent in nature.
- Hardware systems may be distributed in nature.
  - Many components
  - Different rates for processing data, different clocks.
- Hardware systems are timed.
  - All hardware components have inherent delays and hence managing timing is crucial.
- Traditional software design techniques are insufficient.

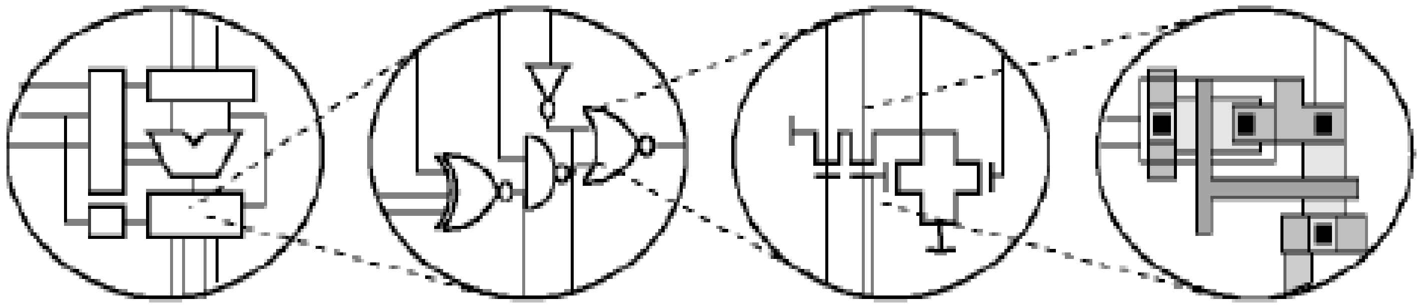
CAD Tools -2 Simulators

- Defn. : A program that models response of a system to the input stimuli.
- Simulation is widely used to establish design correctness.
- Types
  - Deterministic
  - Stochastic

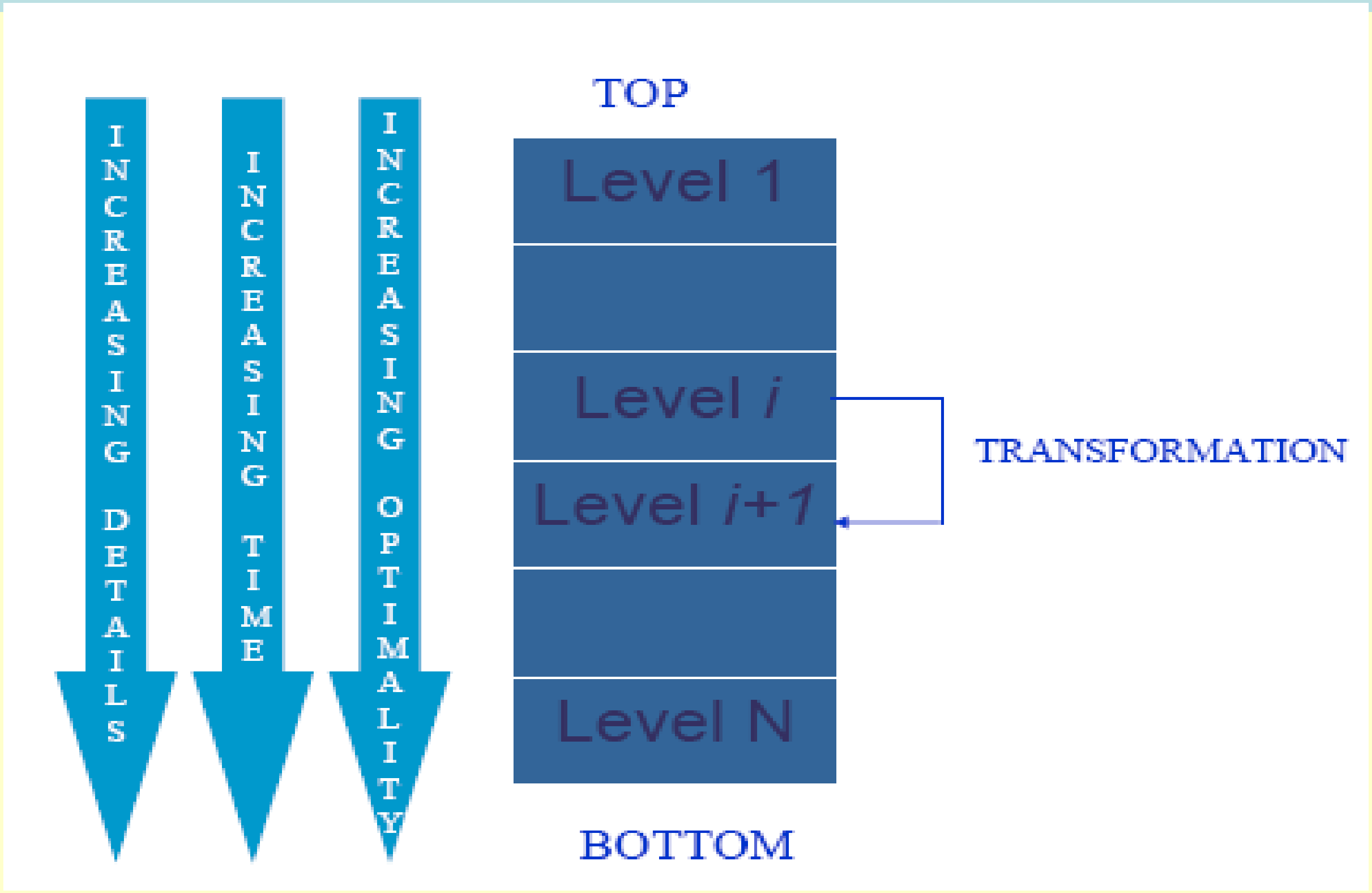
CAD Tools -3 Synthesis Tools

- Synthesis Definition : Transformation of a representation in one hierarchical level to another.
- Different names in different levels :
  - Algorithmic Synthesis – Abstract behavioral to register level or gate level specification
  - Logic Synthesis – RTL specification to gates
  - Physical Synthesis – Structural specification as gates to layout.

Synthesis at different Levels

Design Level	System Design	Logic Design	Circuit Design	Layout Design
Graphical Representation				
Elements	Blocks, sub–blocks	Logic gates	FETs, R, C, etc.	Geometric structures

Synthesis Transformations



VLSI :

Very Large Scale Integration

Very Large Source of Income

EDM – Course Evaluation

- Daily Evaluation – 30% weightage
- Internal Exams – 20% weightage
- Layout Design Practice – 15% weightage
- ASIC Design flow – 15% Weightage
- Project and FPGA Design flow – 20% Weightage

Verilog HDL

Lexical Conventions

- Similar to C
- Whitespace
  - Blank spaces, tabs and newlines
  - \b, \t, \n
  - Whitespace ignored except in
    - Strings and token separation
- Comments
  - // - single line;
  - /* … */ - multiple lines

Operators

- Unary,
  - A = ~b;
- Binary
  - a = b&&c;
- Ternary
  - a = b ? c : d;

Number Specifications

- • Sized numbers :

  <size>’<base format><number>

  – 4’b1111; 12’habc

  – <size> - number of bits
- Unsized numbers
  – 23456 // default is decimal
  – ‘hc3

Number Specification

- Negative numbers
  - <size>'<base format><number>
    - -6'd3 // 6-bit negative number stored as two's complement of 3
    - 4'd-2 //illegal specification

Possible Values

- X or Z values
  - 12’h13x; 6’hx; 32’bz;
  - x or z sets 1,3,4 bits in binary, octal and hexadecimal representations respectively
  - Value extension
    - If most significant bit is
      - 0,x,z then 0,x,z respectively
      - 1 then 0
- Data types
  - Value set – 0,1,x, z
  - Signal strength

Readability

- Readability enhancements
  - 12’b1111_1110_0101   //Underscore
- Strings – sequence of ASCII bytes
  - “Hello Verilog Word”;   // a string

Identifiers and Keywords

- reg value; // reg – keyword
- input clk; // input – keyword
- More keywords as we progress
- Identifiers made of alphanumeric characters, the underscore (_) and the dollar ($) sign and starts with alphabets or underscore.
- The dollar sign as first character is reserved for system tasks.
  - Ex : $monitor

Nets and Regs

- • Nets
  - – Connection between hardware elements
    - wire a; wire b,c; wire d = 1’b0;
    - wand, wor, tri, triand, trior and trireg are other forms of net
- Regs
  - – Data storage elements
  - – Not equivalent to hardware registers
  - – A variable that stores value
  - – Unlike a net, it needs no driver
  - – Default value for a ‘reg’ variable is x

Vectors

- Nets and register
- wire [7:0] busA, busB;
- reg [16:0] address;
- [high# : low#] or [low# : high#], but the most significant bit is the left number in the square bracket
- reg [0 : 40] addr1; wire [15 : 12] addr2;

Addressing Vectors

- Address bits are parts of vectors
  - reg [7:0] busA;
    - busA[7]; //stands for 7th bit
    - busA[2:0]; // first 3 LSBs
  - wire [0:15] addr1;
    - addr1[2:1] is illegal;
    - addr1[1:2] is correct;
  - Writing to a vector :
    - busA[2:0] = 3’d6;

Numbers

- Integers
  - datatype storing signed numbers
  - reg stores unsigned numbers
  - Default length is 32-bits
  - integer counter;
  - counter = -1; // stores as 32-bit two's complement number

Real Numbers

- Decimal and Scientific Notations
  - real delta;
    - delta = 4e10;
    - delta = 2.13;
  - integer a;
    - a = delta; /*‘a’ gets the value 2 which is the rounded value of 2.13*/

Time

- Verilog simulation is done with respect to simulation time
- To store simulation time, a special ‘time’ variable is declared in verilog
- The ‘time’ variable is at least 64-bits
- time save_sim_time; save_sim_time = $time;
- Useful for timing measurements
  - Debug

Arrays

- <array_name> [<subscript>]
  - reg bool[31:0];   //compare it with vectors
  - time chk_point [1:100];
  - integer count [0:7];
  - reg [4:0] port_id[0:7];
  - integer matrix_d[4:0][4:0] // illegal
  - count[5]; chk_point[100];
  - How to access say bit 3 of port_id[5];
    - reg [4:0] a;
    - a = port_id[5];   // a[3] is required bit

Arrays vs Vectors

- A vector is a single element that is n-bits wide;
- Arrays are multiple elements that are 1-bit or n-bit wide.

Memories

- reg mem1bit [0:1023];

  // a memory of 1024, 1-bit words
- reg [7:0] membyte [0:1023];

  // a memory of 1024, 8- bit words
- membyte [511];

  //fetches 1 byte word whose address is 511

Parameters

- Imagine this case
  - You design a 4-bit adder with HAs and FAs
  - You need 8-bit adders too
  - Rewrite is a waste of effort
    - Prone to bugs too
- Parameters
  - Method to define constants inside a module
  - For eg. parameter port_id = 5;
  - Can be overriden at module instantiation time

Strings

- Stored in ‘reg’ variables
  - each character takes 8- bits
- reg [8*19:1] string_value;
- string_value = “Hello Verilog world”;
- string_value is defined as a string that can store 19 characters.

System Tasks

- Vendor specific
- Some generic tasks
  - $display
    - similar to printf and used for displaying values of variables and expressions
  - Values can be printed in decimal (%d), binary (%b), string (%s), hex (%h), ASCII character (%c), octal (%o), real in scientific (%e), real in decimal (%f), real in scientific or decimal whichever is shorter (%g).
  - time format (%t), signal strength (%v) and hierarchy name (%m).

System Tasks

- $monitor
  - Monitor a signal when its value changes
  - Only one active monitor list
    - If more than one then, the last one statement will be the active one.
  - Monitoring turned ON by default at start of simulation and can be controlled during the simulation using **$monitoron** and **$monitoroff**

System Tasks

- $stop – stops simulation and puts it into interactive mode
- $finish – terminates the simulation

Continuous Assignments

- assign out = i1 & i2;
- assign addr[15:0] = addr1[15:0]^addr2[15:0];
- assign {cout,sum[3:0]} = a[3:0]+b[3:0]+c_in;
- wire out;   assign out = in1 & in2;
- is equivalent to
- wire out = in1 & in2; //Implicit continuous assignment

Expressions

- Dataflow modeling describes the design in terms of expressions instead of primitive gates.
- Expressions – those that combine operands and operators
- a ^ b;   addr1[20:17] + addr2[20:17];
- in1 | in2;

Operands

- Constants, integers, real numbers
- Nets, Registers
- Times
- Bit-select
  - One bit of a vector net or vector reg
- Part-select
  - Selected bits of vector net or vector reg
- Memories

Operators -Types

- Arithmetic
- Logical
- Relational
- Equality
- Bitwise
- Reduction // Not available in software languages
- Shift
- Concatenation
- Replication
- Conditional
- Syntax very similar to C

Arithmetic Operators

- Binary Operators

  * - multiply

  / - division

  + - addition

  - subtraction

- Commercial Verilog gives non ‘x’ values whenever possible

  a = 4’b0x11; b = 4’b1000; a+b = 4’b1x11;

  a = 4’b0x11; b = 4’b1100; a+b = 4’bxx11;

Logical Operators

- logical and (&&), logical or (||), logical not (!).
  - They evaluate to a 1-bit value: 0 (false) 1 (true) or x (ambiguous)
  - If an operand is not equal to zero, it is a logical 1 and if it is equal to zero, it is a logical 0.
  - If any operand bit is x or z, then operand is x and treated by simulators as a false condition
  - Logical operators take variables or expressions as operands.

Logical Operators Examples

- A = 3; B = 0;

  A&&B, A||B evaluates to 0 and 1 resp.

  !A, !B evaluates to 0 and 1 resp.

- A = 2’b0x; B = 2’b10;

  A&&B evaluates to x

- (a==2) && (b == 3) //Expressions

Relational Operators

- Greater-than (>)
- Less-than (<)
- Greater-than-or-equal-to (>=)
- Less-than-or-equal-to (<=)
- Evaluates to 1 or 0, depending on the values of the operands
  - If one of the bits is an ‘x’ or ‘z’, it evaluates to ‘x’

Relational Operators (Example)

- //A = 4, B = 3
- //X = 4’b1010, Y = 4’b1101, Z = 4’b1xxx
- A <= B //returns 0
- A > B //returns 1
- Y >= X //returns 1
- Y < Z //returns x

Equality Operators

- Logical equality (==), logical inequality (!=) :

  if one of the bits is ‘x’ or ‘z’, they output ‘x’ else returns ‘0’ or ‘1’
- Case equality (===), case inequality (!==) :

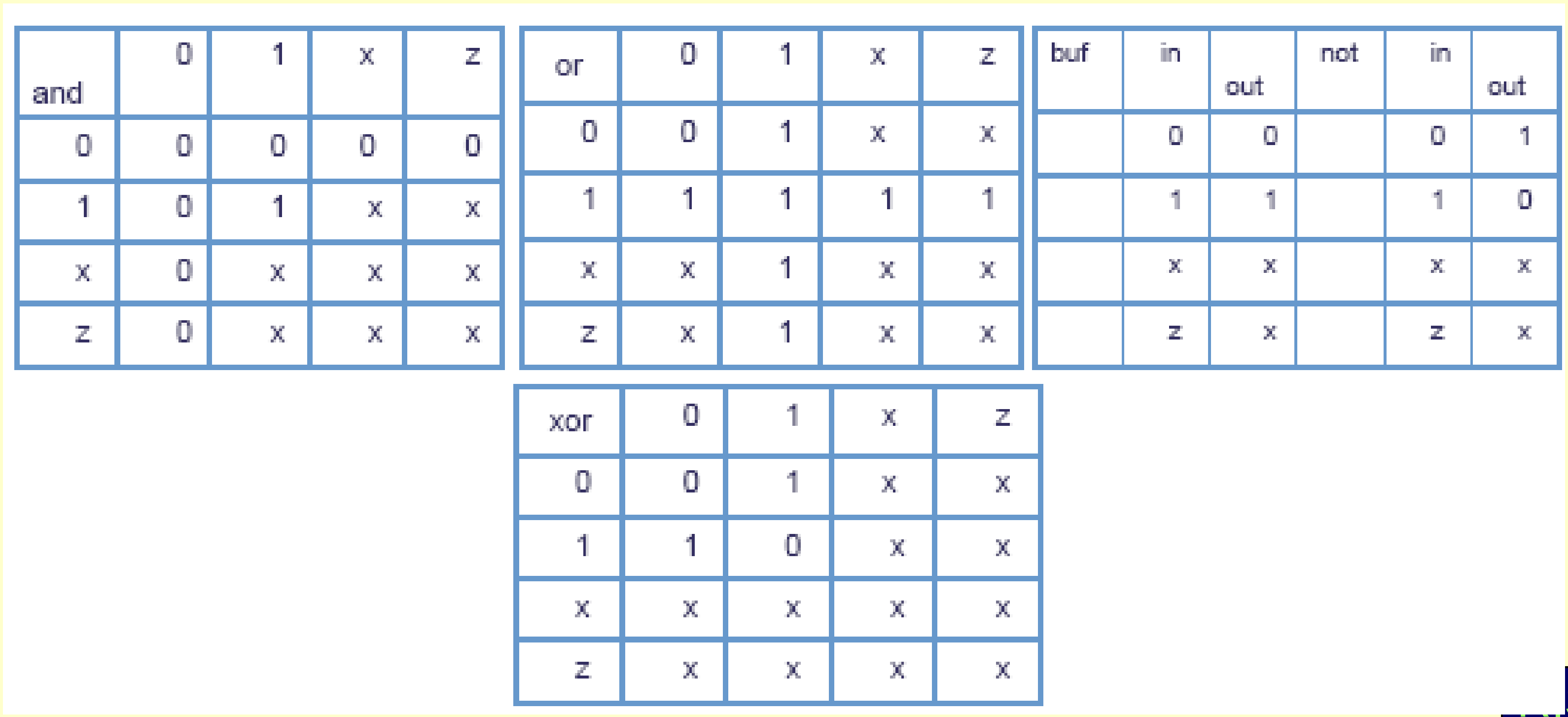
  compares both operands bit by bit and compare all bits including ‘x’ and ‘z’. Returns only ‘0’ or ‘1’

Equality Operators Examples

- //A = 4;B = 3;X = 4’b1010;Y = 4’b1101
- //Z = 4’b1xxz;M = 4’b1xxz;N = 4’b1xxx
- A == B // result is 0
- X != Y //result is 1
- X == Z // result is x
- Z === M //result is 1
- Z === N //result is 0
- M !== N //result is 1

Bitwise operators

- negation (~), and (&), or (|), xor (^), xnor(^~, ~^).
- ‘z’ is treated as ‘x’ in the bitwise operations



and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

buf	in	out	not	in	out
	0	0		0	1
	1	1		1	0
	x	x		x	x
	z	x		z	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Bitwise Operators Examples

- //X = 4’b1010;Y = 4’b1101;Z = 4’b10x1
- ~X // result is 4’b0101
- X & Y // result is 4’b1000
- X | Y // result is 4’b1111
- X ^ Y // result is 4’b0111
- X ^~ Y // result is 4’b1000
- X & Z // result is 4’b10x0

Point to Note

- We distinguish between bitwise operators and logical operators
- //X = 4’b1010;Y = 4’b0000
- X | Y // result is 4’b1010
- X || Y // result is 1

Reduction Operators

- Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- Reduction operators work bit by bit from right to left.

Reduction Operators Examples

- and (&), nand (~&), or (|), nor (~|), xor (^); and xnor (~^,^~)
- This is a UNARY operation on vectors
- Let X = 4’b1010
- &X is 1’b0   //0 & 1 & 0 & 1 = 1’b0
- |X is 1’b1
- ^X is 1’b0
- A reduction xor or xnor can be used for even or odd parity generation of a vector.

Shift Operators

- Right shift (>>) and left shift (<<)
- //X = 4’b1100
- Y = X >> 1; // Y is 4’b0110
- Y = X << 1; // Y is 4’b1000
- Y = X << 2; // Y is 4’b0000
- Very useful for modeling shift-and-add algorithms for multiplication.

