Chapter 4 Introduction to logic design with Verilog

- **Designers using an HDL:**

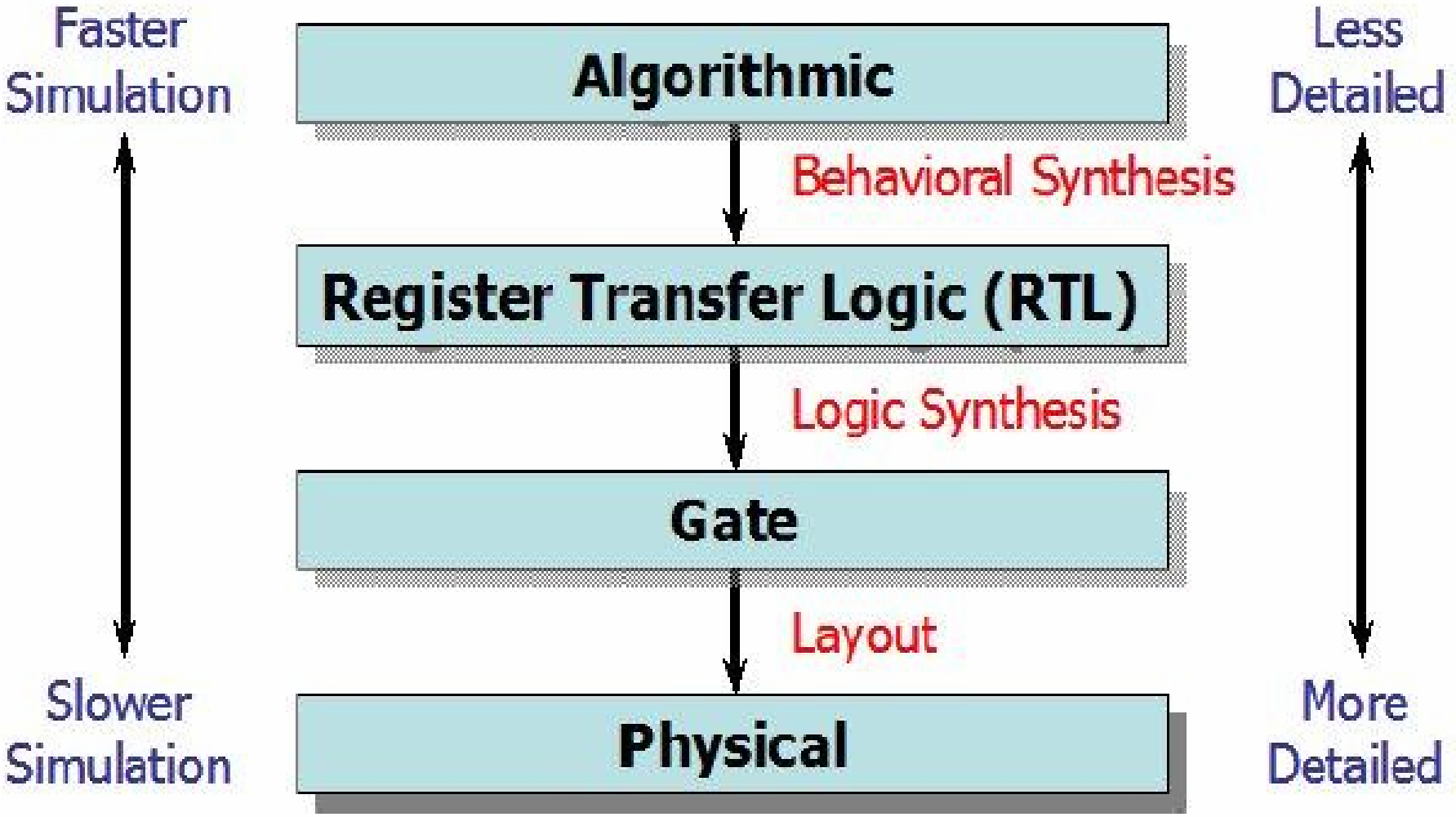
  VHDL or Verilog

  - **Write a text-based description** (model) of a circuit
  - **Compile the description to verify its syntax**
  - **Simulate the model to verify the functionality of the design**