Concatenation Operators

- Denoted by ({,})
- Append multiple 'sized' operands. Unsized operands are NOT allowed as size of each operand should be known to compute size of the result
- //A=1'b1;B=2'b00;C=2'b10;D=3'b110;
- Y = {B,C} // Y is 4'b0010
- Y = {A,B,C,D,3'b001} // Y = 11'b10010110001
- Y = {A, B[0],C[1]} // Y is 3'b101

Replication Operators

- Repetitive concatenation of the same number can be represented using a replication constant
- A = 1'b1; B=2'b00;
- Y = { 4{A} }; //Y is 4'b1111
- Y = {4{A},2{B}}; //Y is 8'b11110000

Conditional Operator

- Usage: condition_expr? true_expr : false_expr;
- If condition evaluates to ‘x’, then both expressions are evaluated and compared bit by bit to return for each bit position, an ‘x’ if the bits disagree, else the value of the bit.
- The conditional expression models a 2-to-1 multiplexer
- assign out = control ? in1 : in0;

Compiler Directives

- Usage : `<keyword>
- `define WORD_SIZE 32; compared with #define in C
- `include header.v compared with #include in C.
- `ifdef and `timescale

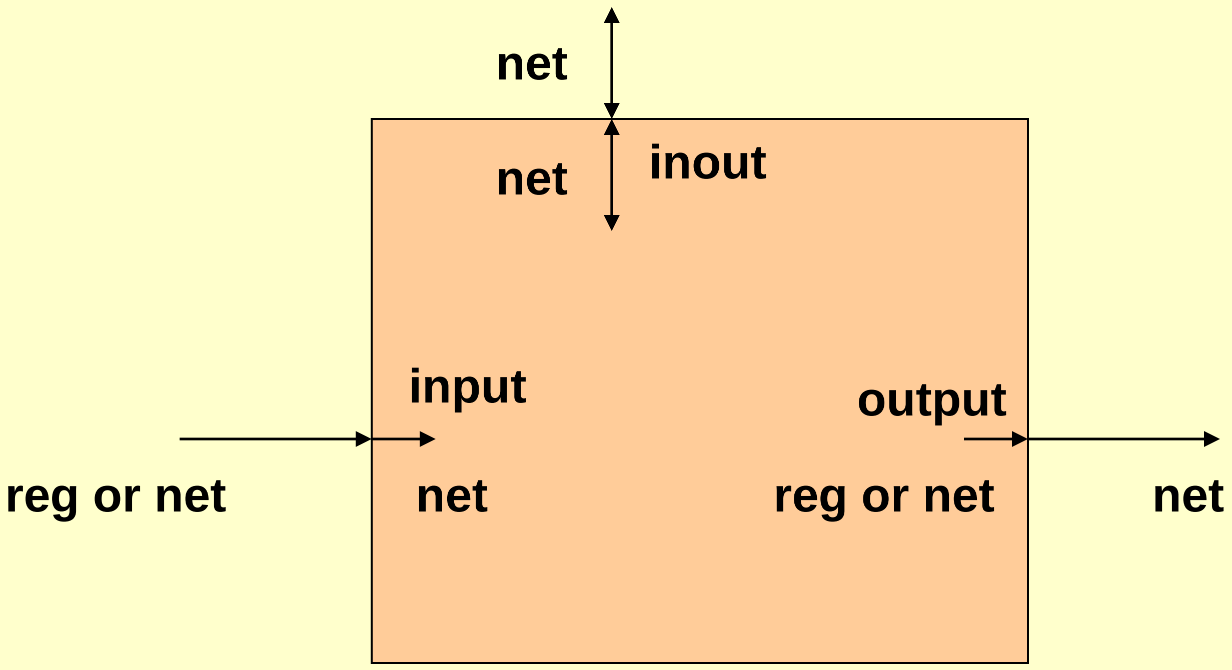
Modules

- General Structure
  - Module name, port declarations, parameters (optional)
  - Declaration of wire and reg variables
  - Data flow statements (assign)
  - Instantiation of lower level modules
  - always and initial blocks (all behavioral statements)
  - Tasks and Functions
  - endmodule

Ports

- Port declarations – input, output, inout
- Port connection rules
  - there are two ends to a port with respect to a module, one internal and another external
  - Inputs – input ports are to be Nets inside the module and can be reg or net external to the module

Ports Connection Rules



Port Connection Rules

- Outputs – Internal reg or net and external should be a net
- Inouts – Both internal and external to be connected to a net
- Width matching – width may not match – only warning issued
- Unconnected ports – again only warning
  - fulladd4(SUM, , A,B, C_IN); the C_OUT is not included

Ports and external Connections

- Connecting by ordered list
- Connecting by name
  - wire C_OUT; wire [3:0] SUM;
  - reg [3:0] A,B; wire C_IN;
  - Fulladd4 f1 (.c_out(C_OUT), .sum(SUM), .a(A), .b(B), .c_in(C_IN));
- Syntax
  - .< name in instantiated module>(name in instantiating module)

Connecting by Name

- Advantages
  - Remembering order of say, 50 ports is difficult
  - Can drop any port during instantiation.
  - Can rearrange the port list of a module without modifying the code that instantiates it.

Verilog Modeling Techniques

- Gate Level Modeling
- Dataflow Modeling (RTL)
- Behavioral Modeling (Algorithmic level)

Gate Level Modeling

- Gate level modeling
  - Logic gate primitives and their instantiation
  - Construct Verilog description from the logic diagram
  - Delay modeling (rise, fall and turn-off)
  - min, max and typ delays in gate level design

Gate Types

- wire OUT, IN1, IN2, IN3;
- and a1(OUT,IN1,IN2);
- //similarly nand, nor, or, xor and xnor
- //extended to more than two inputs
- nand na1_3input(OUT, IN1,IN2,IN3);
- //Name for an instantiation is optional
- and (OUT, IN1, IN2);

Gate Delays

- Rise Delay - 0,x or z → 1
- Fall Delay - 1,x or z → 0
- Turn-off delay – 0, 1 or x → z
- Min of above three for – 0, 1 or z → x

Delay Specification

- No delay specified then default is 0
- If one delay, then used for all transitions
- If two delays, then refers to
  - rise and
  - fall.
  - Turn-off is min of these two.
- If three delays, then refers to
  - rise,
  - fall and
  - turn-off .

Delay Specification Example

- and #(5) a1(out, i1, i2);
- and #(4, 6) a2(out, i1, i2);
- bufif0 #(3, 4, 5) b1 (out,i1,i2);

- assign #10 out = in1 & in2;
- wire #10 out = in1 & in2;
- wire #10 out;
- assign out = in1 & in2;

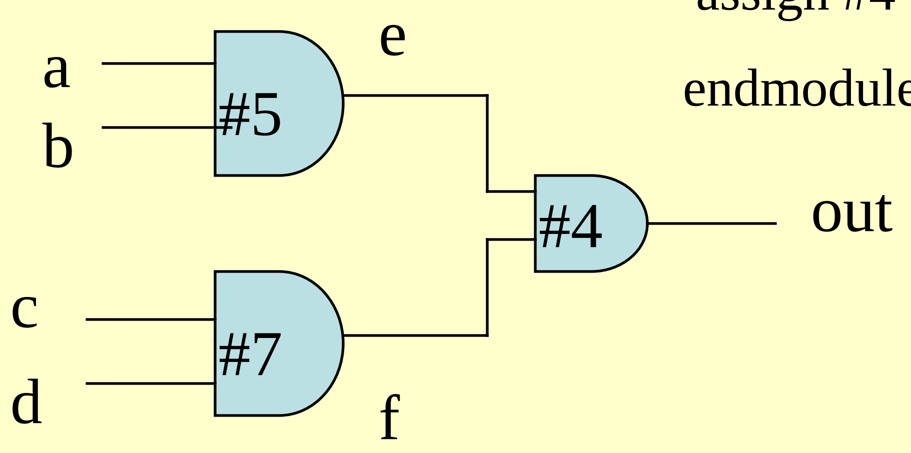
Delay models

- Three types of delay models used in Verilog
  - Distributed delay model
  - Lumped Delay model
  - Pin-to-pin (path) Delay model

Distributed Delay Model

- Delays that are specified on a *per element* basis

```
module M (out,a,b,c,d);
output out;
input a,b,c,d;
wire e,f;
and #5 a1(e,a,b);
and #7 a2(f,c,d);
and #4 a3(out,e,f);
endmodule
```



```
module M(out,a,b,c,d);
output out;
input a,b,c,d;
wire e,f;
assign #5 e = a & b;
assign #7 f = c & d;
assign #4 out = e & f;
endmodule
```

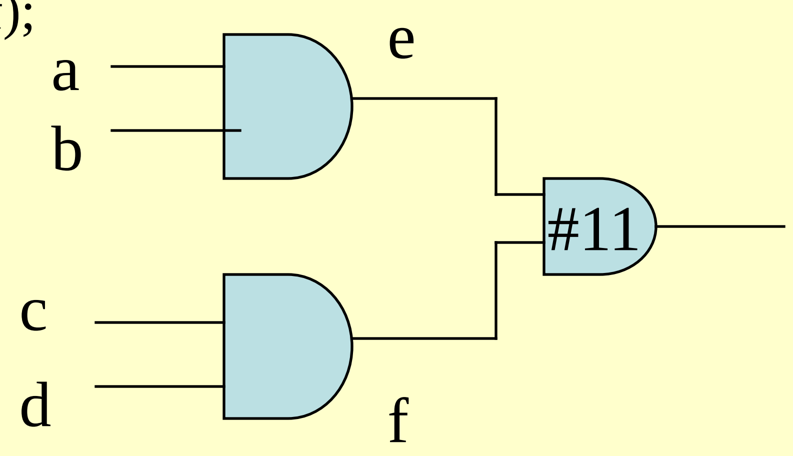
Lumped delays

- Lumped delays are specified on a *per module* basis.
- Single delay on the output gate of the module – cumulative delays of all paths is lumped at one location.
- They are easy to model compared with distributed delays

Lumped Delay Model Example

```
module M (out,a,b,c,d);
output out;
input a,b,c,d;
wire e,f;
and  a1(e,a,b);
and a2(f,c,d);
and #11 a3(out,e,f);
endmodule
```

```
module M(out,a,b,c,d);
output out;
input a,b,c,d;
wire e,f;
assign  e = a & b;
assign  f = c & d;
assign #11 out = e & f;
endmodule
```



Pin-to-Pin Delays

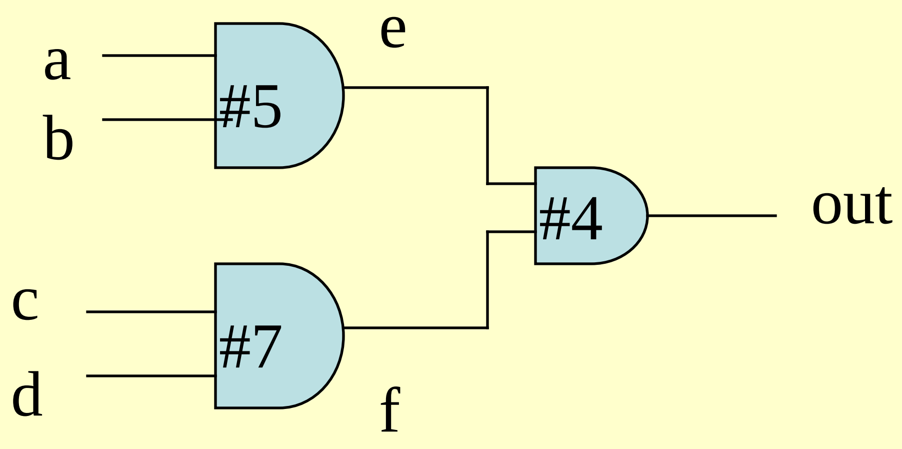
- Delays are assigned individually to paths from each input to each output.
- Delays can be separately specified for each input/output path

path a-e-out, delay = 9

path b-e-out, delay = 9

path c-f-out, delay = 11

path d-f-out, delay = 11



Behavioral Modeling

- Learning Objectives
  - Use of structured procedures – **always** and **initial**
  - Delay-based, Event-based and Level-sensitive timing controls
  - Conditional statements – **if** and **else**
  - Multiway branching – **case**, **casex** and **casez**
  - Looping Statements – **while**, **for**, **repeat** and **forever**
  - Blocks – **sequential** and **parallel** blocks, **naming** and **disabling** blocks.

The Initial Statement

- An initial block – (initial begin … end) may consist of a group of statements (within begin … end) or one statement
- Each initial block starts at execution time 0
- All initial blocks starts concurrently executing and finishes independent of each other.
- Typically used for initialization, monitoring, waveforms and to execute one-time executable processes.

The always Statement

- Model a block of activity that is repeated continuously in a digital circuit
- Equivalent to an infinite loop
- Stopped only by $finish (power-off) or $stop (interrupt)

Timing Controls

- Delay-Based
  - Regular delay control
  - Intra-assignment delay control
  - Zero delay control

Regular Delay Control

```
parameter latency = 20;
parameter delta = 2;
reg x, y, z, p, q;

initial
begin
        x=0; // no delay control
        #10 y = 1; //delay control with a constant
        #latency z = 0; //delay control with identifier
        #(latency + delta) p = 1; //delay control with expression
        #y x = x + 1; //delay control with identifier
        #(4:5:6) q = 0; //delay with min, typ, and max values
end
```

Intra-assignment delay control

- Assigning delay to the right of the assignment operator
- The intra-assignment delay computes the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Equivalent to regular delays with a temporary variable to store the current value of a right-hand-side expression

Intra-assignment Delay

```
reg x,y,z;
 initial
 begin
            x = 0; z = 0;
            y = #5 x + z;
            //Take value of x and z at the time = 0, evaluate x+z and then
            //wait 5 time units to assign value to y.
            //Any  change to x and z after time = 0 and before 5 will not affect
             the value of y
  end
//The above code is equivalent to
//x = 0; y = 0;                    temp_xz = x + z;
                                   #5 y = temp_xz;
// where, temp_xz is a temporary variable
```

Zero delay control

- Procedural statements inside different always-initial blocks may be evaluated at the same simulation time.
- The order of execution of these statements in different always-initial blocks is nondeterministic and might lead to race conditions.
- Zero delay control ensures that a statement is executed last, after all other statements in that simulation time are executed.
- This can eliminate race conditions, unless there are multiple zero delay statements, in which case again non-determinism is introduced.

Zero delay Control

```
initial
 begin
          x = 0;
          y = 0;
 end
 initial
 begin
          #0 x = 1; //zero delay control
          #0 y = 1;
 end
          //At time = 0, x = 1 and y = 1
```

Event-based timing control

- Regular event control
- Named event control
- Event OR control

Regular Event Control

- @(clock) q = d; // q=d is executed whenever clock signal changes value;
- @(posedge clock) q = d; // q=d whenever clock does positive transition (0 to 1, x or z, x to 1, z to 1)
- @(negegde clock) q = d; // q=d whenever clock does negative transition (1 to 0, x or z, x to 0, z to 0)
- q = @(posedge clock) d; // d is evaluated immediately and assigned to q at the positive edge of clock

Named event control

```
event received_data; //define an event called
  received_data
always @(posedge clock)
begin
        if (last_data_packet)
                received_data = 1;  // trigger the event
end
always @(received_data)
   data_buf = {data_pkt[0], data_pkt[1]};
```

Event OR control

```
// A level sensitive latch with asynchronous reset
// Sensitivity list reset, clock and d
 always @(reset or clock or d)
      // if any one of reset, clock, d changes its value
 begin
         if (reset)
                   q = 1'b0;
         else  if (clock)
                   q = d;
 end
```

Sensitivity List with Comma Operator

- always @(reset, clock, d)

```
begin
    if(reset)
        q = 1'b0;
    else if(clock)
        q = d;
end
```

- always @(posedge clk, negedge reset)

```
if(!reset)
    q <= 0;
else
    q <= d;
```

Use of @* Operator

- // combination logic block using the or operator
- always @(a or b or c or d or e or f or g or p or m)

```
begin
      out1 = a ? b + c : d + e;
      out2 = f ? g + h : p + m;
end
```

- // Instead of above method use @(*) symbol
- always @(*)

```
begin
      out1 = a ? b + c : d + e;
      out2 = f ? g + h : p + m;
end
```

Level Sensitive Timing Control

- // Ability to wait for a certain condition to be true before a statement or a block of statements is executed
- Keyword “wait” is used for level sensitive constructs
- always

```
      wait (count_enable) #20 count = count+1;
```

// if count_enable = 1; count is incremented every 20 time units

Conditional Statements

- The if-else statement
- if (logical_expression) then <block> else <block>
- <block> = single statement or **begin** <block> **end**
- Nested if-else
  - if <block> else if <block> else if <block

Conditional Statements

- // Type 1: No else statement
- If (expression)

```
begin
    true_statement;
end
```

Conditional Statements

- // Type 2: One else statement
- If (expression)

```
If (expression)
      true_statement;
else
      false_statement;
```

Conditional Statements

- // Type 3: Nested if-else-if statement
- If (expression1)

```
        true_statement1;
else if (expression2)
        true_statement2;
else if (expression3)
        true_statement3;
else
        default_statement;
```

Multiway Branching

- The case statement
- The casex and casez statements

- case (expression)

```
        alternative1: statement1;
        alternative2: statement2;
        ….
        default: default_statement;
endcase
```

The case statement

- This compares 0,1,x and z of the condition, bit by bit with the different case options.
- If width of condition and a case option are unequal, they are zero filled to match the bit width of the widest of both.

The case Statement

```
//A 4-to-1 Multiplexer
 always @(s1 or s0 or i0 or i1 or i2 or i3)
 case ({s1,s0})
          2’d0: out = i0;
          2’d1: out = i1;
          2’d2: out = i2;
          2’d3: out = i3;
          2’bx0, 2’bx1, 2’bxx,2’b0x,2’b1x: out = 1’bx;
          2’bxz, 2’bzx, 2’bzz: out = 1’bz;
   default: $display(“Invalid Control signals”);
 endcase
```

The casex and casez statements

- casez does not compare z-values in the condition and case options. All bit positions with z may be represented as ? in that position.
- casex does not compare both x and z-values in the condition and case options

```
casex (encoding)
        4'b1xxx: next_state = 3;
        4'bx1xx: next_state = 2;
        4'bxx1x: next_state = 1;
        4'bxxx1: next_state = 0;
 endcase
//encoding = 4'b10xz will cause
   next_state=3
```

Loops

- while (condition) <block>
- for (count = 0; count < 128; count = count + 1) <block>
- repeat (value) <block>
  - The value should be a constant or variable, and the evaluation of the variable will be done at start of loop and not during execution
- forever loop
  - initial
  - begin
  - clk = 1’b0;
  - forever #10 clock = ~clock;
  - end

While Loop

- Integer count;
  initial
  begin
  count = 0;
  while (count < 128) // execute loop till count is 127 exit at count 128
  begin
  $display (“Count = %d”, count);
  count = count + 1;
  end
  end

```
Integer count;
initial
begin
      count = 0;
      while (count  < 128) // execute loop till count
is 127 exit at count 128
      begin
            $display (“Count = %d”, count);
            count = count + 1;
      end
end
```

For Loop

- integer count;

```
initial
    for (count=0; count<128; count=count+1)
        $display("Count = %d", count);
```

Repeat Loop

- // increment and display count from 0 to 127

- integer count;

```
integer count;
initial
begin
        count = 0;
        repeat(128);
        begin
                $display ("Count = %d", count);
                count = count + 1;
        end
end
```

Forever Loop

- // Clock Generation
- // Use forever loop instead of always block

```
reg clock;
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock; // clock with
end                              //period of 20 units
```

Forever Loop

- //Synchronize two register values at every positive edge of clock

```
reg clock
reg x, y;
initial
    forever @(posedge clock) x = y;
```

Sequential Blocks

- The statements in a sequential block are processed in the order they are specified.
- A statement is executed only after its preceding statement completes execution.
- Except Non-blocking assignment with intra-assignment timing control.
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution

Sequential Blocks

- // Keywords begin and end are used to group statements into sequential blocks

```
// Sequential block without delay
    reg x, y;
    reg [1:0]  z, w;
    initial
    begin
          x = 1’b0;
          y = 1’b1;
          z = {x, y};
          w = {y, x};
    end
```

Sequential Blocks

- // Sequential blocks with delay

```
reg x, y;
reg [1:0]  z, w;
initial
begin
    x=1’b0; // completes at simulation time 0
    #5 y = 1’b1; // completes at time 5
    #10 z={x,y}; //completes at time 15
    #20 w={y, x}; //completes at time 35
end
```

Parallel Block

- Specified by fork and join
- Statements in a parallel block are executed concurrently
- Ordering of the statement controlled by the delay or event control assigned to each statement
- If delay or event control is specified, it is relative to the time the block was entered.

Parallel Blocks

- // Parallel blocks with delay

```
reg x, y;
reg [1:0]  z, w;
initial
fork
     x =1’b0;  //completes at time 0
     #5 y = 1’b1; // completes at time 5
     #10 z = {x, y}; // completes at time 10
     #20 w = x & y; // completes at time 20
join
```

Parallel Blocks

- // Parallel blocks with deliberate race condition

```
reg x, y;
reg [1:0] z, w;
initial
fork
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};
join
```

// Variables z and w will get values 1 and if x and y executes first

// Variables z and w will get values 2'bxx and 2'bxx if x and y executes last

// Results of z and w are non deterministic

// Its values depends on the simulator

Nested Blocks

- // Sequential and Parallel blocks can be mixed

```
initial
begin
    x = 1’b0;
    fork
        #5 y = 1’b1;
        #10 z = {x, y};
    join
    #20 w = {y, x};
end
```

Named Blocks

- Blocks can be given Names.
- Local variables can be declared for the named block.
- Named blocks are a part of the design hierarchy.
- Variables in a named block can be accessed by using hierarchical name referencing.
- Named blocks can be disabled, i.e., their execution can be stopped.

Named Blocks

```
initial
begin: block1 //Sequential block named block1
integer i;
    .....
disable block1;
end
    initial
    fork: block2
    reg i;
    ....
join
```

Procedural Assignment

- Two types
  - Blocking Assignments
  - Non Blocking Assignments

Blocking Assignments

They are denoted by ‘=‘ operator.

In a sequential block it blocks any statement beyond it.

```
reg x,y,z
        x = 0;
        #5 x = 1;     //After 5 units
        #5 y = 1;     //After 10 units
        z = #5 x;     // After 15 units
        #5 x = 0;     // After 20 units
```

In a parallel block, however it does not block.