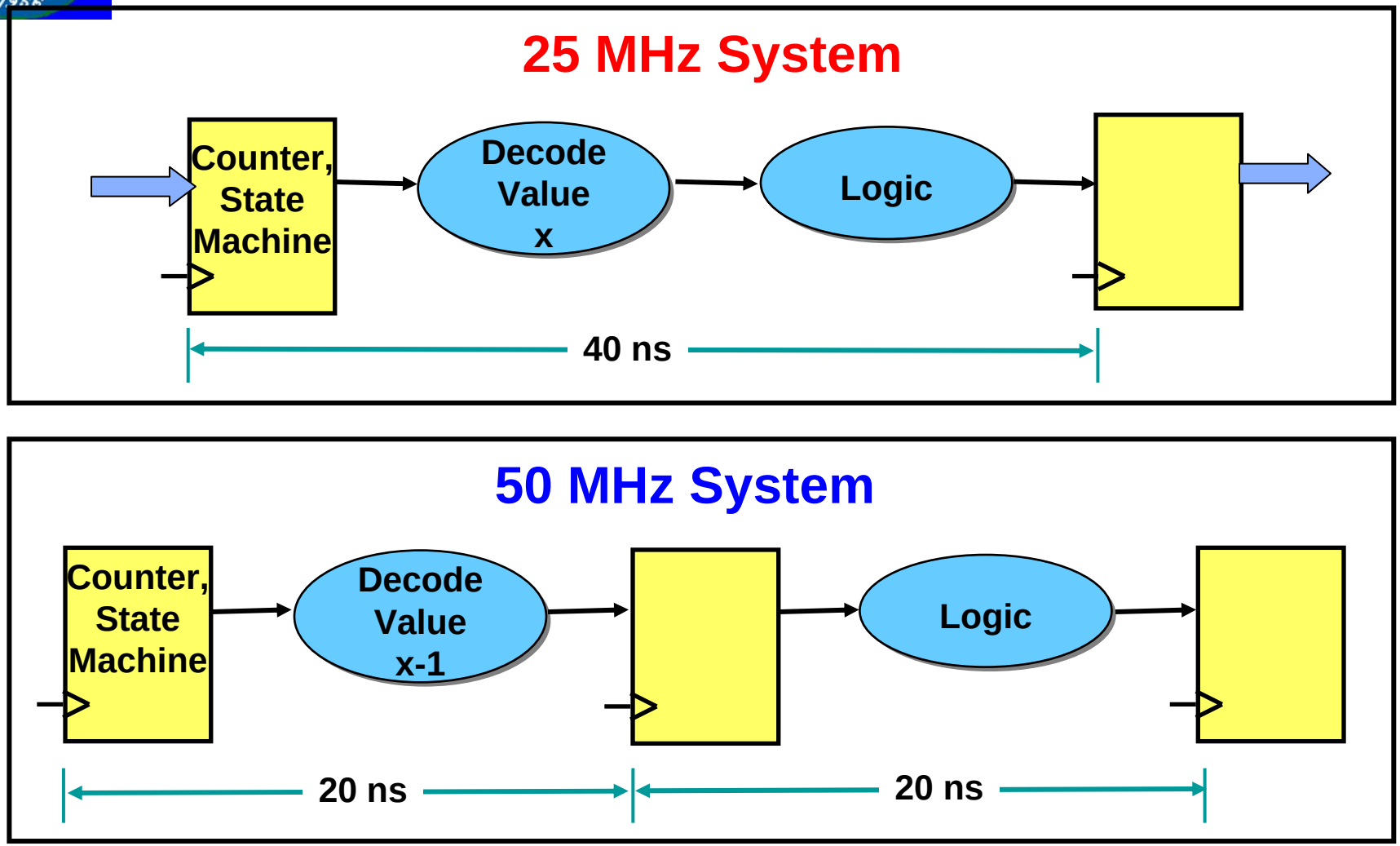
Content of this chapter

- Structure models of combinational logic
- Logic Simulation (not Emulation), design verification, and test methodology
- Propagation delay
- Truth table models of combinational and sequential logic with Verilog

ASIC Desin Flow



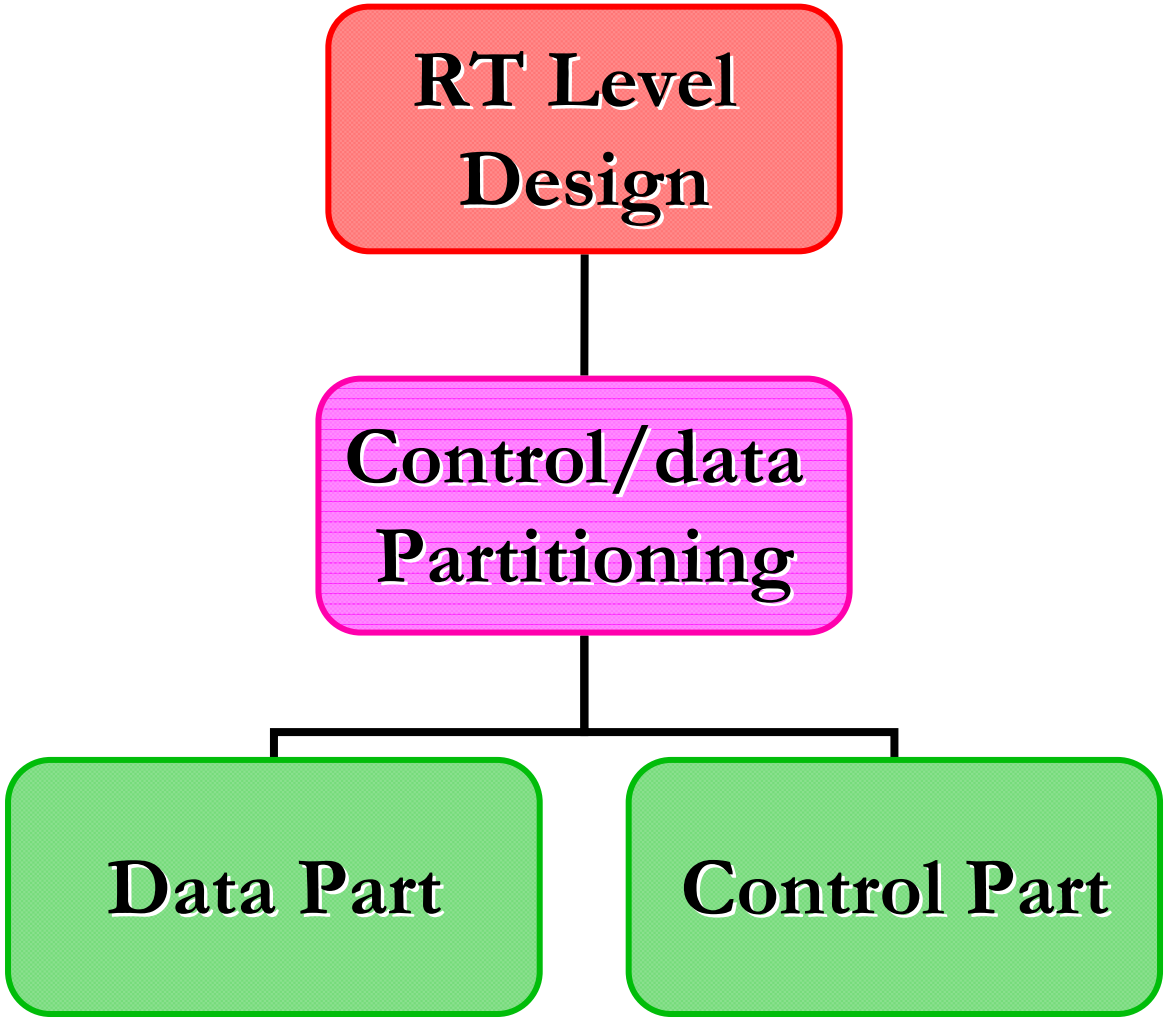
Register Transfer Level Design



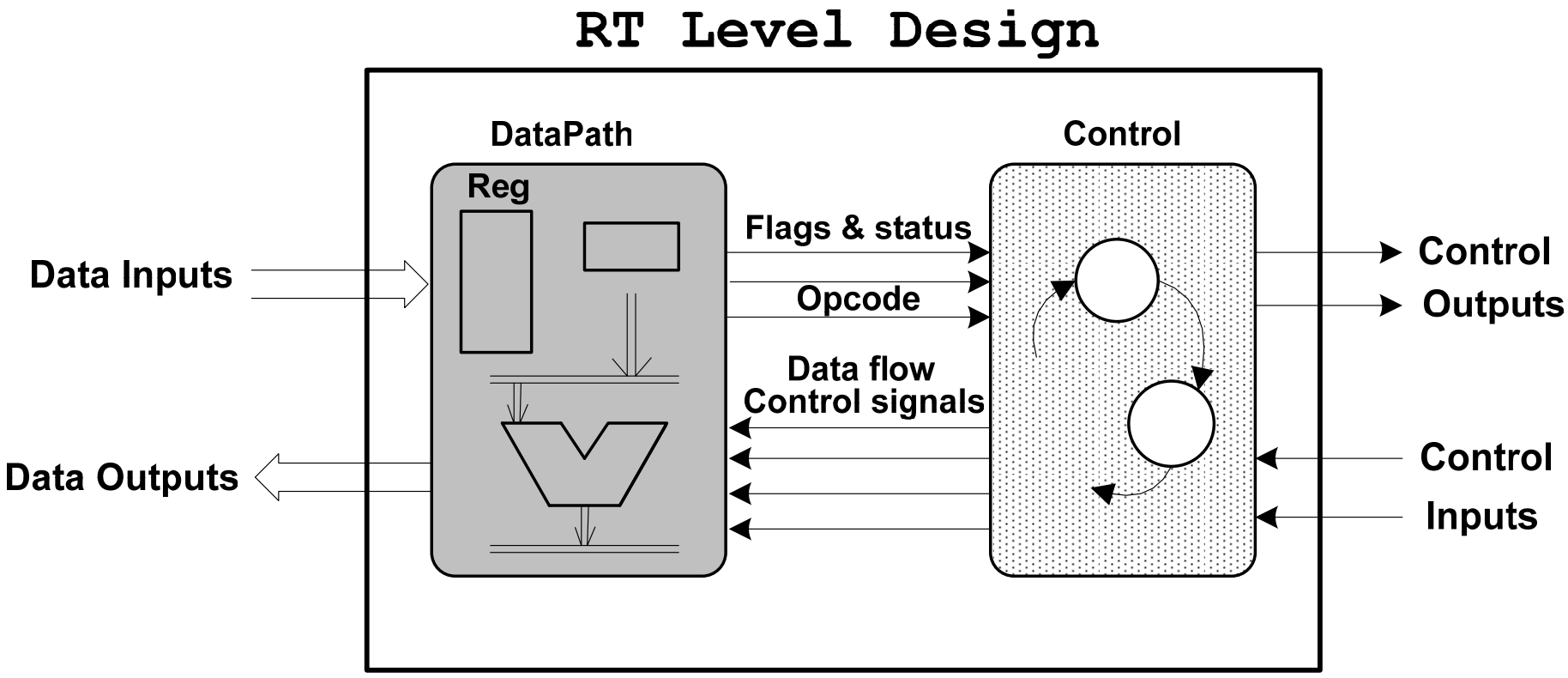
RTL Design

- **RT level design:**
  - **Taking a high level description of a design**
  - **Partitioning**
  - **Coming up with an architecture**
  - **Designing the bussing structure**
  - **Describing and implementing various components of the architecture**
- **Steps in RT level design:**
  - **Control/Data Partitioning**
  - **Data Part Design**
  - **Control Part Design**