Non blocking Assignments

- These assignments do not block execution of statements that follow them in a sequential block.
- Denoted by <=

```
    x = 0;
reg_a[2] <= #15 1'b1; // at 15 units
reg_b[15:13] <= #10 {x,y,z} // at 10 units
count <= count + 1; // at 0 units
```

Applications of Non Blocking

- Distinguish between the following
- Type 1 // Race condition – Simulator dependent
  - always @(posedge clock)
  - a=b;
  - always @(posedge clock)
  - b = a;
- Type 2 // ‘a’ swaps with ‘b’ – No Race condition
  - always @(posedge clock)
  - a <= b;
  - always @(posedge clock)
  - b <= a;

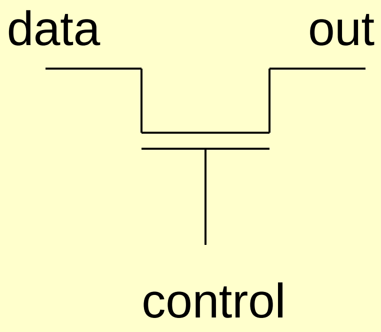
Non-Blocking Assignment using Blocking Assignment

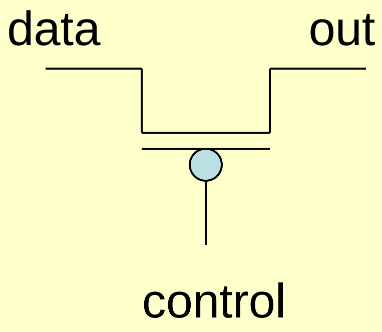
- //Using the temporary variables and blocking assignments

```
always @(posedge clock)
begin
// store right hand side expression in temporary variables
	temp_a = a;
	temp_b = b;
//Assign values of temporary variables to left hand side
	a = temp_a;
	b = temp_b;
end
```

Switch-level Modeling

- MOS switches
- nmos n1(out,data,control)
- pmos p1(out,data,control)





CMOS switch

- cmos (out, data, ncontrol, pcontrol)
- ncontrol = 1 for passing
- Equivalent to
  - nmos(out,data,ncontrol)
  - pmos(out,data,pcontrol)
- Power and Ground
  - supply1 vdd;
  - supply0 gnd;
  - assign a = vdd;
  - assign b = gnd;

Synthesis of Verilog HDL Constructs

Synthesizable Verilog Constructs

Construct Type	Keyword or Description	Notes
ports	Input, inout, output	
parameters	parameter	
Module definition	module	
Signals and variables	wire, reg, tri	Vectors are allowed

Synthesizable Verilog Constructs

Construct Type	Keyword or Description	Notes
instantiation	Module instances Primitive gate instances	Eg. my_mux m1(out,i0,i1,s);
Functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported

Synthesizable Verilog Constructs

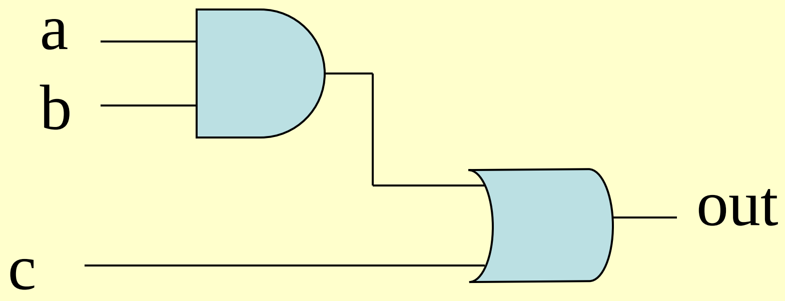
Construct Type	Keyword or Description	Notes
Procedural blocks	begin, end, named blocks, disable	Disabling of named blocks not allowed
Data flow	assign	Delay information is ignored
loops	for, while, forever	while and forever loops must contain @(posedge clk) or @(negedge clk)

Synthesizable Verilog Operators

- All arithmetic, logical, relational, equality (except ===, !==), bit-wise, reduction, shift, concatenation and conditional are synthesizable.

Simple Examples

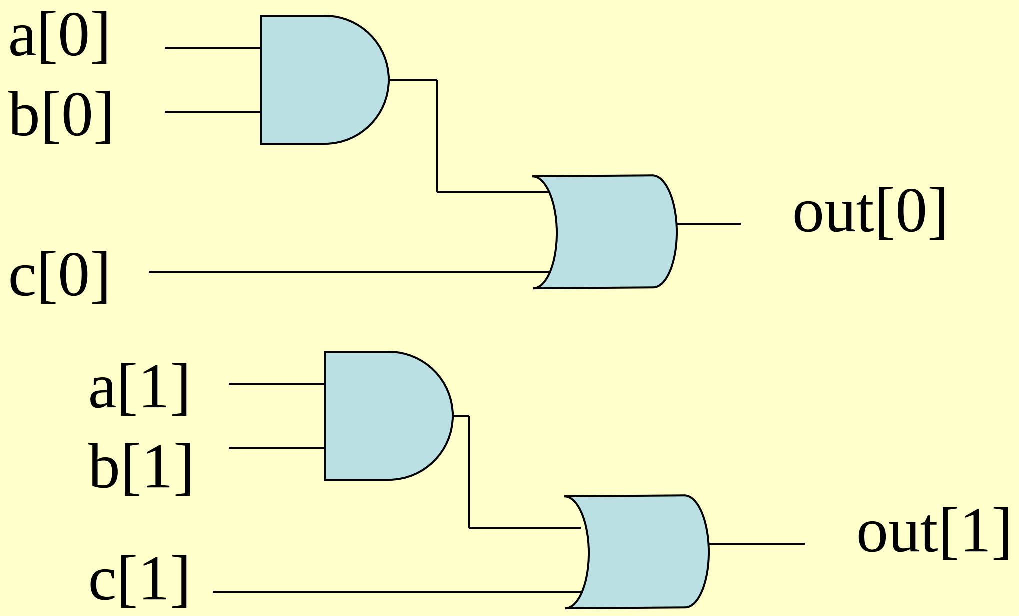
```
module m (out, a, b, c );
input a, b, c;
output out;
   assign out = (a & b) | c;
endmodule
```



Simple Examples

```
// If all are 2-bit vectors


module m (out , a, b, c);
input [1:0] a, b, c;
output [1:0] out;
 assign out = (a & b) | c;
endmodule
```



Simple Examples

- assign {c_out,sum} = a + b + c_in;

  // yields a Full-adder

- assign out = (s) ? i1 : i0;

  // yields a 2-to-1 multiplexer. Always the '?' construct yields a multiplexer.

-
  ```
  if (s)
      out = i1;
  else
  out = i0;    // has the same effect
  ```

Simple Examples

- case (s)

  1’b0: out = i0;

  1’b1: out = i1;

  endcase //yields the same result(2:1 Mux)
- Large ‘case’ statements may be used to infer large multiplexers

Simple Examples

- The *for* loops may be used to form cascaded combinational logic

```
c = c_in;
for (i=0; i <= 7; i = i + 1)
        {c, sum[i]} = a[i] + b[i] + c;
c_out = c;
```

- //8-bit ripple adder

Simple Examples

- *always* statement infers both sequential and combinational circuits.

-
  ```
  always @(posedge clk)
    q = d;        //This is a flipflop
  ```

-
  ```
  always @(clk or d)
  if (clk)
        q = d;   // This is a latch
  ```

Simple Examples

- always @(a or b or c_in)

  {c_out, sum} = a + b + c_in;

  //This yields a full adder
- The *function* statement synthesize to combinational blocks with one output variable. The output may be a scalar or a vector.

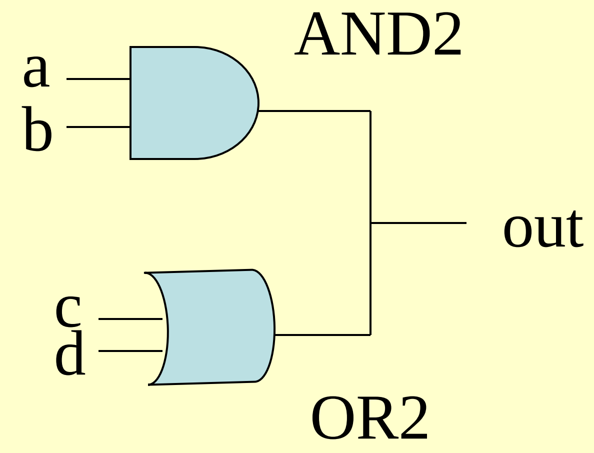
Logic Value System

- 0 < -- > logic-0
- 1 < -- > logic-1
- z < -- > high-impedance
- x < -- > don’t-care (when assigned to a variable)
- z < -- > don’t-care **(in casex and casez statement)**
- x < -- > unknown (when not assigned to a variable)

Data types

- Net data type
  - wire, wor, wand, tri, supply0, supply1
  - Bit-width is specified explicitly, else the default size is one bit.

```
module wireexample (out, a, b, c, d);
input a,b,c,d;
output out;
assign out = a & b;
assign out = c | d;
endmodule
```



The Modules used

- module AND2 (A, B, Z);

```
 input A,B;
 output Z;
 // Z = A & B;
endmodule
```

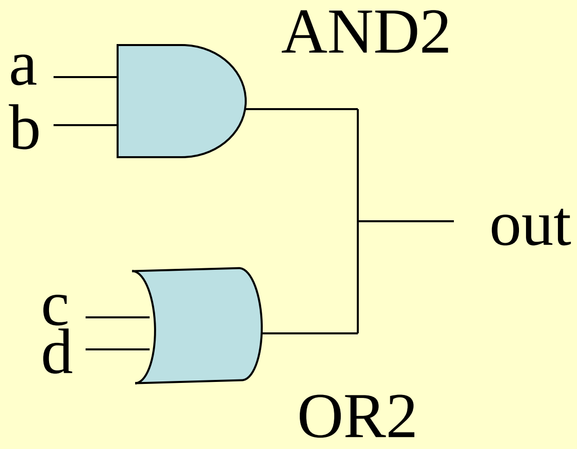
- module OR2 (A, B, Z);

```
 input A,B;
 output Z;
 // Z = A | B;
 endmodule
```

Data Types

- Intermediate Translation into Generic Library modules

```
module wireexample (out, a, b, c, d);
input a,b,c,d;
output out;
AND2(a,b,out);
OR2(a,b,out);
endmodule
```

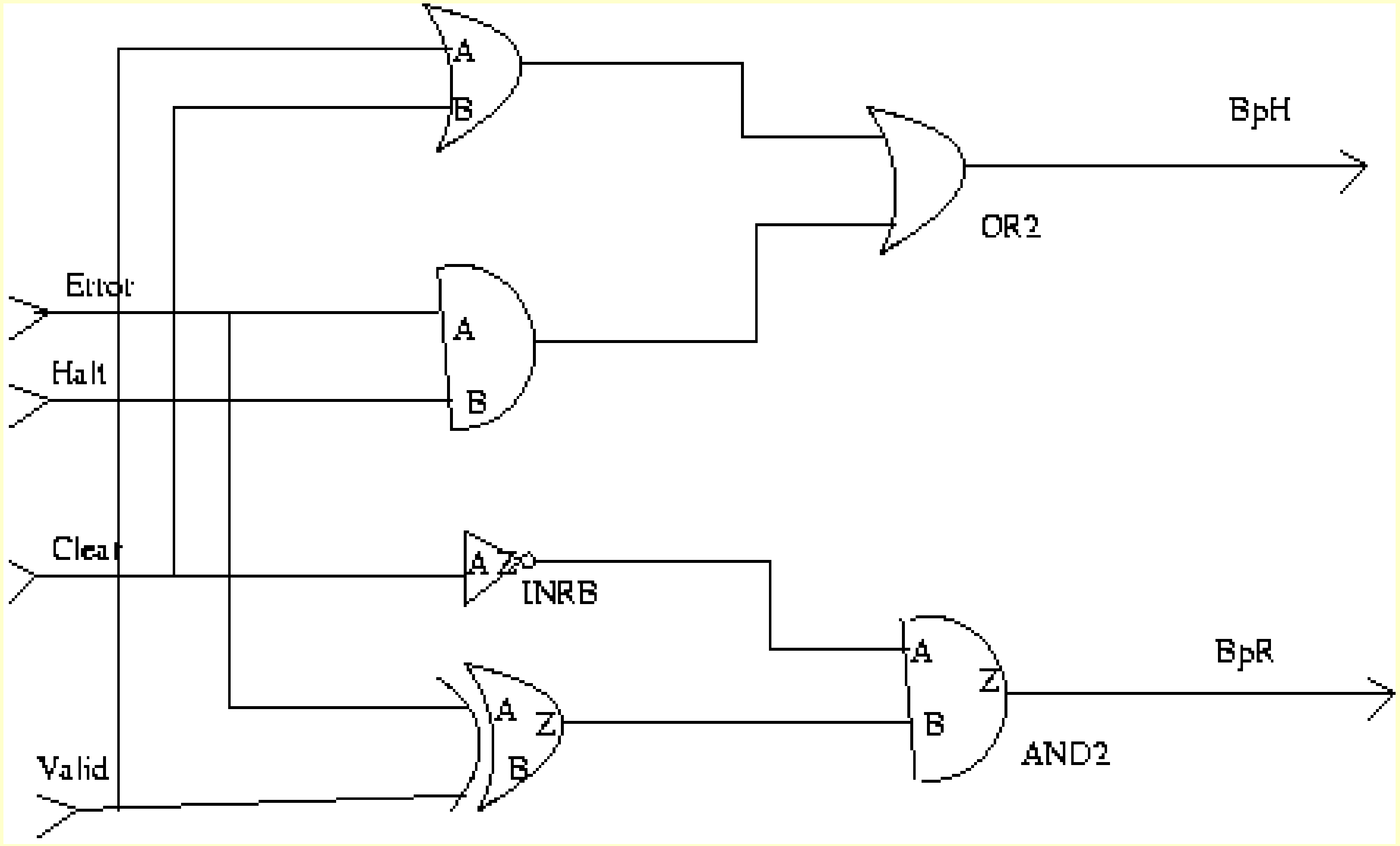


Example

- module UsesGates (BpW, BpR, Error, Wait, Clear);

```
input Error, Wait, Clear;
output BpW, BpR;
wor BpW;
wand BpR;
        assign BpW = Error & Wait;
        assign BpW = Valid | Clear;
        assign BpR = Error ^ Valid;
        assign BpR = ! Clear;
endmodule
```

Synthesized netlist



Constants

- 30 Signed, 32 bits
- -2 Signed, 32 bits
- 2’b10 Unsigned, 2 bits
- -6’d4 Unsigned, 6 bits, 2’s complement of 4
- -’d10 Unsigned, 32 bits, 2’s complement of 10

Synthesis of Parameters

- Parameter is a named constant
- Its size is the same as the size of the constant.
- parameter RED = -1; //32 bit, signed
- parameter READY = 2'b01;//2 bit, unsigned

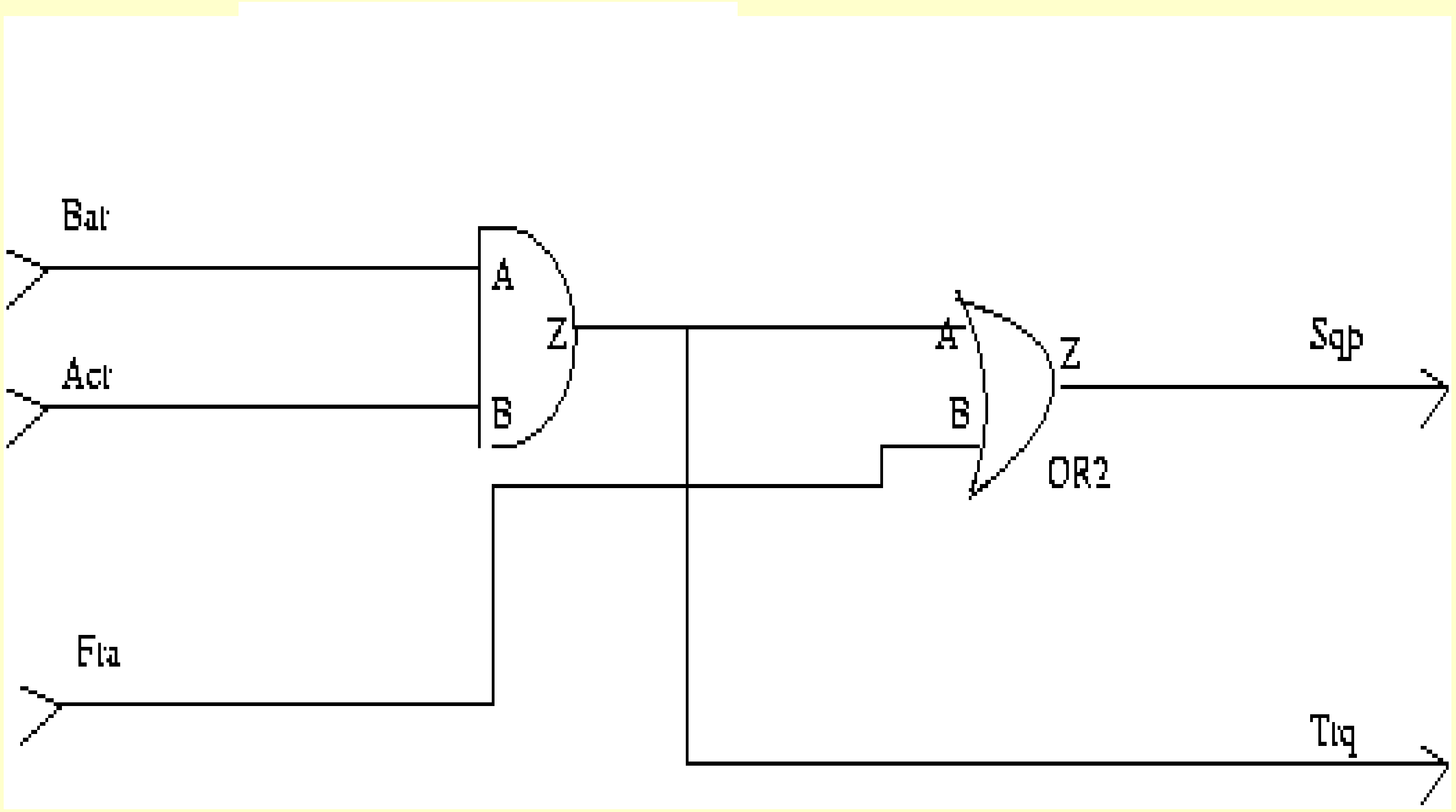
Value Holders

- Value holders in Hardware:
  - Wire
  - Flip-flop
  - Latch
- Variable of net type maps onto wires
- Variable of reg type maps onto wires or registers (collection of flip-flops or latches).

Synthesis of ‘reg’ variables

```
wire Acr, Bar, Fra;
reg Trq, Sqp;
always @(Bar or Acr or Fra)
begin
    Trq = Bar & Acr;
    Sqp = Trq | Fra;
end
```

- Now, the ‘reg’ variables DO NOT synthesize as sequential memory elements.


Bat
Act
Fla
A
B
Z
A
B
Z
OR2
Sqp
Tlq

Synthesis of ‘reg’ variables

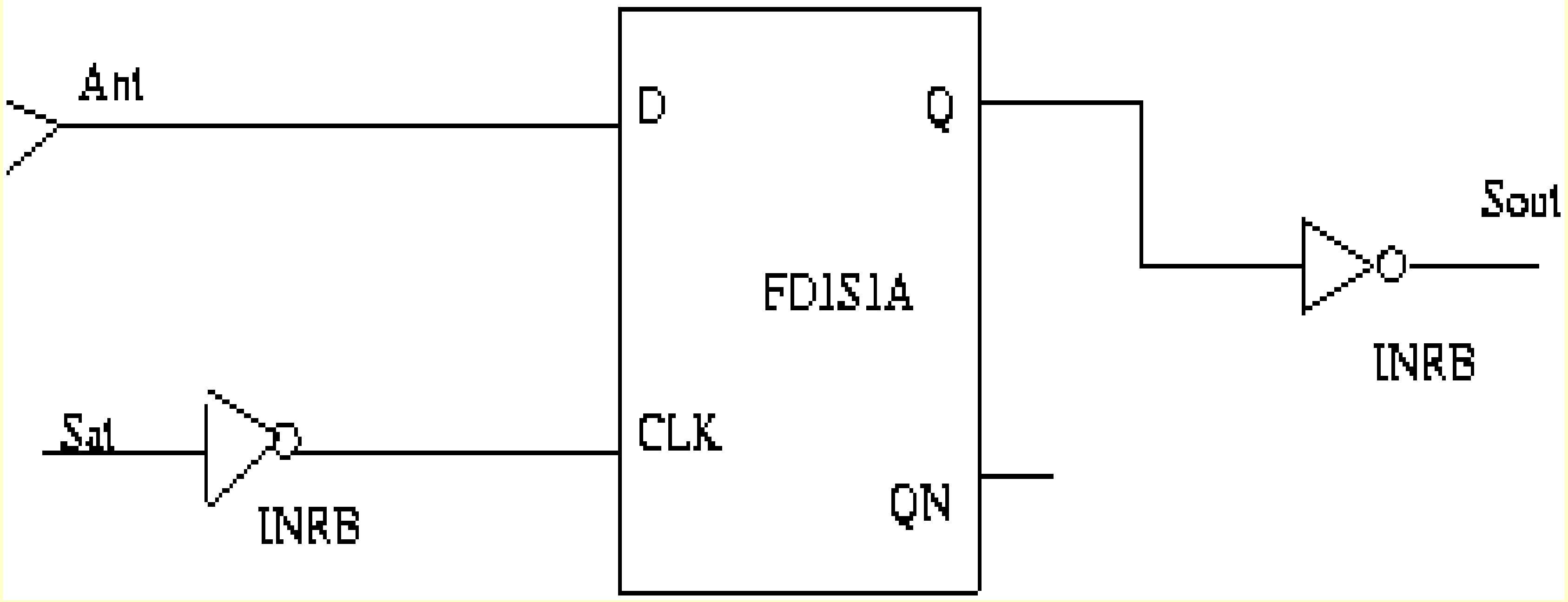
```
wire Acr, Bar, Fra;
reg Trq, Sgp;
always @(Bar or Acr or Fra)
begin
    Sqp = Trq | Fra;
    Trq = Bar & Acr;
end
```

- Now, the Trq ‘reg’ variable is used before it is assigned and hence it should be a memory element.