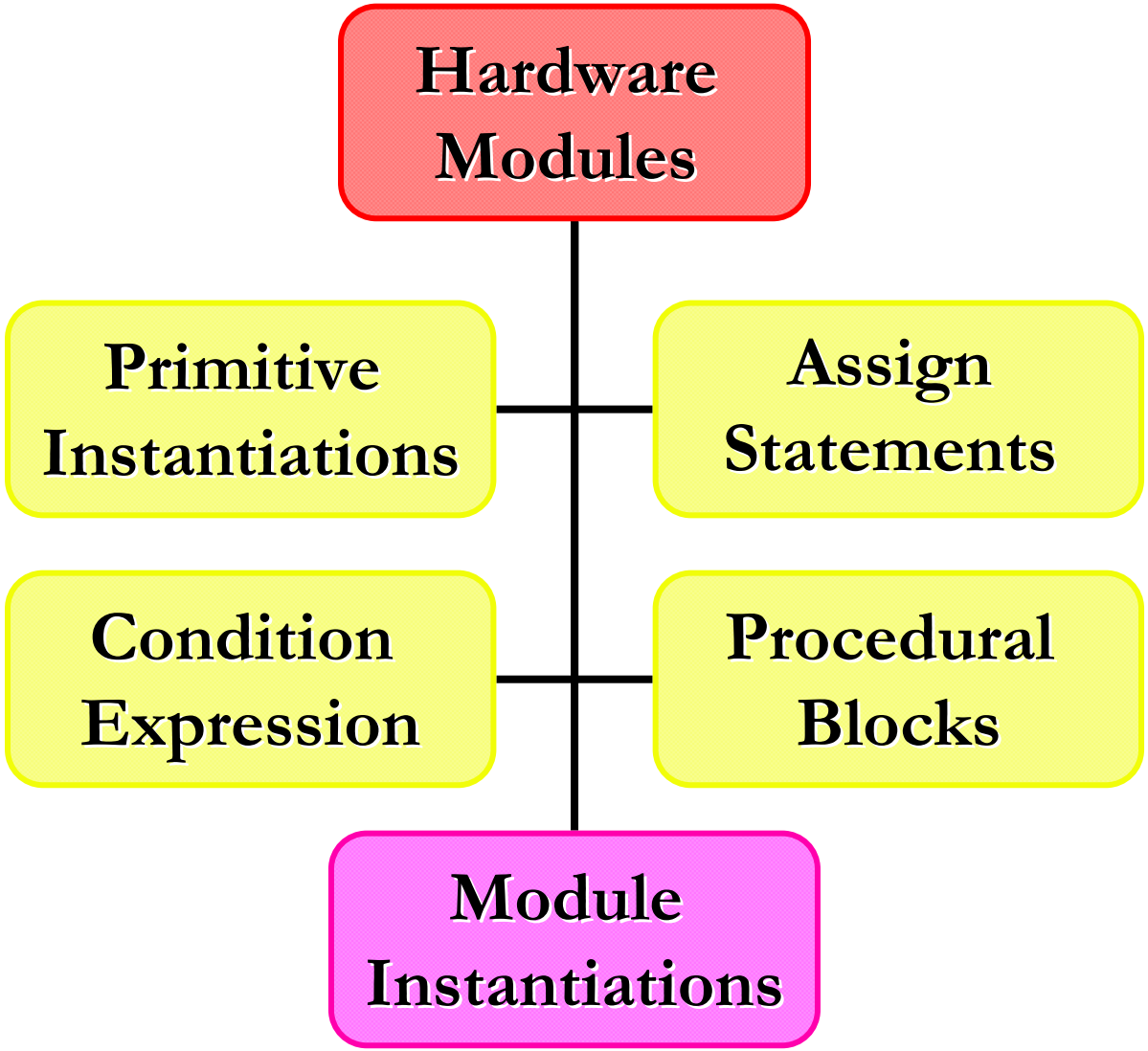
RT Level Design



Control/Data Partitioning



Elements of Verilog



Hardware Modules

module : The Main Component of Verilog

Keyword *module*

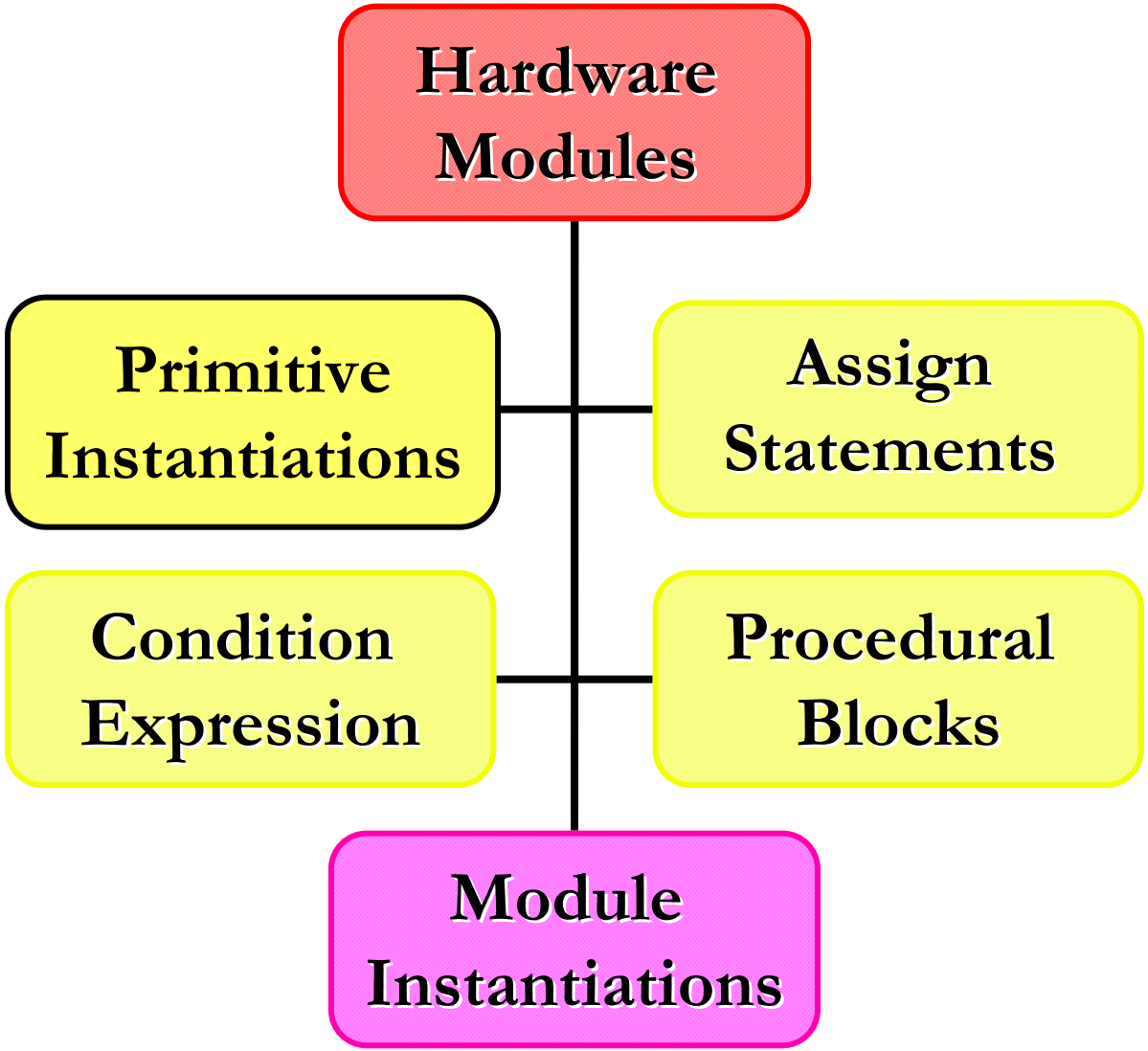
```
module module-name
  List of ports;
  Declarations
  ...
  Functional specification of module
  ...
endmodule
```

Variables, wires, and module parameters are declared.