Synthesis of ‘reg’ variables

```
 wire Sat, Ant;
  reg Fox,  Sout;
  always @ (Sat or Ant)
  begin
      if (! Sat)
               Fox = Ant;
       Sout = !Fox;
  end
```

- The variable Fox is not assigned in the ‘else’ branch and hence it is inferred as a latch. Fox should retain its old value when Sat is true.


Ant
D
Q
Sout
FD1S1A
INRB
Sat
CLK
QN
INRB

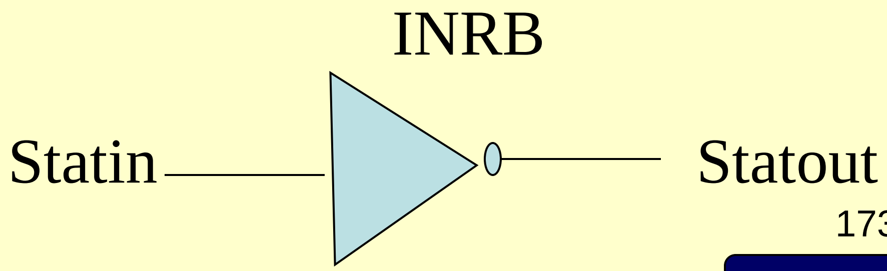
Flip-flop Vs Latch

- Understand why Fox is inferred as a latch and not a flip-flop.
- This depends on the context under which a variable is assigned a value.

Verilog Constructs to Gates

- Continuous Assignment
  - This represents in hardware, logic that is derived from the expression on the right-hand-side of the assignment statement driving the net that appears on the left-hand-side of the assignment statement.
- module Continuous (Statin, StatOut);

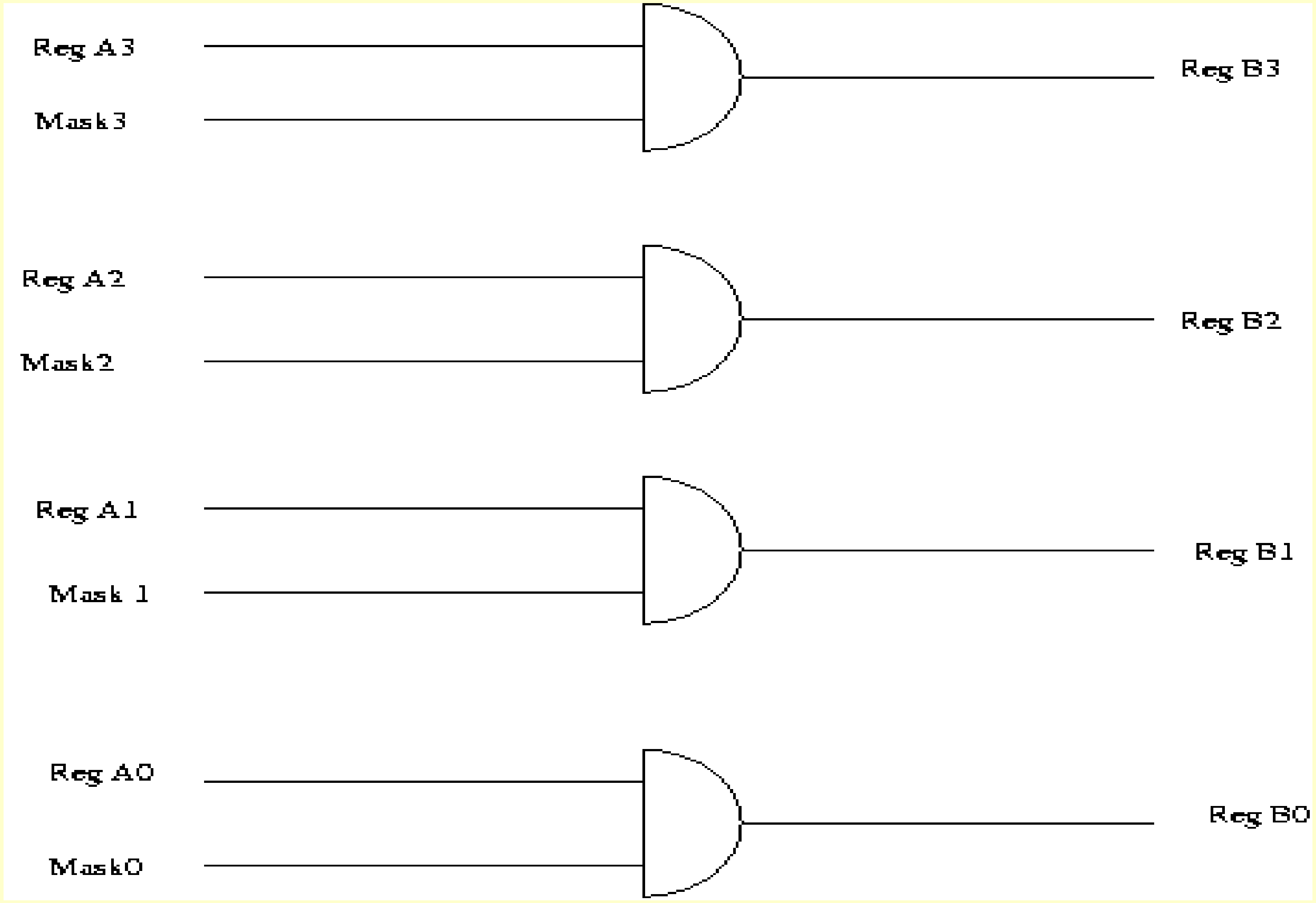
```
input Statin;
output Statout;
    assign Statout = ~Statin;
endmodule
```



Non-Blocking Procedural Assignment

```
• module NonBlocking (RegA, Mask, RegB);
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;
  always @(RegA or Mask)
          RegB <= RegA & Mask;
   endmodule
```

Synthesized Circuit

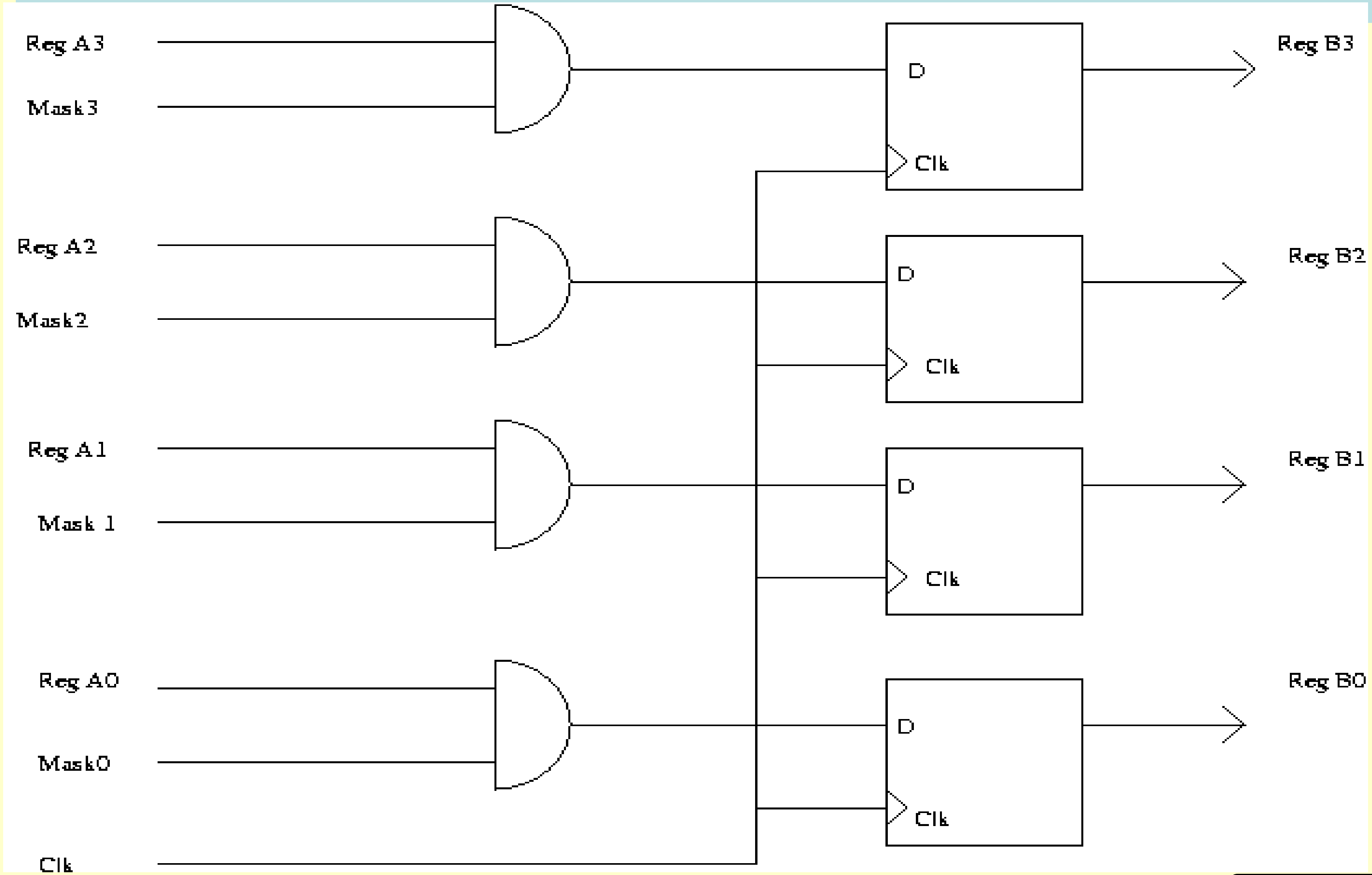


Target of Assignment

- Target of a procedural assignment is synthesized to a wire, a flip-flop, or a latch, depending on the context.
- If the assignment is under the control of an edge of a clock, it infers a flip-flop.
-

```
module Target(Clk, RegA, RegB, Mask);
input Clk;
input [3:0] RegA, Mask;
output [3:0] RegB;
reg [3:0] RegB;
always @(posedge Clk)
        RegB <= RegA & Mask;
endmodule
```

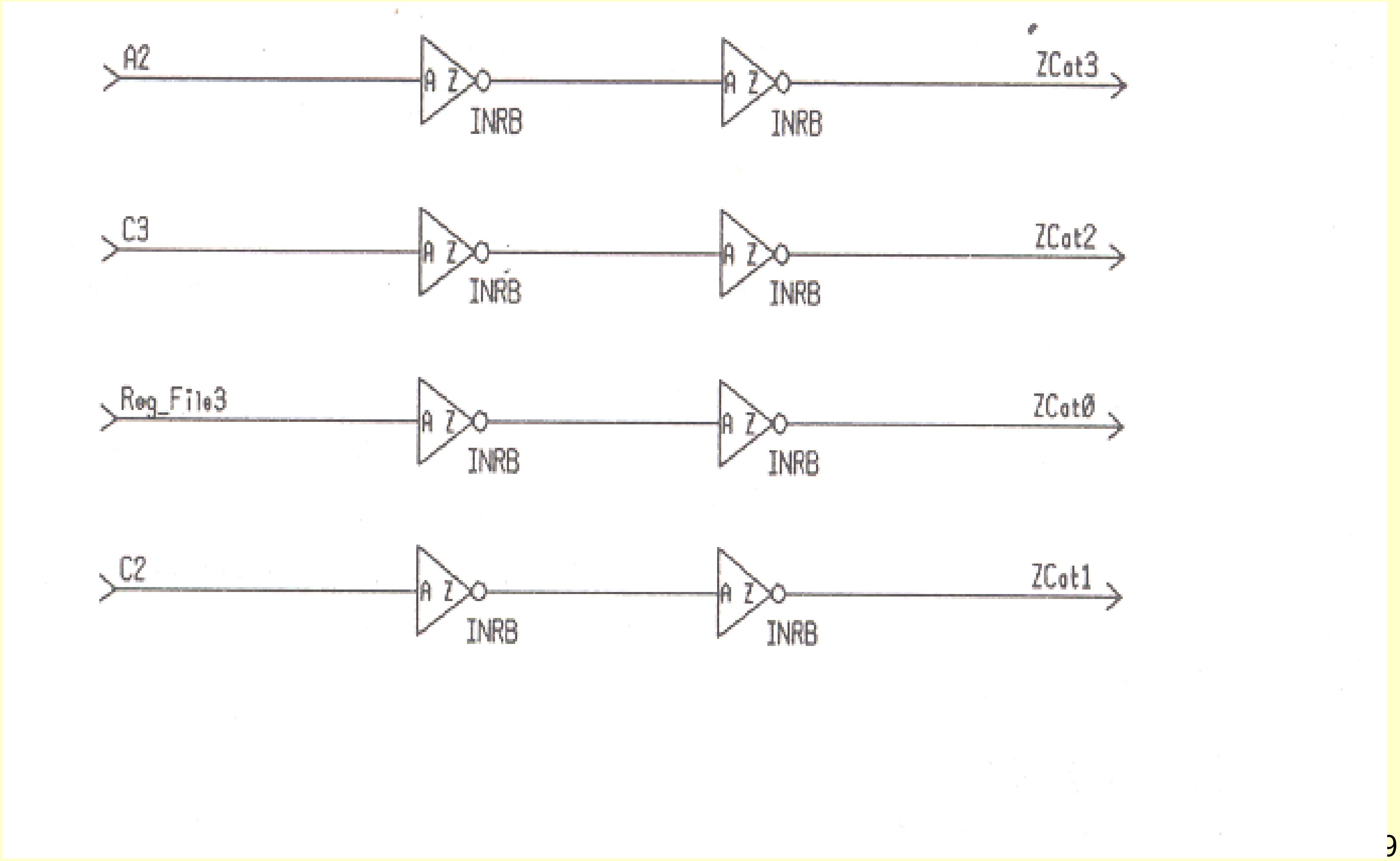
Synthesized Circuit



RISE Group

Constant Index

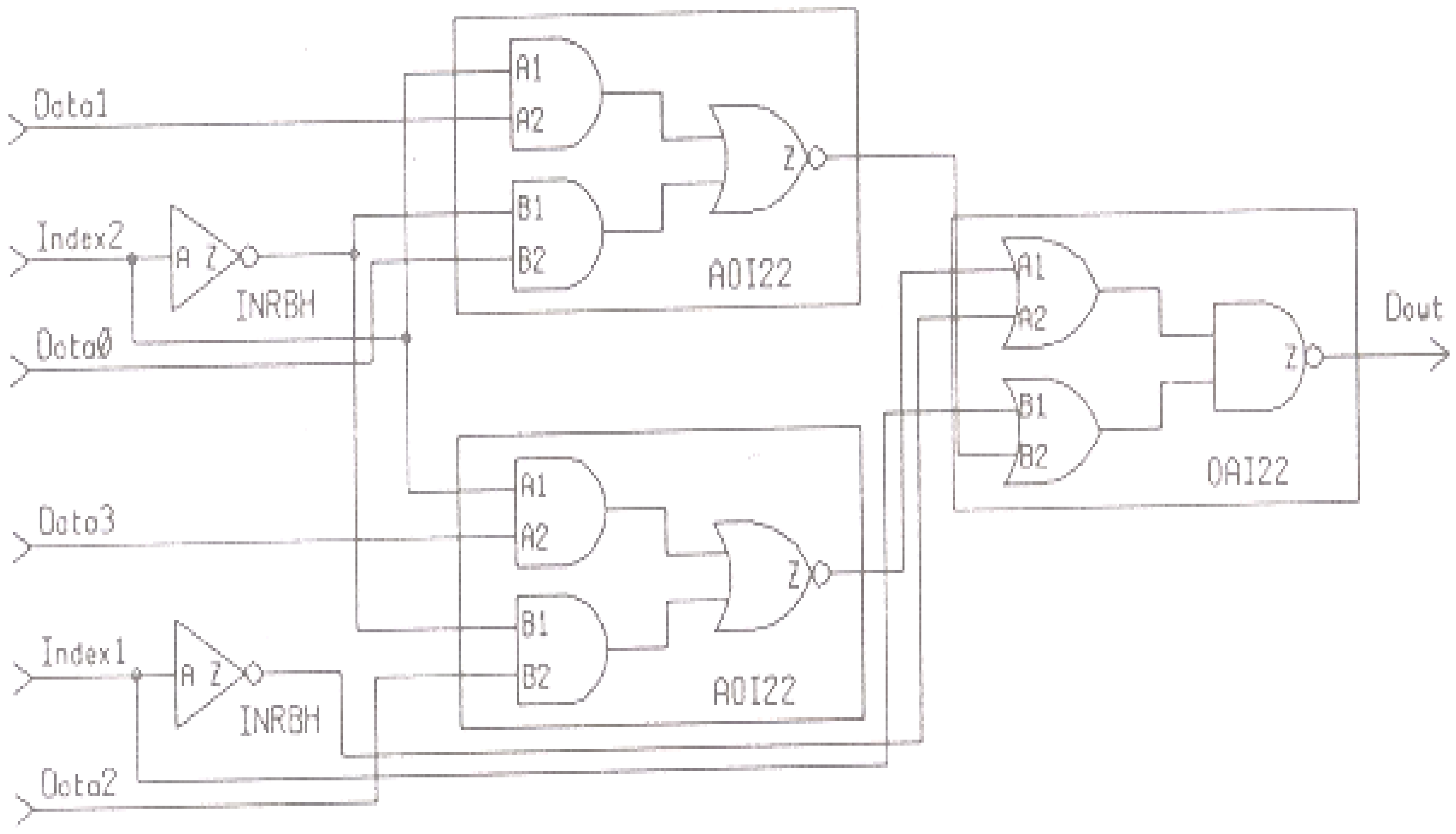
```
• module ConstantIndex (A, C, Reg_File, Zcat);
  input [3:0] A,C;
  input [3:0] Reg_File;
  output [3:0] Zcat;
          assign Zcat[3:1] = {A[2],C[3:2]};
          assign Zcat[0] = Reg_File[3];
  endmodule
```


A2
A Z
INRB
A Z
INRB
ZCat3
C3
A Z
INRB
A Z
INRB
ZCat2
Reg_File3
A Z
INRB
A Z
INRB
ZCatØ
C2
A Z
INRB
A Z
INRB
ZCat1

Non-Constant Index in Expression

```
module NonComputeRight (Data, Index, Dout);
  input [0:3] Data;
  input [1:2] Index;
  output Dout;
      assign Dout = Data[Index];
endmodule
```

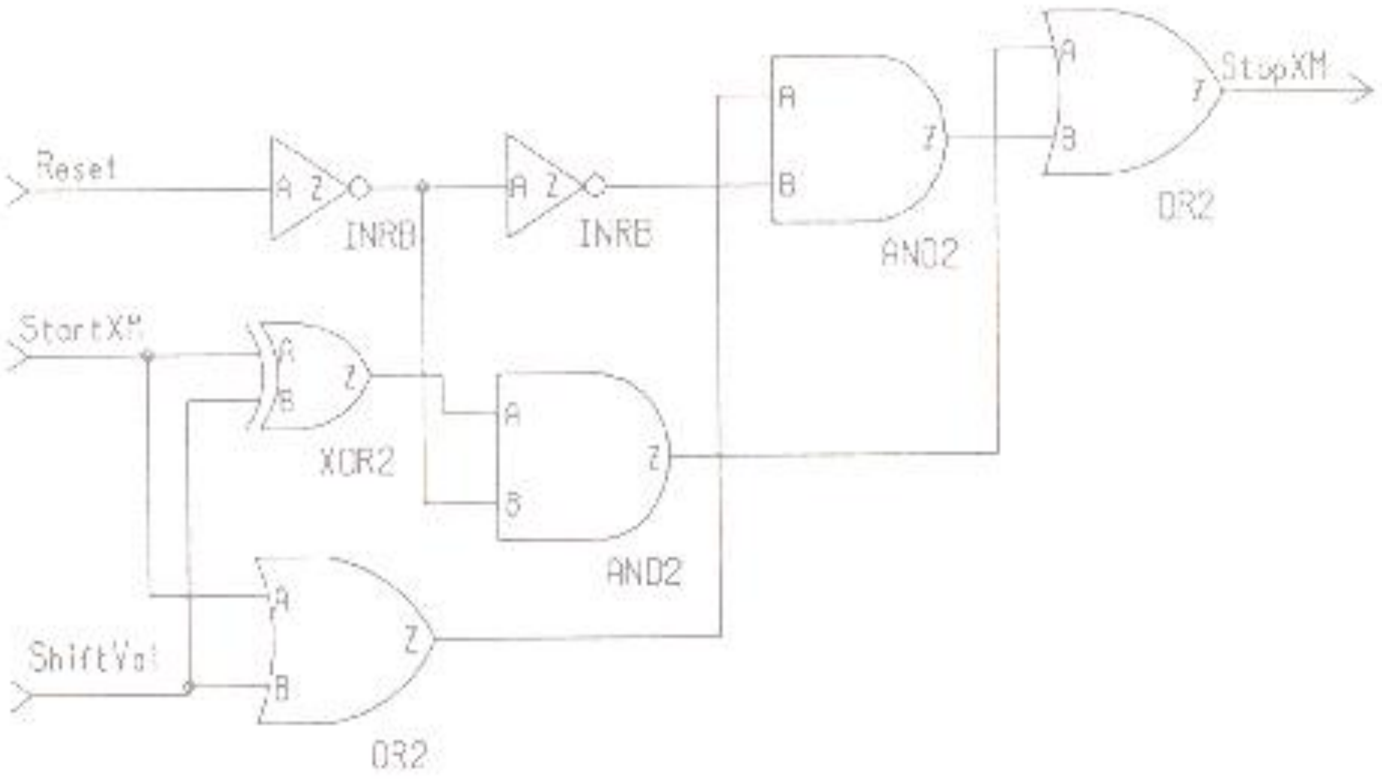
Synthesized circuit



Conditional Expression

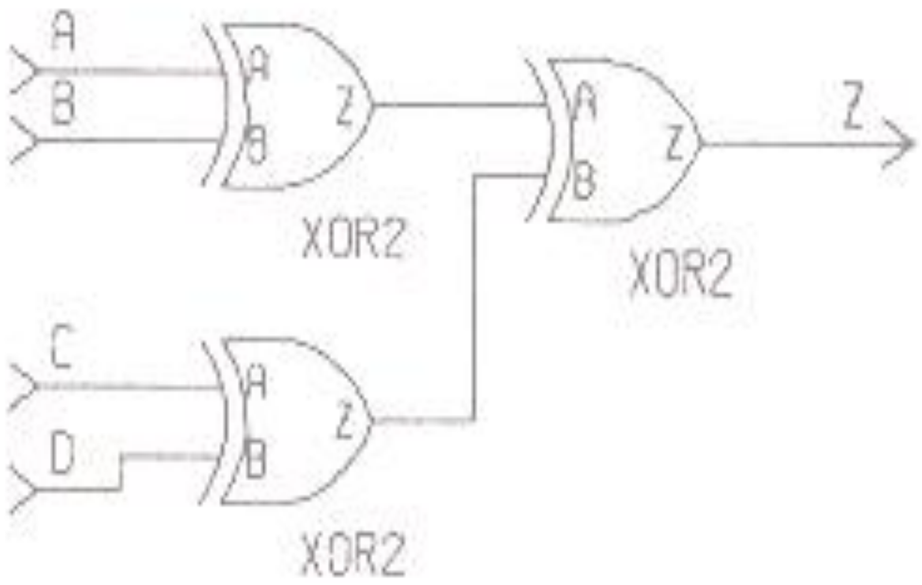
```
module ConditionalExpression (StartXM, ShiftVal, Reset, StopXM);
input StartXM, ShiftVal, Reset;
output StopXM;
    assign StopXM = (! Reset)? StartXM ^ ShiftVal: StartXM | ShiftVal;
endmodule
```

- A 2-to-1 Multiplexer, with Conditional expression as control, first expression as ‘1’ option and second expression as ‘0’ option


Reset
StartXM
ShiftVal
StopXM
INRB
INRB
XOR2
AND2
AND2
OR2
OR2
A
B
Z

Always Statement

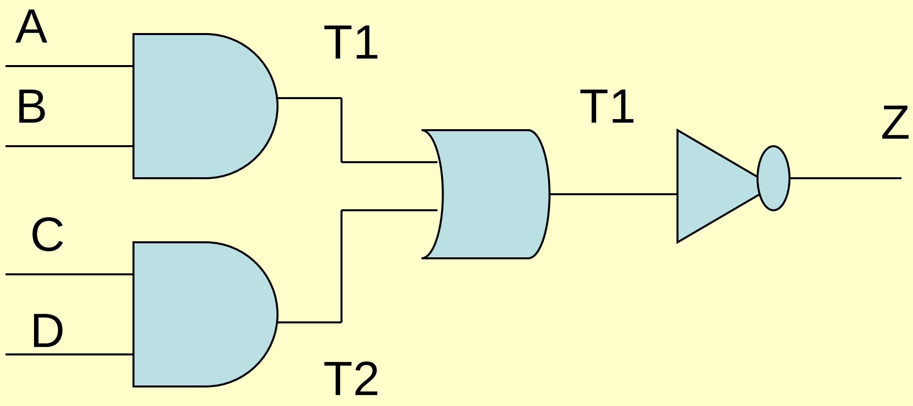
```
module EvenParity (A, B, C, D, Z);
input A, B, C, D;
output Z;
reg Z, Temp1, Temp2;
        always @(A  or B or C or D)
        begin
                Temp1 = A ^ B;
                Temp2 = C ^ D;
                Z = Temp1 ^ Temp2;
        end
endmodule
```


A
B
C
D
A
B
Z
XOR2
A
B
Z
XOR2
A
B
Z
XOR2
Z

Temporary Variables

- Each assignment (even to same variable) implies a unique wire

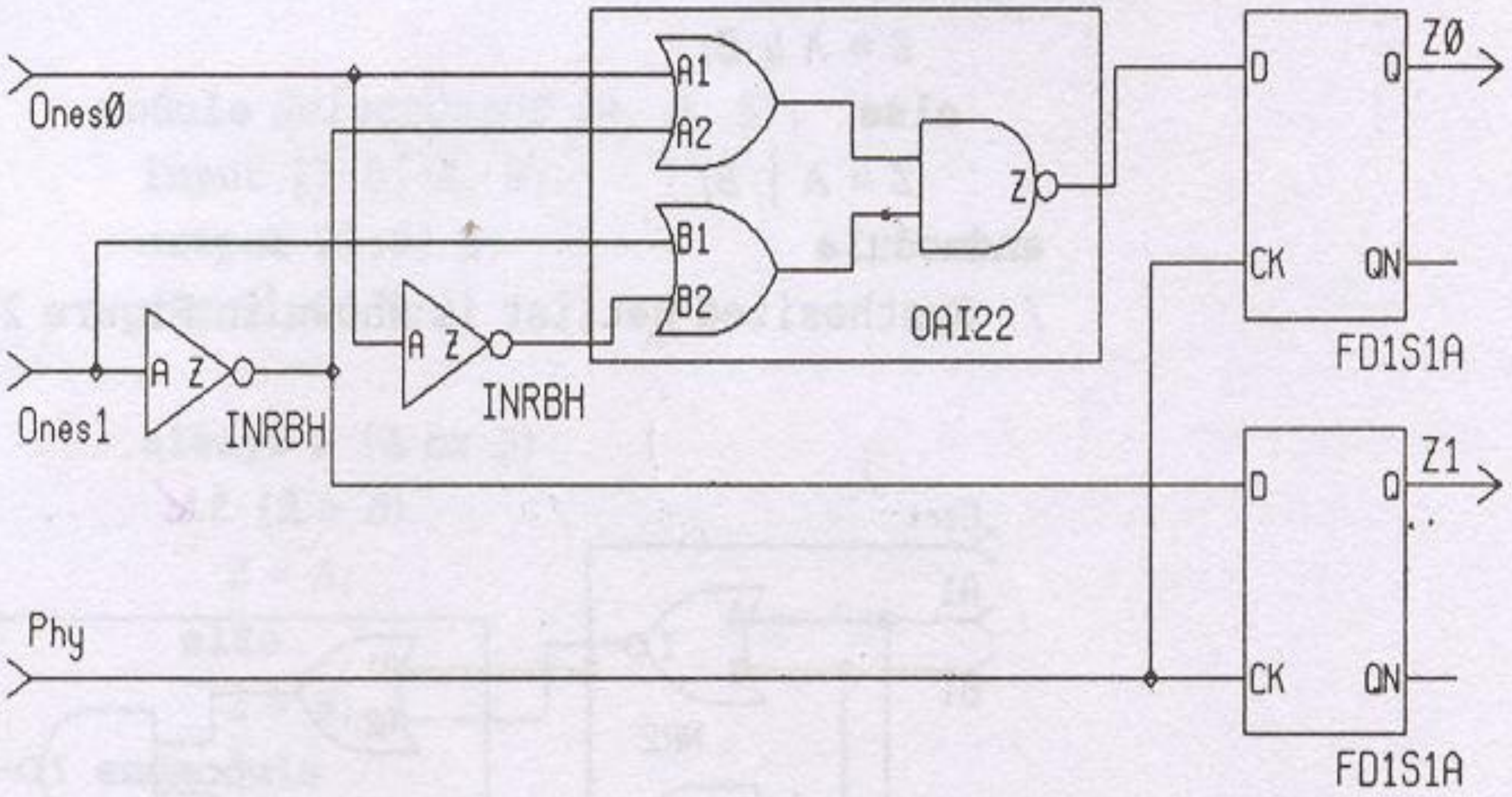
```
 module VariablesAreTemporaries (A, B, C, D, Z);
  input A, B, C, D;
  output Z;
  reg Z;
        always @(A or B or C or D)
        begin: VAR_LABEL
                integer T1, T2;
                T1 = A & B;
                T2 = C & D;
                T1 = T1 | T2;
                Z = ~T1;
     end
  endmodule
```



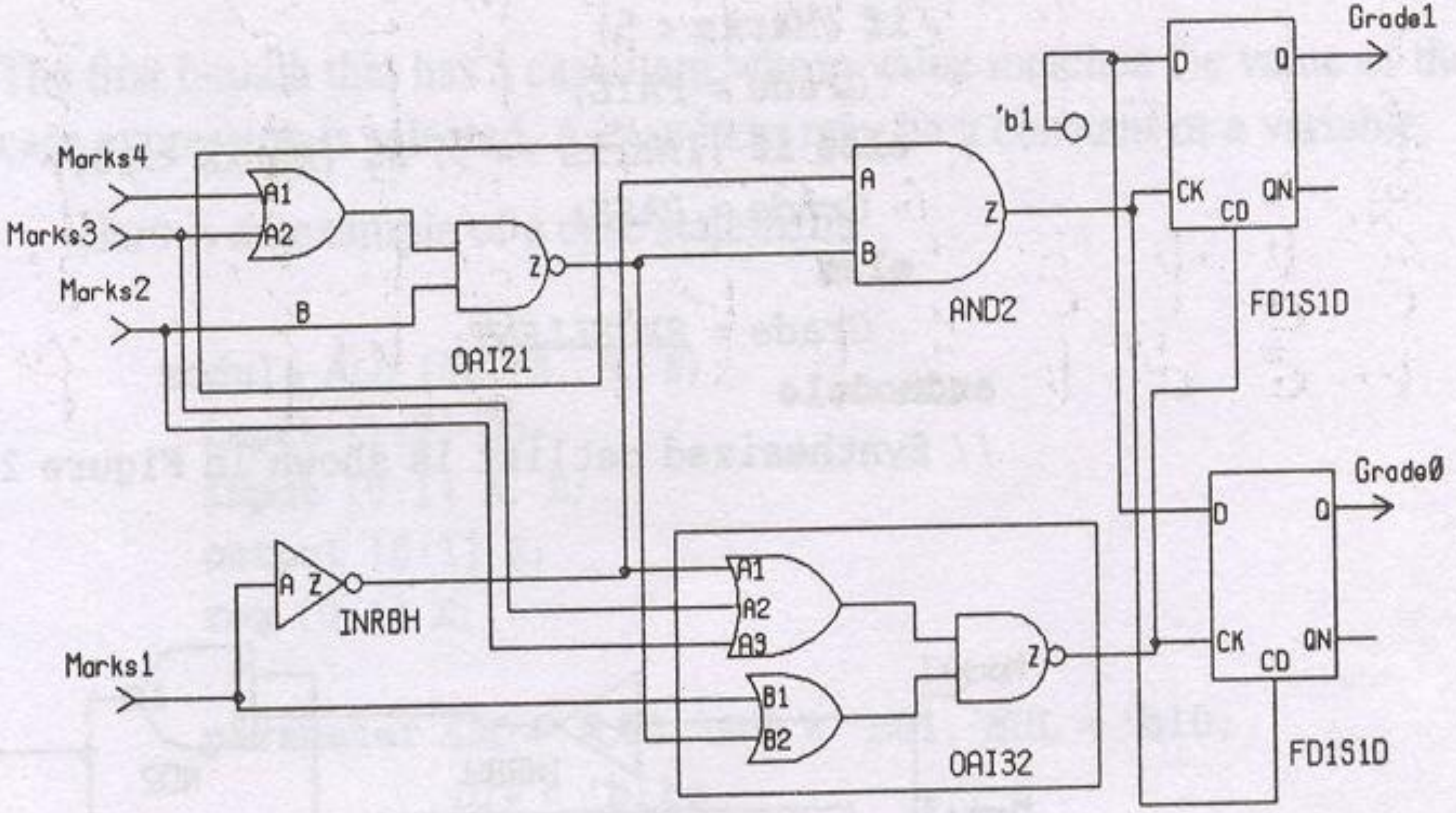
If statement and Latches

```
module Increment(Phy, Ones, Z);
input Phy;
input [0:1] Ones;
output [0:1] Z;
reg [0:1] Z;
        always @(Phy or Ones)
                if (Phy) Z = Ones + 1;
endmodule     // latch inferred
```

Latch Inferred



```
 module Compute (Marks, Grade);
input [1:4] Marks;
output [0:1] Grade;
reg [0:1] Grade;
parameter FAIL = 1; PASS =2; EXCELLENT = 3;
   always @(Marks)
         if (Marks < 5) Grade = FAIL;
                  else if ((Marks >= 5) & (Marks < 10))
                           Grade = PASS;
endmodule   // Again a Latch, what if Marks < 16?
```

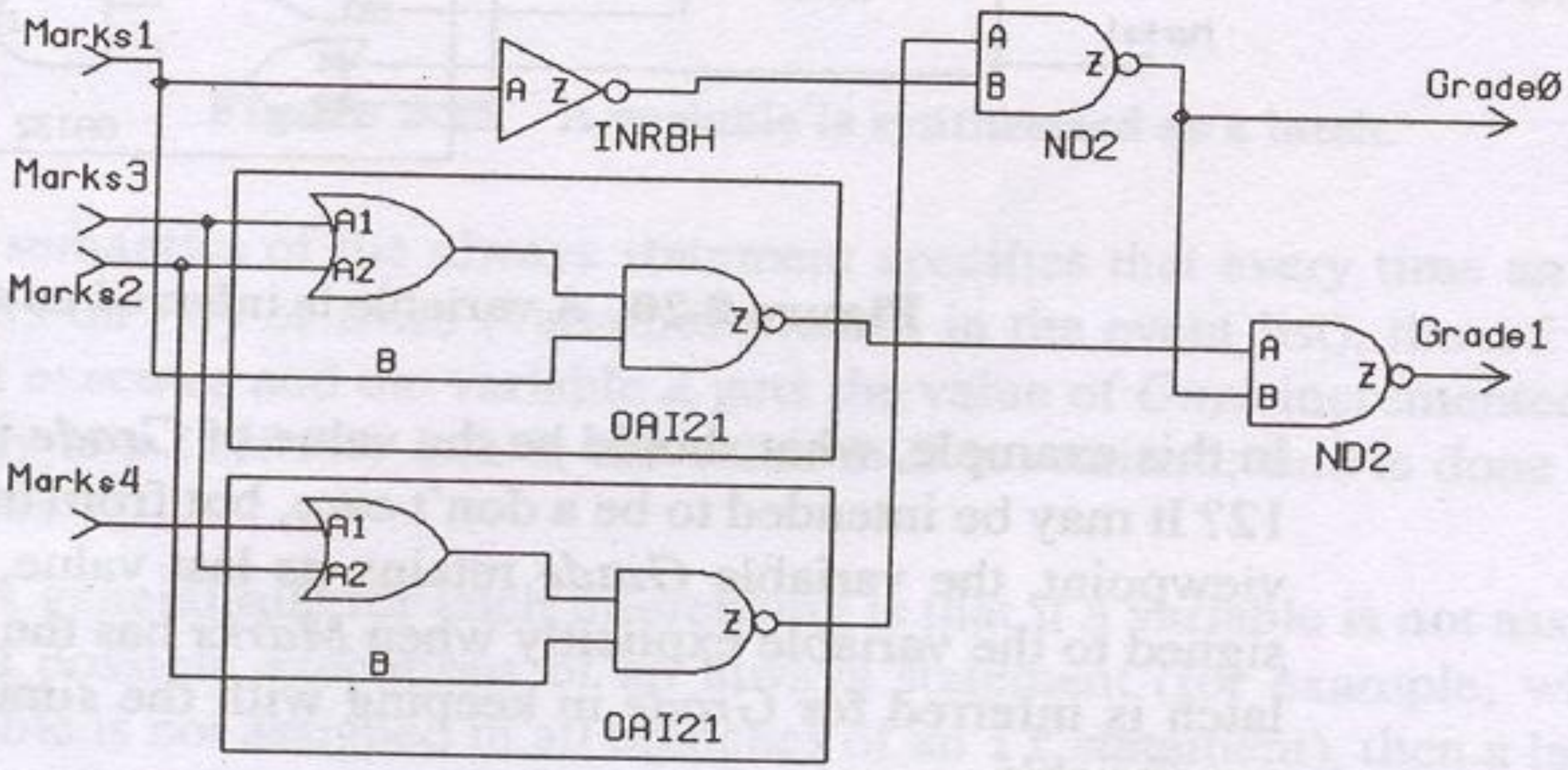


Grade1
D
Q
'b1
CK
QN
CD
Marks4
A1
A2
Marks3
A
Z
B
AND2
Z
FD1S1D
Marks2
B
OAI21
Grade0
D
Q
A1
A Z
A2
INRBH
A3
Z
CK
QN
Marks1
CD
B1
B2
OAI32
FD1S1D

If statement and Latches

```
module Compute (Marks, Grade);
input [1:4] Marks;
output [0:1] Grade;
reg [0:1] Grade;
parameter FAIL = 1; PASS =2; EXCELLENT = 3;
  always @(Marks)
        if (Marks < 5) Grade = FAIL;
              else if ((Marks >= 5) & (Marks < 10))
                    Grade = PASS;
              else Grade = EXCELLENT;
endmodule
```

No latches

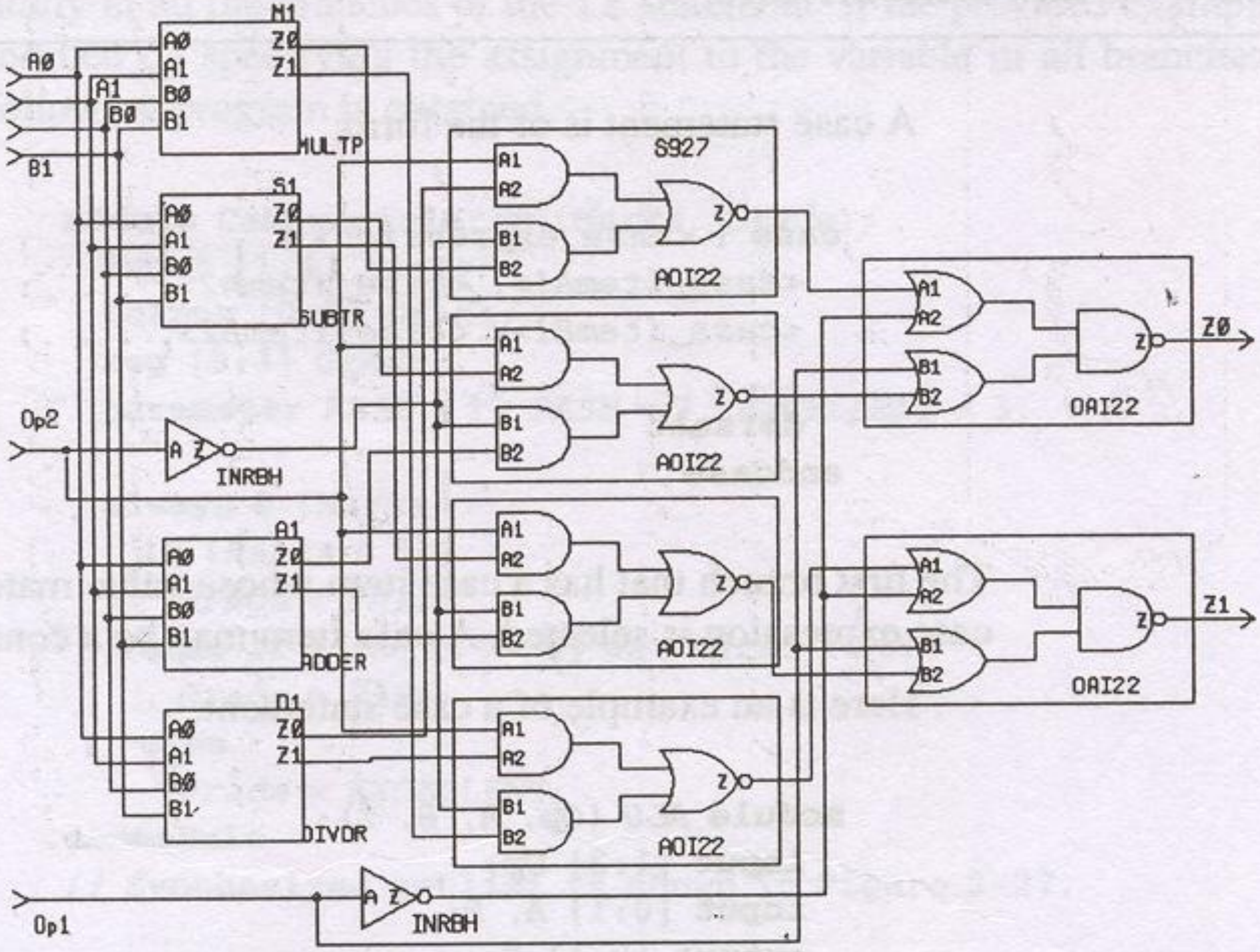

Marks1
Marks3
Marks2
Marks4
A
Z
INRBH
A
B
Z
ND2
Grade0
A1
A2
Z
B
OAI21
A1
A2
Z
B
OAI21
Grade1
A
B
Z
ND2

Case Statement

```
module ALU (Op, A, B, Z);
input [1:2] Op;
input [0:1] A, B;
output [0:1] Z;
reg [0:1] Z;

parameter ADD = ’b00, SUB = ’b01, MUL = ‘b10, DIV = ‘b11;

        always @(Op or A or B)
        case (Op)
                ADD: Z = A + B;
                SUB: Z = A – B;
                MUL: Z = A * B;
                DIV: Z = A/B;
        endcase
endmodule
```


M1
MULTP
S1
SUBTR
A1
ADDER
D1
DIVDR
A0
A1
B0
B1
Z0
Z1
Op2
Op1
INRBH
S927
AOI22
OAI22
Z0
Z1

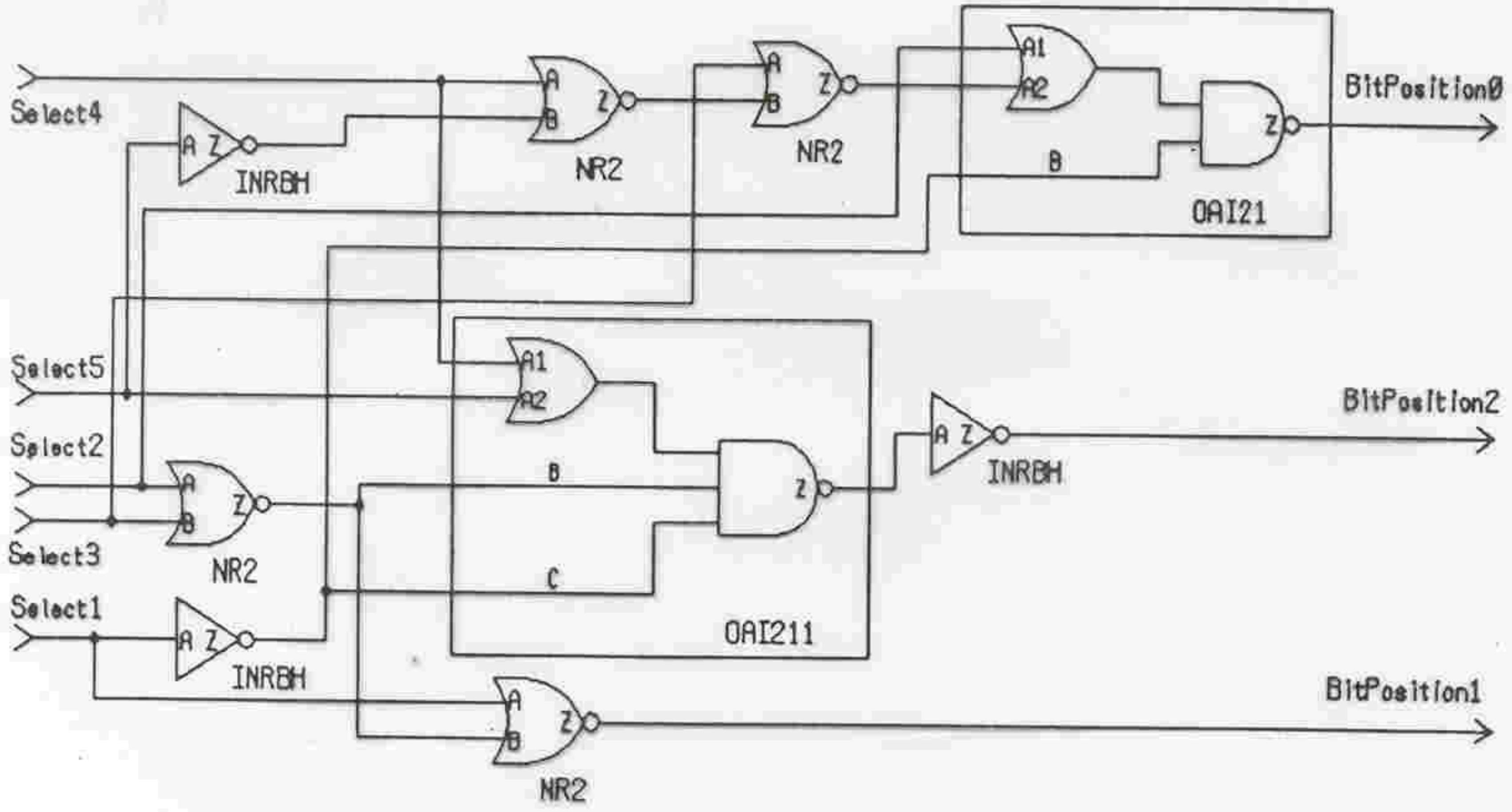
Synthesis of casex

```
 module PriorityEncoder (Select, BitPosition);
input [5:1] Select;
output [2:0] BitPosition;
reg [2:0] BitPosition;
   always @(Select)
        casex (Select)
                 5’bxxxx1 : BitPosition = 1;
                 5’bxxx1x : BitPosition = 2;
                 5’bxx1xx : BitPosition = 3;
                 5’bx1xxx : BitPosition = 4;
                 5’b1xxxx : BitPosition = 5;
                 default : BitPosition = 0;
        endcase
endmodule
```

The Semantics

-
  ```
  if (Select[1]) BitPosition = 1; else
  if (Select[2]) BitPosition =  2; else
  if (Select[3]) BitPosition = 3; else
  if (Select[4]) BitPosition = 4; else
  if (Select[5]) BitPosition = 5; else
  BitPosition = 0;
  ```

The Synthesized Netlist



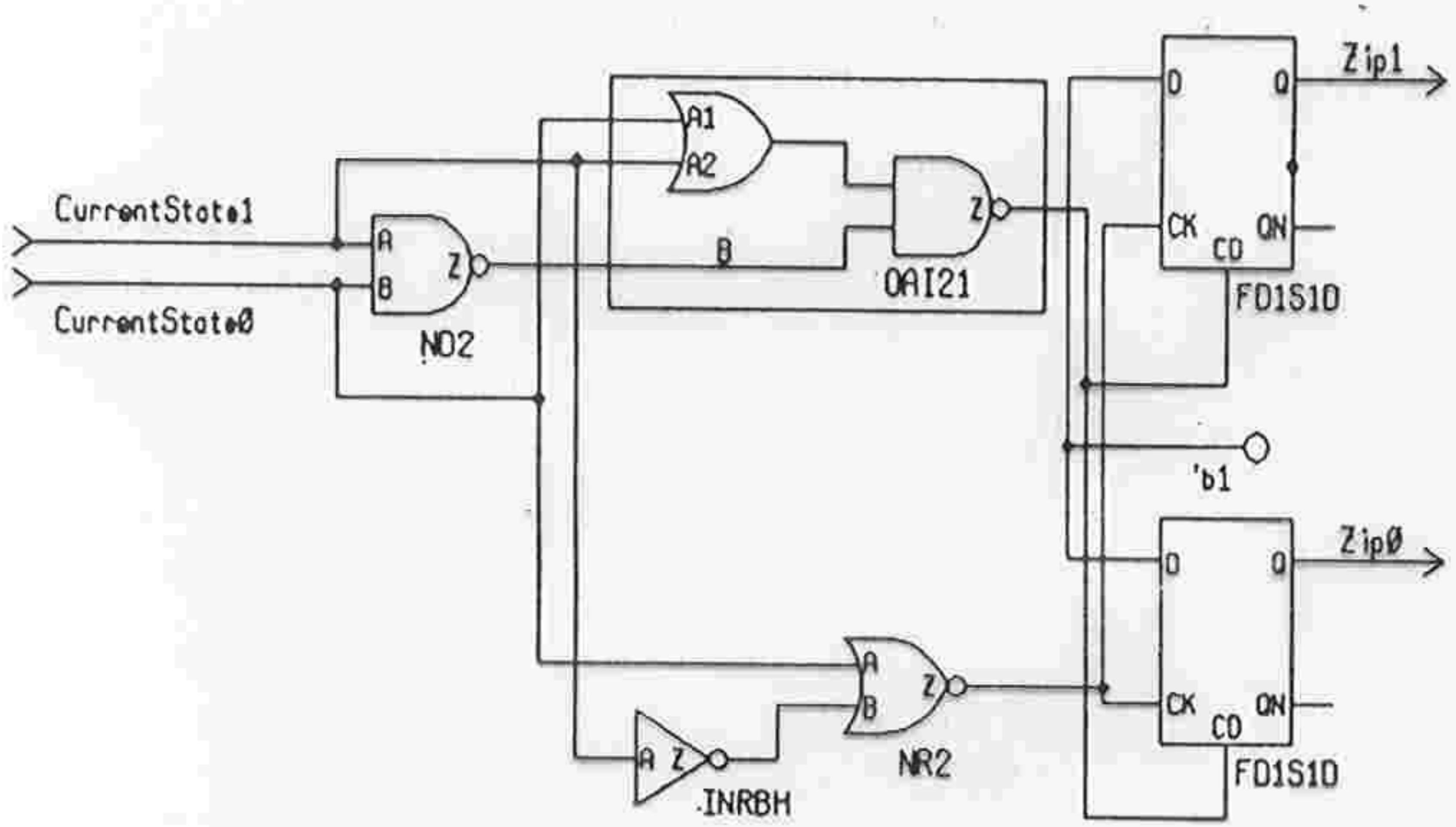
A priority encoder using casex statement.

Inferring Latches in case statements

- If a variable is assigned a value only in some branches of a ‘case’ statement, and not in all possible branches, then, a latch is inferred for that variable.
- The rules apply equally for casex and casez statements

Example

```
module StateUpdate (CurrentState, Zip);
 input [0:1] CurrentState;
 output [0:1] Zip;
 reg [0:1] Zip;
 parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
 always @(CurrentState)
   case (CurrentState)
   S0, S3: Zip = 0;
   S1: Zip = 3;
   endcase
  endmodule
```



Latch inferred for a variable in a case statement.

To Avoid Latches

```
module StateUpdate (CurrentState, Zip);
 input [0:1] CurrentState;
 output [0:1] Zip;
 reg [0:1] Zip;
 parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
 always @(CurrentState)
 begin
  Zip = 0;    //This statement is added – mimics 'default'
         case (CurrentState)
                   S0, S3: Zip = 0;
                   S1: Zip = 3;
         endcase
 end
  endmodule
```

Synthesis of For Loops

- Unrolling the For loop – All statements within the For loop are replicated, once for each value of the For-loop index.
- The restriction is that the loop bounds have to be constants – else synthesis is beyond any scope.

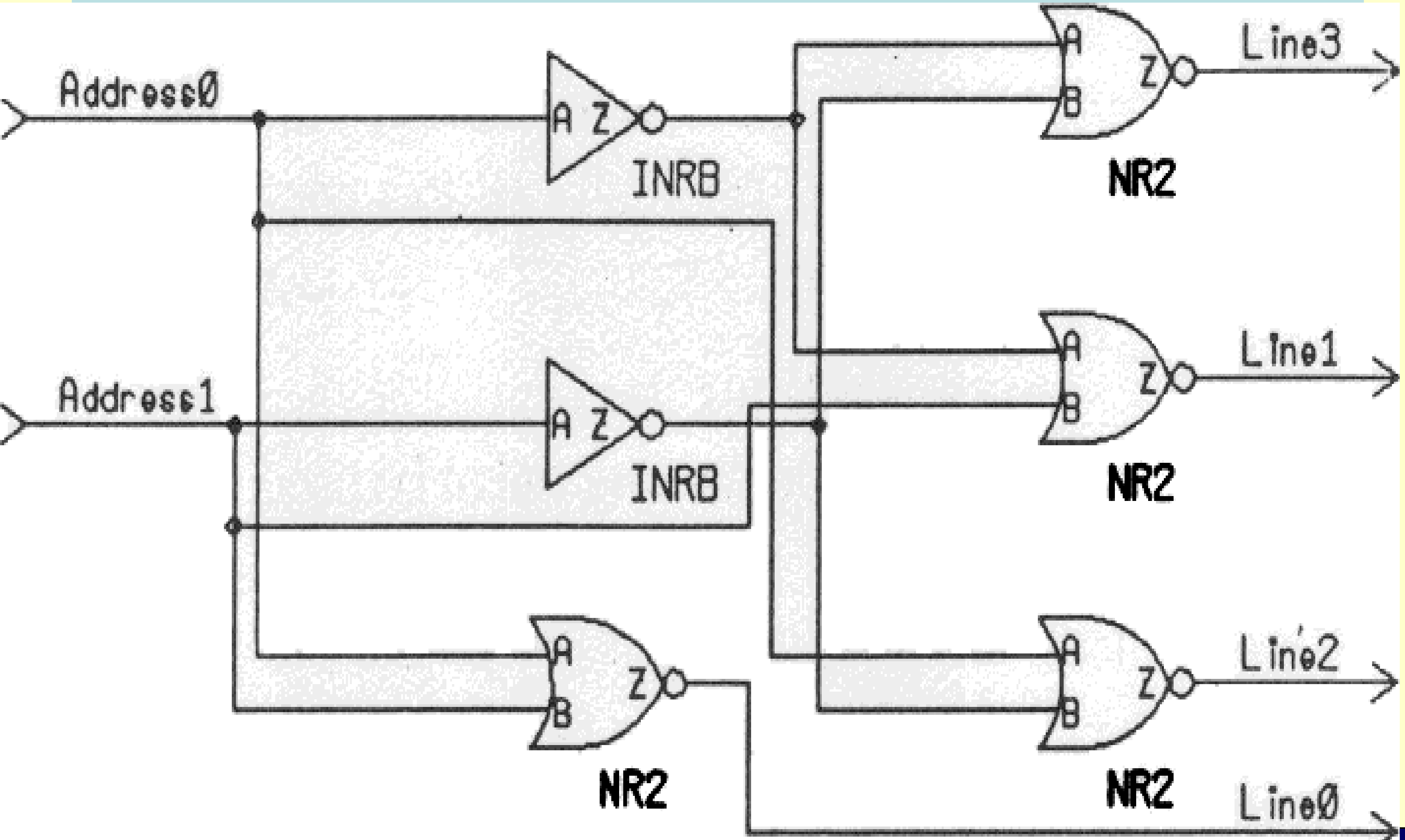
Example

```
•  module Demultiplexer(Address, Line);
•  input [1:0] Address;
•  output [3:0] Line;
•  reg [3:0] Line;
•  integer J;
•   always @(Address)
•    for (J = 3; J >= 0; J = J –1)
•      if (Address == J) Line[J] = 1;
•      else Line[J] = 0;
•  endmodule
```

Loop Unrolling

```
 if (Address == 3) Line[3] = 1; else Line[3] = 0;
if (Address == 2) Line[2] = 1; else Line[2] = 0;
if (Address == 1) Line[1] = 1; else Line[1] = 0;
if (Address == 0) Line[0] = 1; else Line[0] = 0;
```

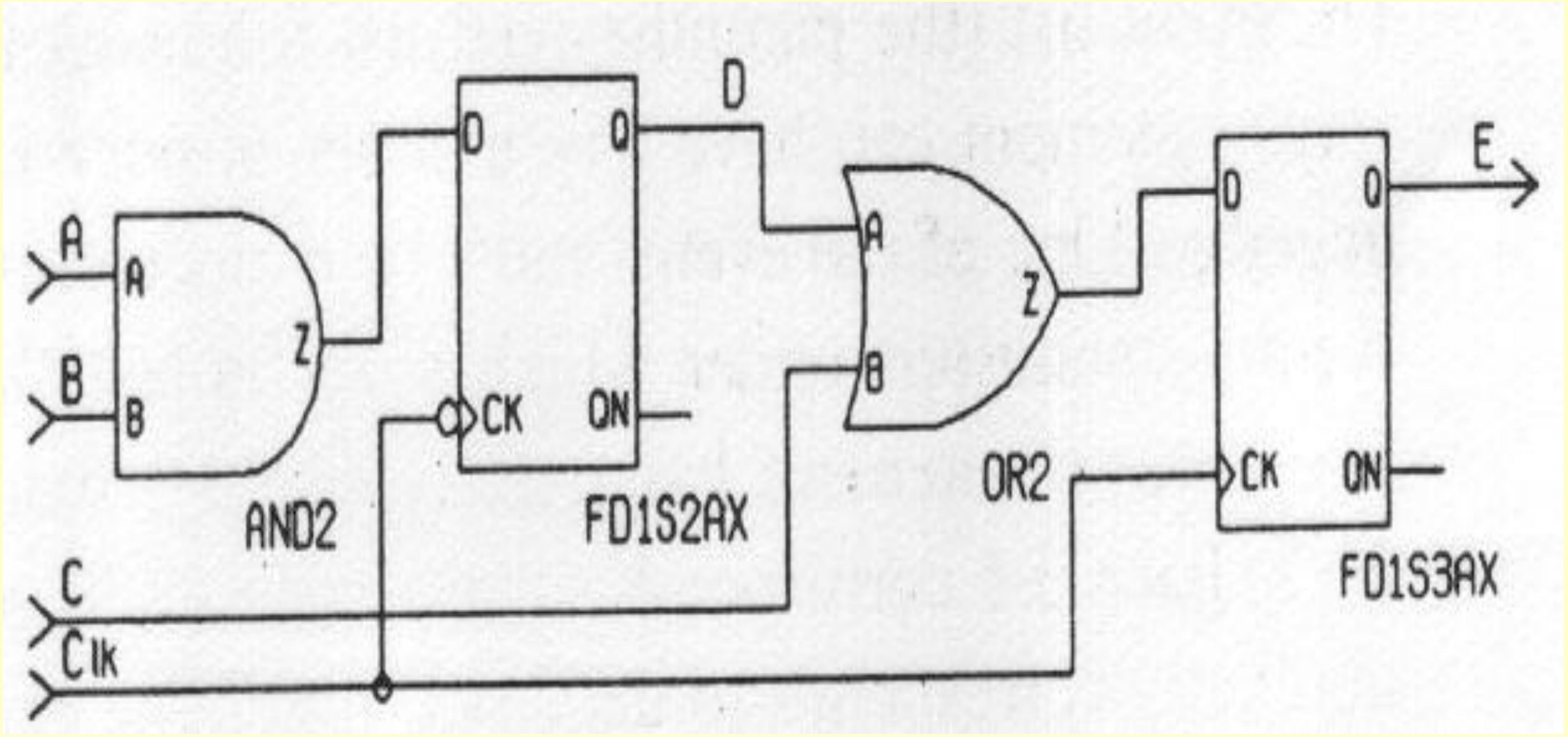
The Synthesized Netlist



Multi-phase Clocks

```
module MultiPhaseClocks (Clk, A,B,C,E);
  input Clk, A, B, C;
 output E;
 reg E,D;
 always @(posedge Clk)
   E <= D | C;
 always @(negedge Clk)
   D <= A & B;
 endmodule
```

The Synthesized Circuit



References

- Verilog HDL (2nd Edition)
  Samir Palnitkar, Prentice Hall Publications, 2003
  ISBN : 0130449113
- J. Bhasker's, "Verilog HDL Synthesis – A Practical Primer" – book.
- HDL Chip Design : A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog
  Douglas J Smith, Doone Publications, 1998
  ISBN : 0965193438

High Performance Circuits Design

High Performance Circuit Design

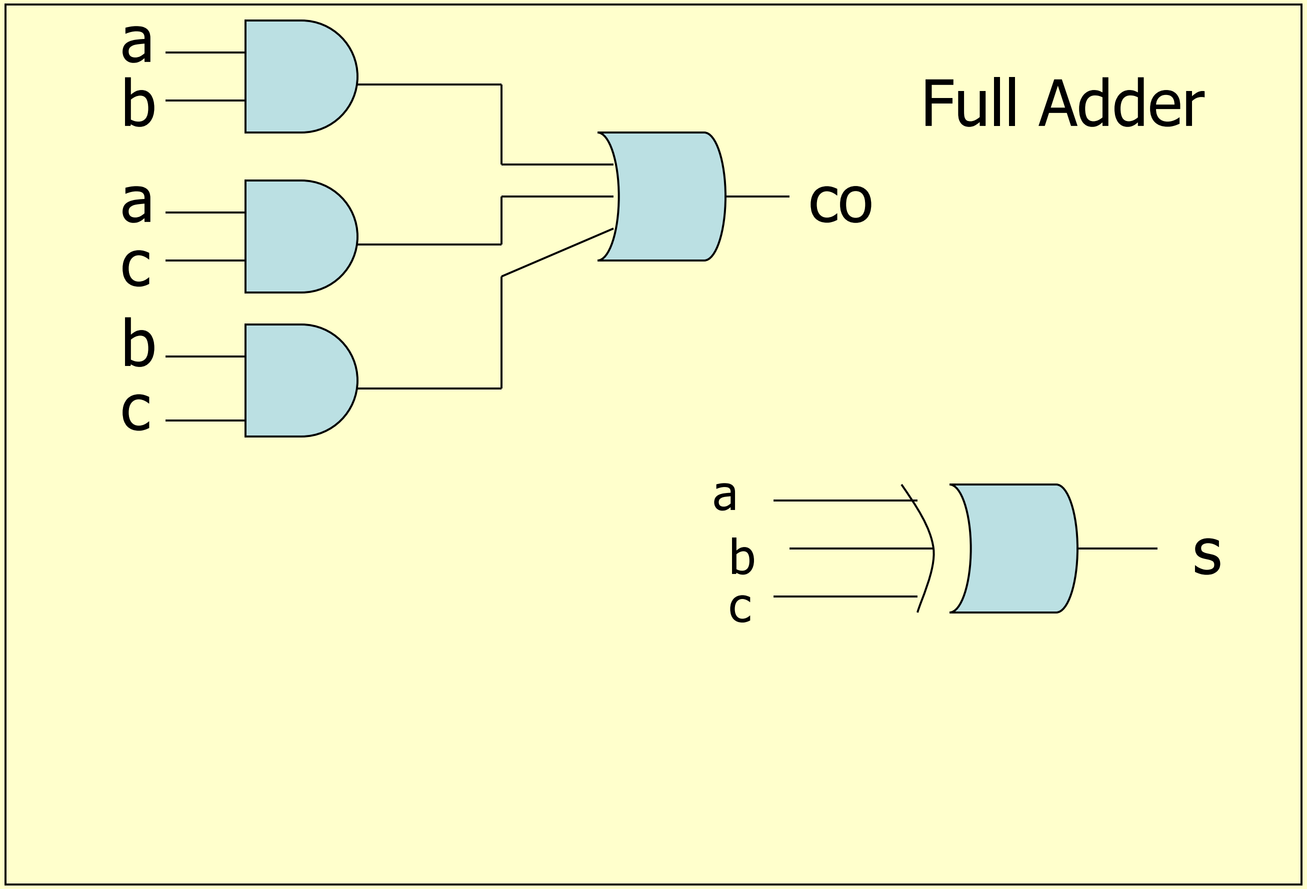
- High speed Adder Circuits
  - Carry Ripple – Inherently Sequential
  - Carry Look ahead – Parallel version
  - You should understand the conversion portion
- Multipliers
  - Wallace-tree multipliers

High Speed Circuit Design

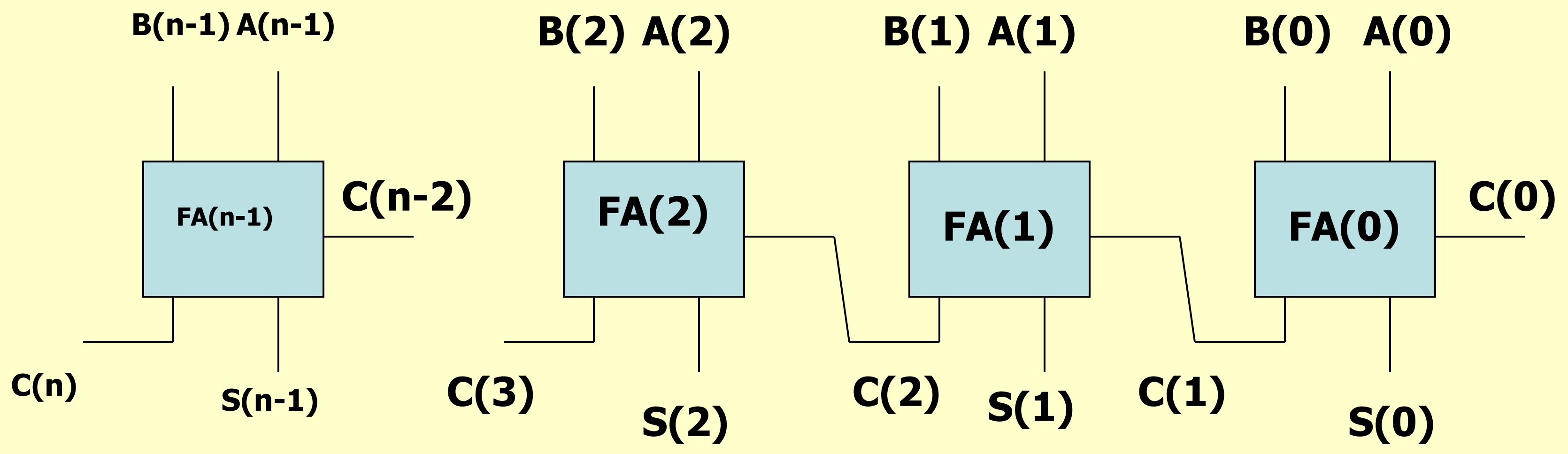
- Performance of a circuit
  - Circuit depth – maximum level in the topological sort.
  - Circuit Size – Number of combinational elements.
- Optimize both for high performance.
- Both are inversely proportional – so a balance to be arrived.

Carry Ripple Adder

- Given two n-bit numbers (a(n-1), a(n-2), a(n-3), …, a(0)) and (b(n-1), b(n-2), b(n-3) ,…, b(0)).
- A full adder adds three bits (a,b,c), where ‘a’ and ‘b’ are data inputs and ‘c’ is the carry-in bit. It outputs a sum bit ‘s’ and a carry-out bit ‘co’


a
b
a
c
b
c
co
Full Adder
a
b
c
s

n-bit carry ripple adder



Circuit Depth is ‘n’.

Circuit area is ‘n’ times size of a Full Adder

Carry look ahead adder

- The depth is ‘n’ because of the carry.
- Some interesting facts about carry

a(j)	b(j)	c(j)	c(j+1)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If a(j) = b(j) then

c(j+1) = a(j) = b(j)

If a(j) <> b(j) then

c(j+1) = c(j)

Carry Look-ahead Circuit

a(j)	b(j)	c(j+1)	Status (**x(j+1)**)
0	0	0	Kill (**k**)
0	1	c(j)	Propagate (**p**)
1	0	c(j)	Propagate (**p**)
1	1	1	Generate (**g**)

Carry Lookahead Circuit

x(j+1)

x(j)

(*)	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

New (j+1)th carry status as influenced by x(j)

(*) is associative

y(j) = x(0) (*) x(1) (*) … x(j)

x(0) = k

If y(j) = k then c(j) = 0

If y(j) = g then c(j) = 1

Note that y(j) <> p

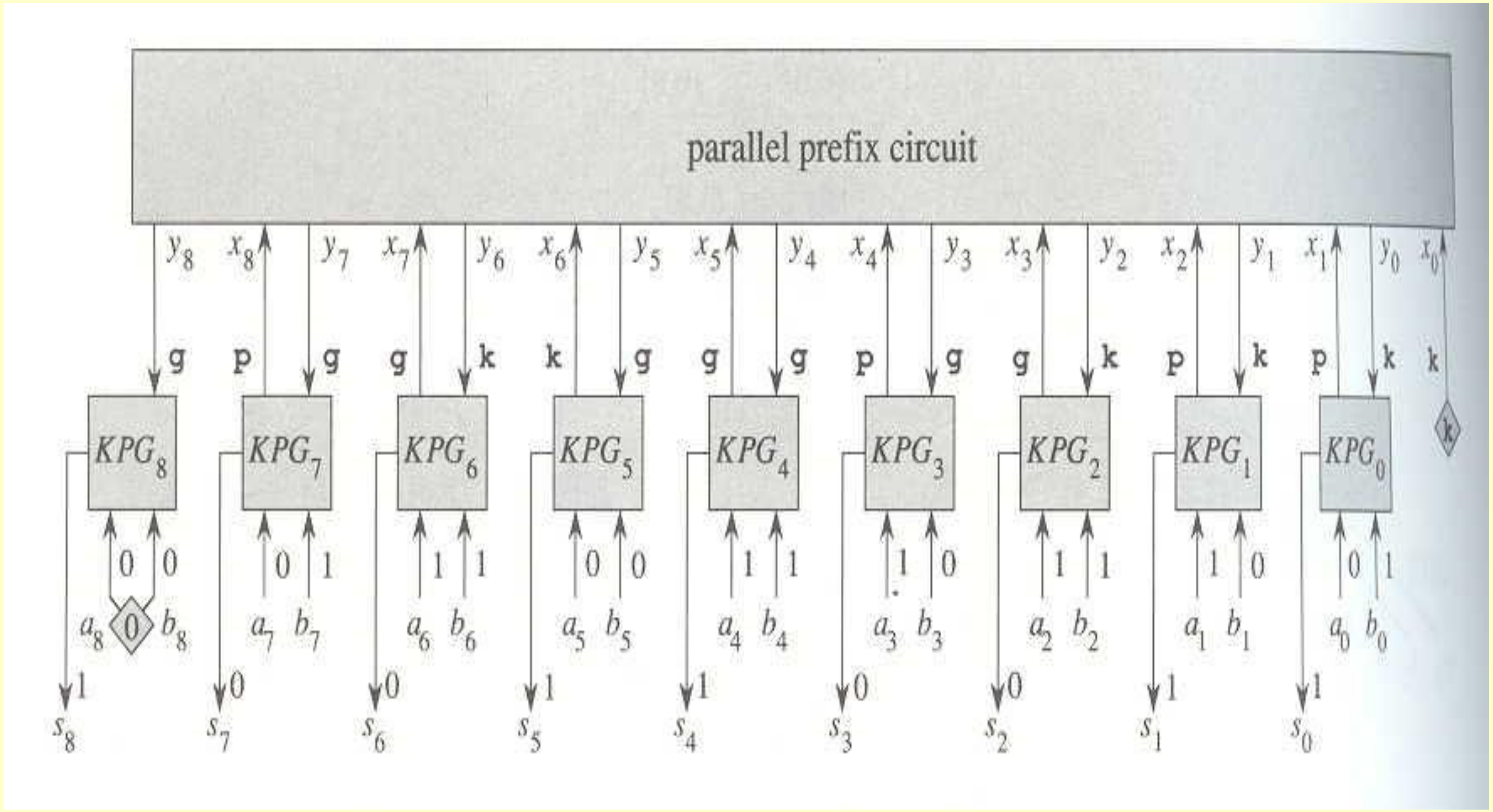
Carry Calculation

- A prefix computation
  - y(0) = x(0) = k
  - y(1) = x(0) (*) x(1)
  - y(2) = x(0) (*) x(1) (*) x(2)
  - …….
  - y(n) = x(0) (*) x(1) (*) …. x(n)
- Let [i,j] = x(i) (*) x(i+1) (*) … x(j)
- [i,j] (*) [j+1,k] = [i,k]
  - By associative property

Parallel Prefix circuit

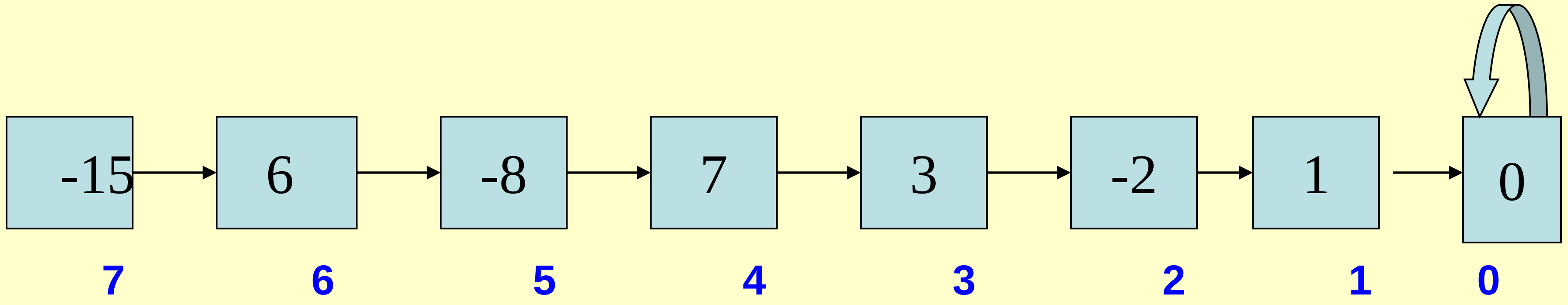
- Input x(0), x(1), … x(n) – for an n-bit CLA, where x(0) = k.
- Each x(i) is a 2-bit vector
- To compute the prefix (*) and pipeline the same.
- y(i) = x(0) (*) x(1) (*) … x(i)
- We can use the Recursive Doubling Technique described as follows

An 8-node Carry look ahead adder


parallel prefix circuit
y_8 x_8 y_7 x_7 y_6 x_6 y_5 x_5 y_4 x_4 y_3 x_3 y_2 x_2 y_1 x_1 y_0 x_0
g p g g k k g g g p g g k p k p k k
KPG_8 KPG_7 KPG_6 KPG_5 KPG_4 KPG_3 KPG_2 KPG_1 KPG_0
0 0 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 1
a_8 0 b_8 a_7 b_7 a_6 b_6 a_5 b_5 a_4 b_4 a_3 b_3 a_2 b_2 a_1 b_1 a_0 b_0
1 0 0 1 1 0 0 1 1
s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

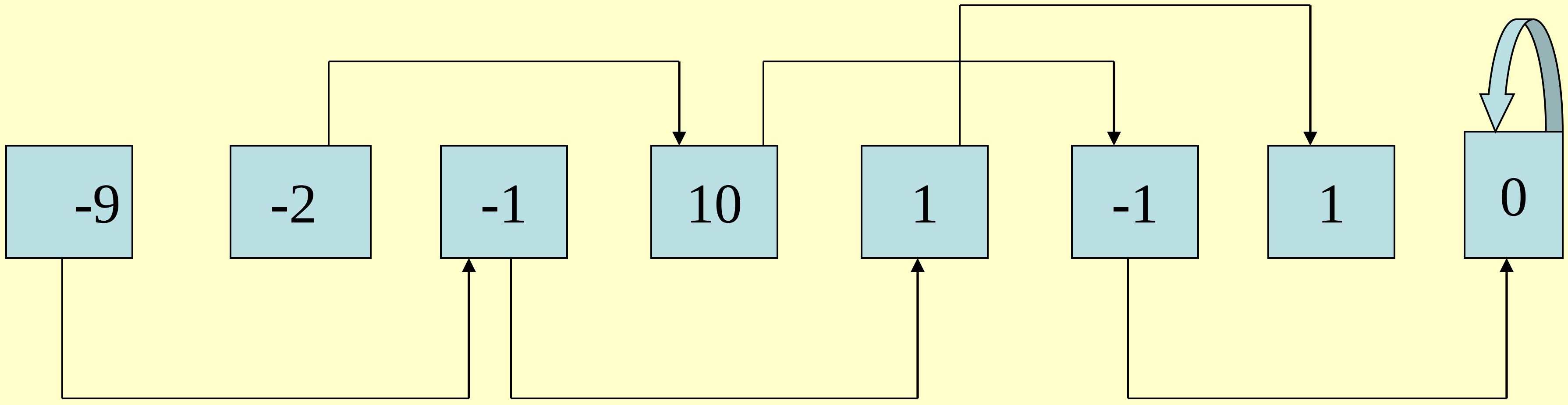
Parallel Techniques:Recursive Doubling Technique

- **Finding Prefix Sum of ‘8’ Numbers**



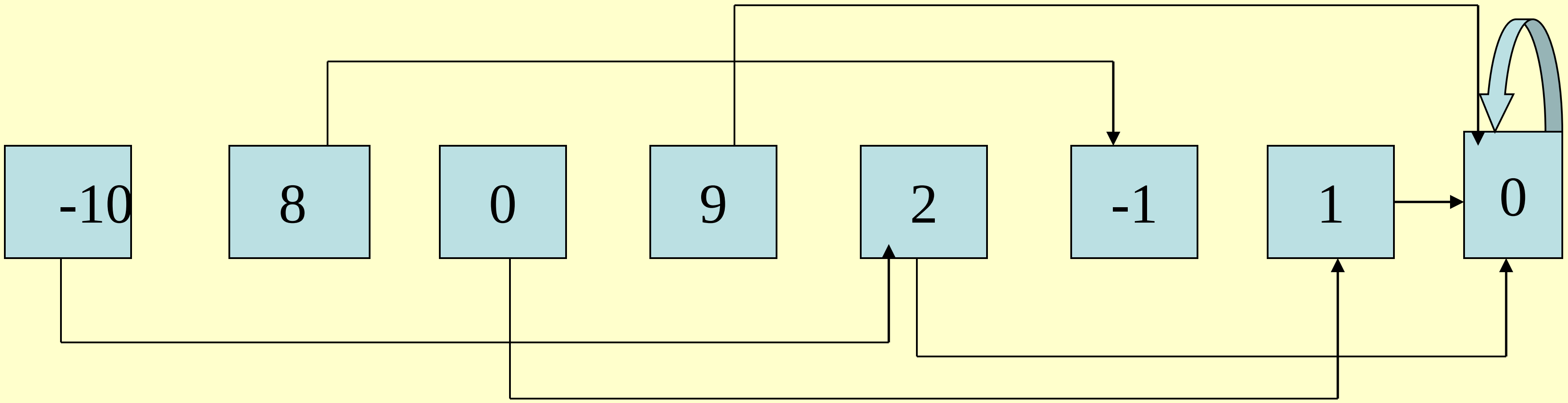
Recursive Doubling Technique (Step 1)

- Finding Prefix Sum of ‘8’ Numbers



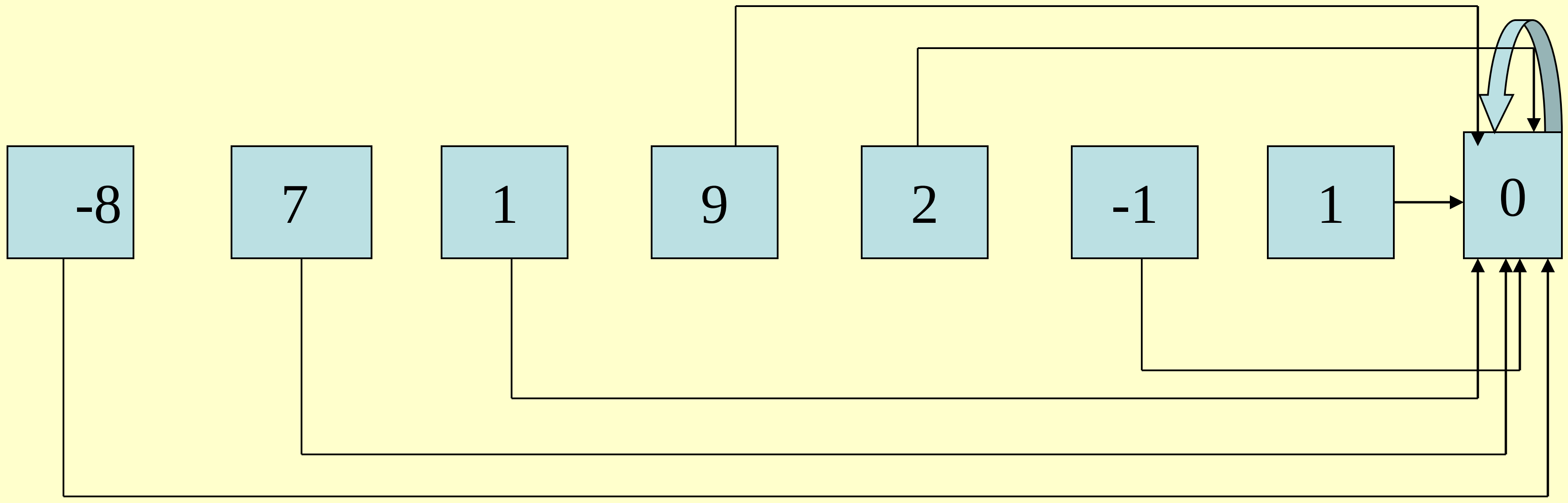
Step 1

Recursive Doubling Technique (Step 2)



Step 2

Recursive Doubling Technique (Step 3)

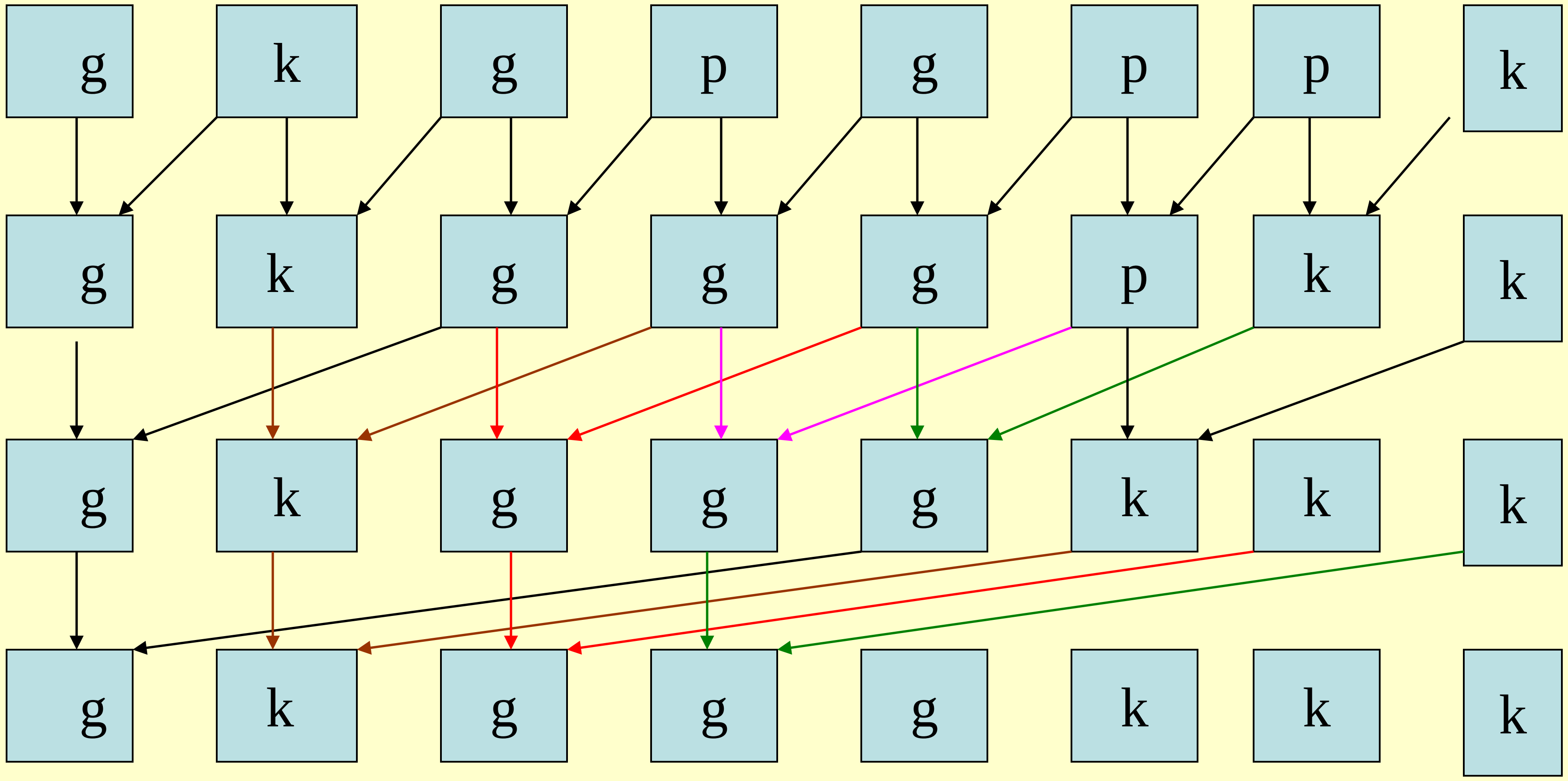


Step 3

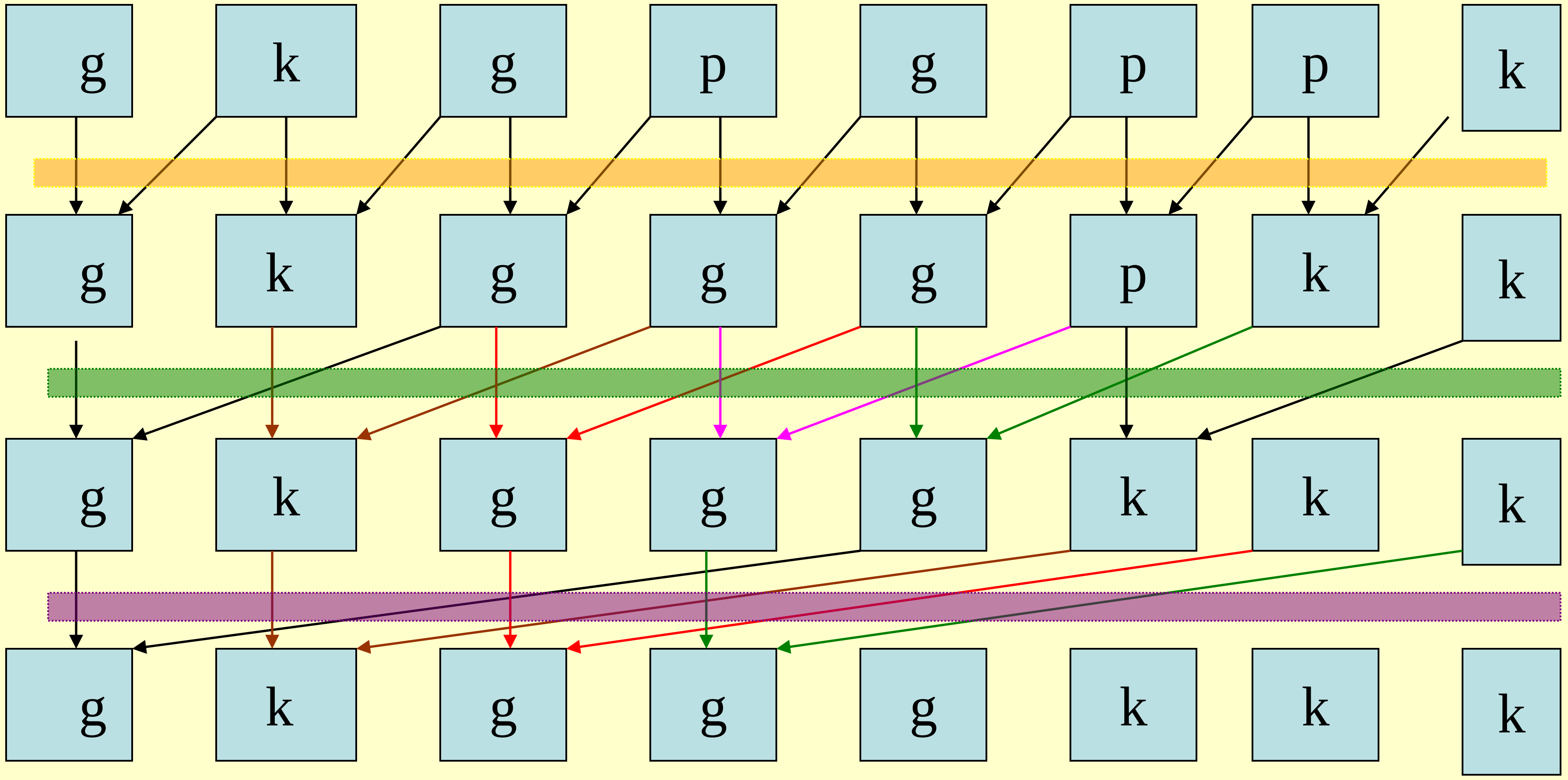
- **Prefix Sum of n numbers in log n steps**

- **Applicable for any semi-group operator like min , max , mul etc. that is associative**

Pipelined Prefix Calculation based on Recursive Doubling Technique



Pipelined Prefix Calculation based on Recursive Doubling Technique



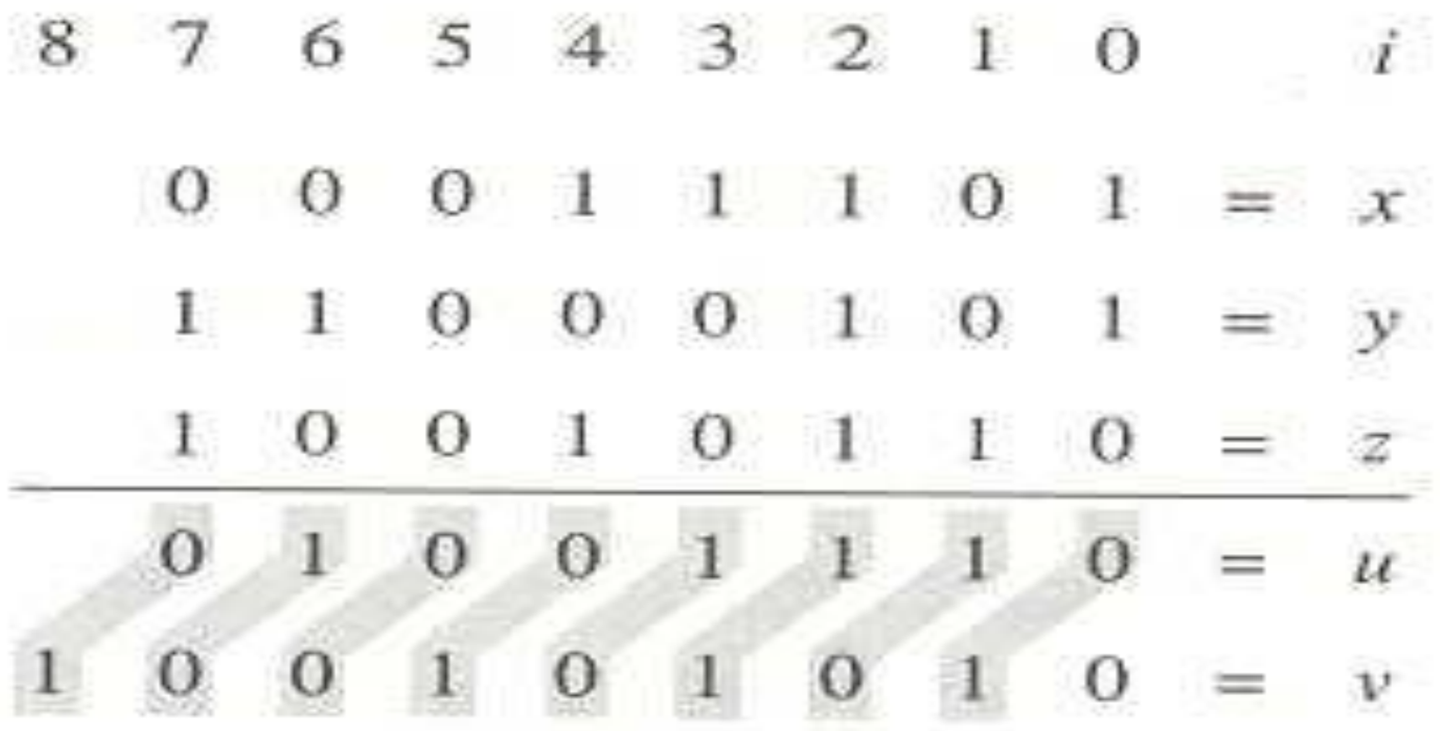
What is achieved?

- The depth of circuit reduced from O(n) to O($\log_2 n$)
- Size is still O(n)
- This results in a fast adder.

Carry Save Addition

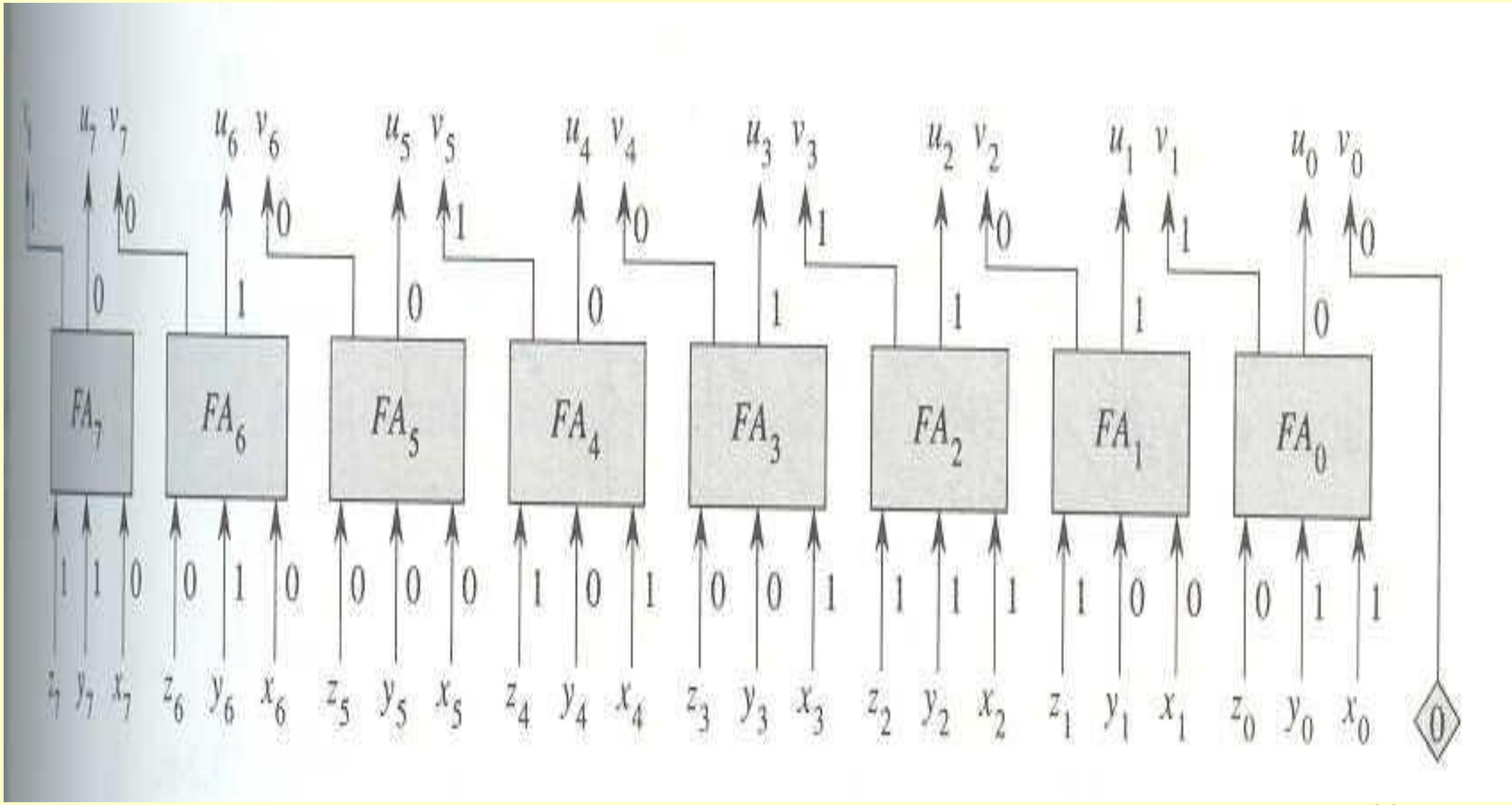
- Given three n-bit numbers x, y and z.
- The circuit computes a n-bit number ‘u’ and a (n+1)-bit number ‘v’ such that
  - $x+y+z = u + v$.

Carry Save addition - example



8	7	6	5	4	3	2	1	0		i
	0	0	0	1	1	1	0	1	=	x
	1	1	0	0	0	1	0	1	=	y
	1	0	0	1	0	1	1	0	=	z
	0	1	0	0	1	1	1	0	=	u
1	0	0	1	0	1	0	1	0	=	v

Carry Save Adder Circuit



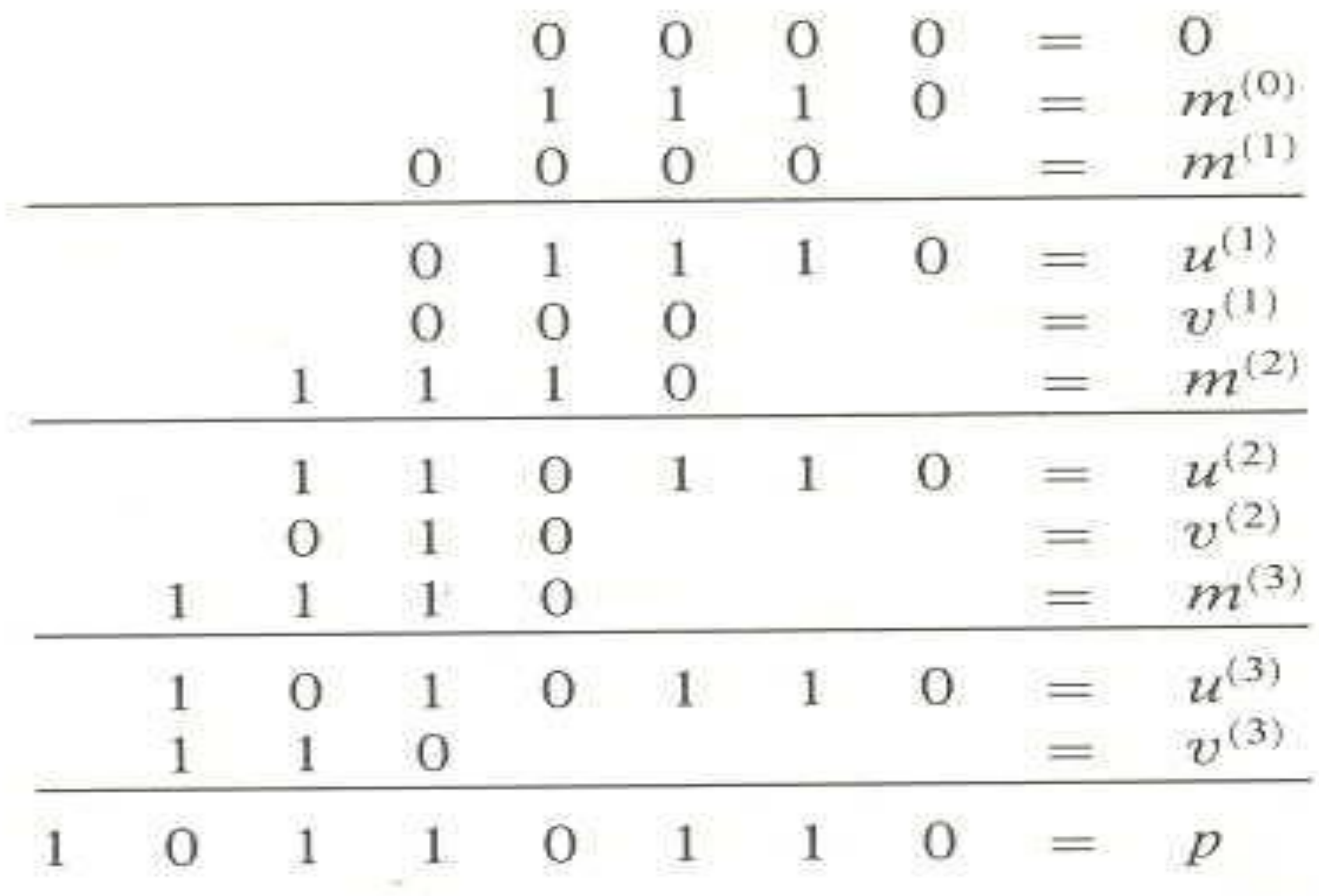
Multipliers

- Simple grade-school multiplication method.
  - Concept of partial-products
- Partial products generated in parallel and carry save addition results in faster array multiplier

Grade-school multiplication

```
•               1 1 1 0 = a
•               1 1 0 1 = b
• ------------------------------
•               1 1 1 0 = m(0)
•             0 0 0 0   = m(1)
•           1 1 1 0     = m(2)
•         1 1 1 0       = m(3)
• -------------------------------
•       1 0 1 1 0 1 1 0 = p
```

Carry Save Addition based Multiplication

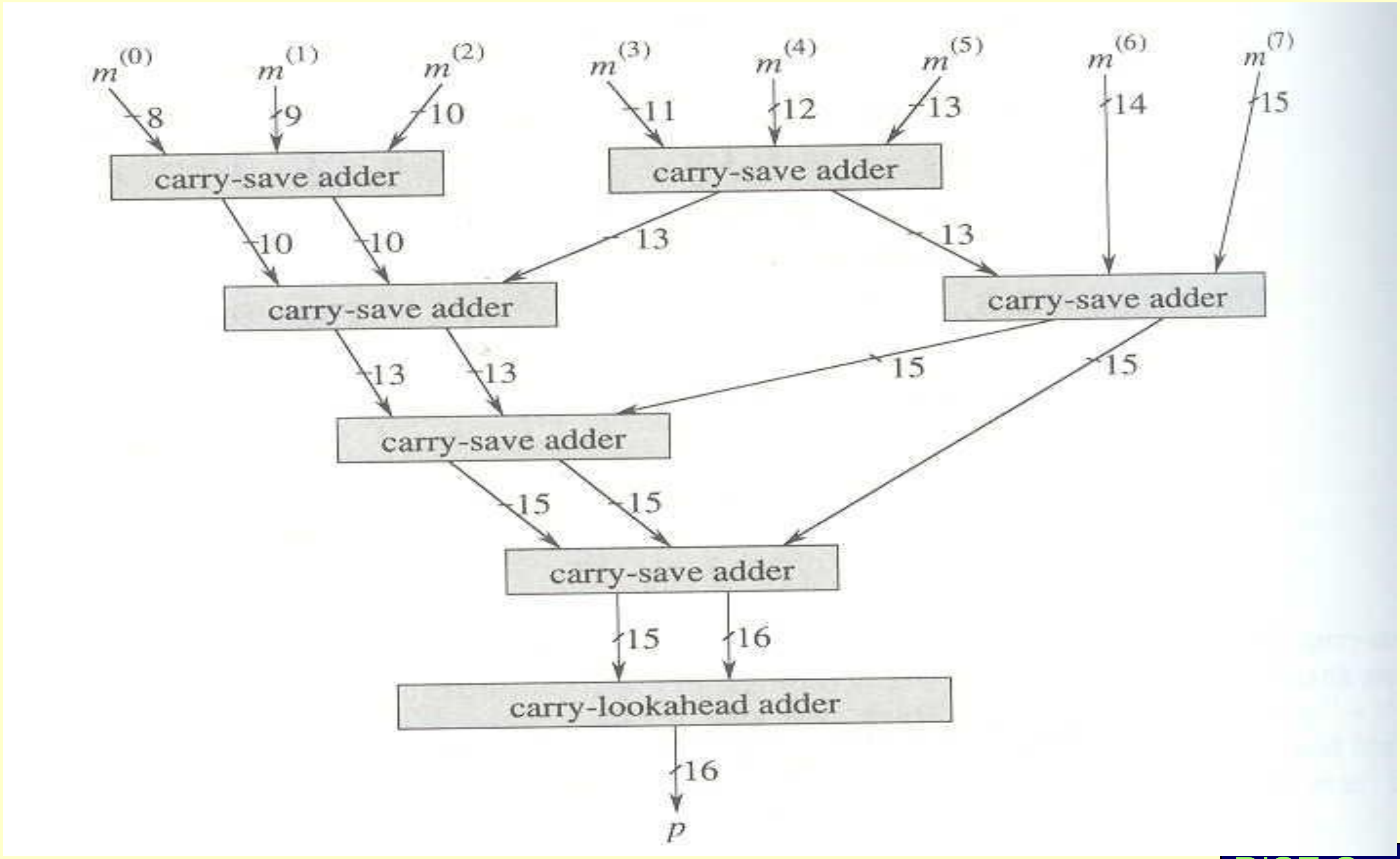


				0	0	0	0	=	0
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
			0	1	1	1	0	=	$u^{(1)}$
			0	0	0			=	$v^{(1)}$
		1	1	1	0			=	$m^{(2)}$
		1	1	0	1	1	0	=	$u^{(2)}$
		0	1	0				=	$v^{(2)}$
	1	1	1	0				=	$m^{(3)}$
	1	0	1	0	1	1	0	=	$u^{(3)}$
	1	1	0					=	$v^{(3)}$
1	0	1	1	0	1	1	0	=	p

Wallace-Tree Multipliers

- While multiplying two n-bit numbers a total of n partial products have to be added.
- Use floor(n/3) carry save adders and reduce the number to ceil(2n/3).
- Apply it recursively.
- O(log n) depth circuit of size O(n^2).

8-bit Wallace-tree multiplier



Questions ??

Resources

- www.cs.iitm.ernet.in/~noorse (My Home Page)
- http://vlsi.cs.iitm.ernet.in
- ftp://icarus.com/pub/eda/verilog/v0.8/
- http://www.icarus.com/eda/verilog
- http://bleyer.org/icarus/ (For windows)
- http://www.geda.seul.org/download.html
- http://www.geocities.com/SiliconVally/Campus/3216/GTKwave/gtkwave-win32.html (Windows)
- http://www.altera.com/education/univ/unv-index.html
- http://www.xilinx.com/univ/