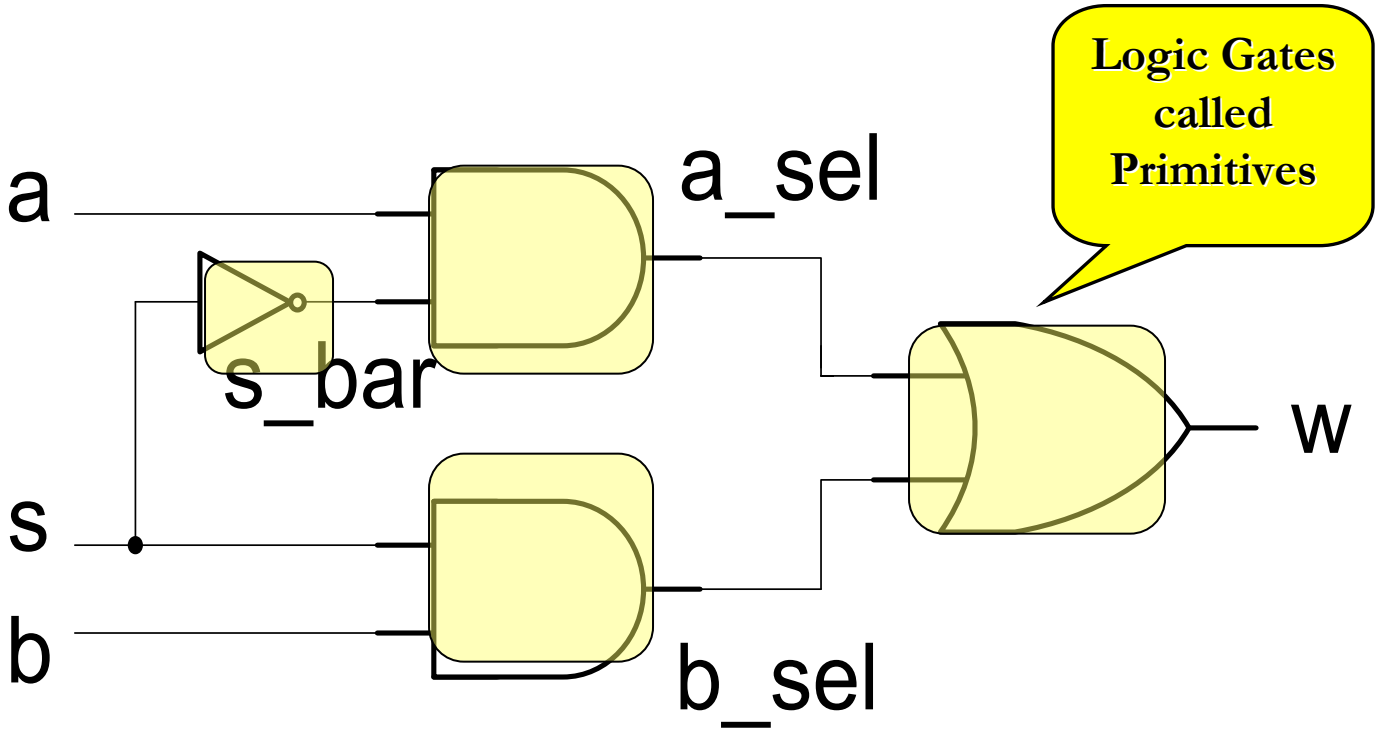
Keyword *endmodule*

- Module Specifications

Primitive Instantiations



Primitive Instantiations



- A Multiplexer Using Basic Logic Gates

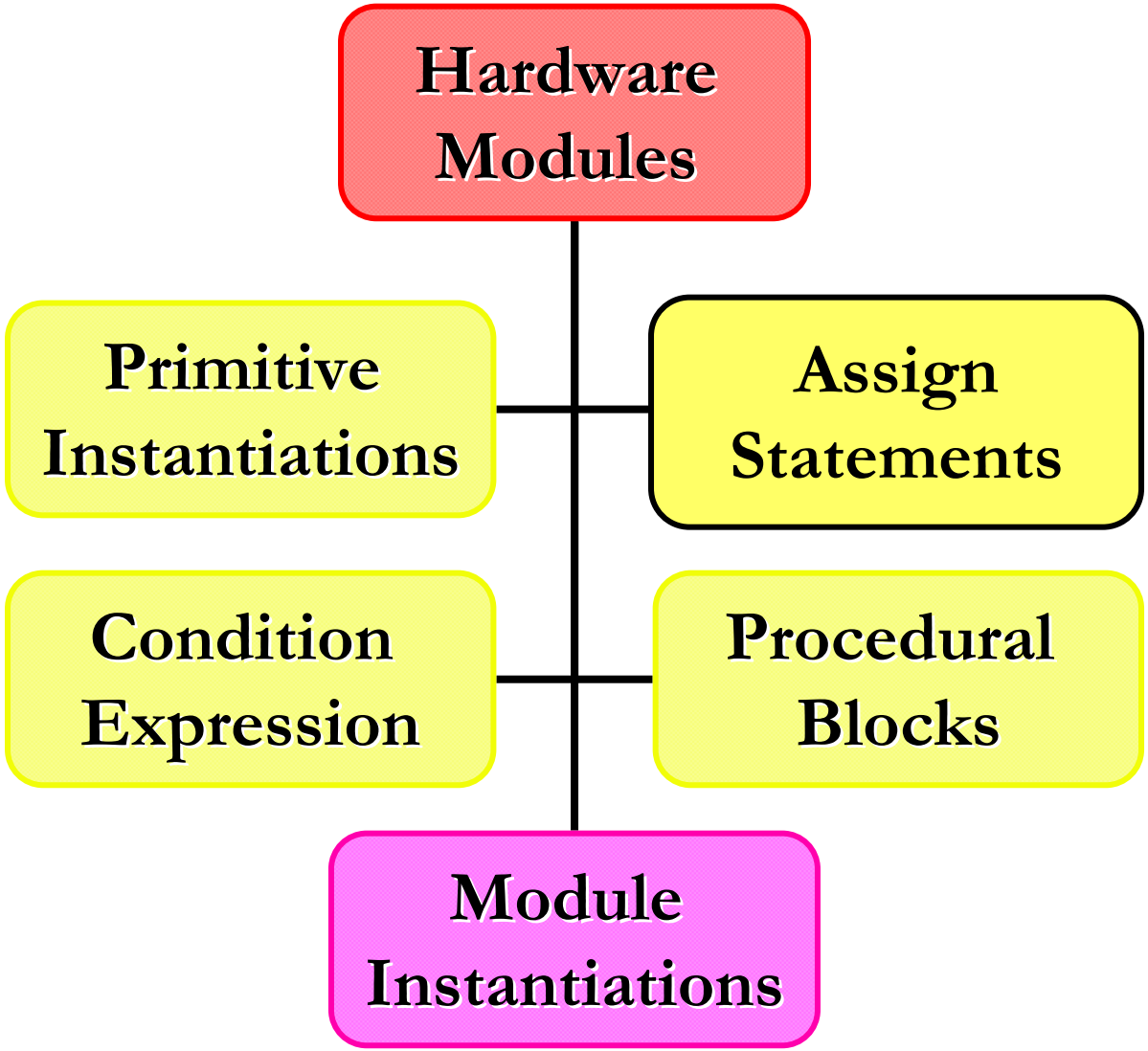
Primitive Instantiations

```
module MultiplexerA (input a, b, s, output w);
  wire a_sel, b_sel, s_bar;
  not  U1 (s_bar, s);
  and  U2 (a_sel, a, s_bar);
  and  U3 (b_sel, b, s);
  or   U4 (w, a_sel, b_sel);
endmodule
```

Instantiation of Primitives

- Primitive Instantiations

Assign Statements



Assign Statements

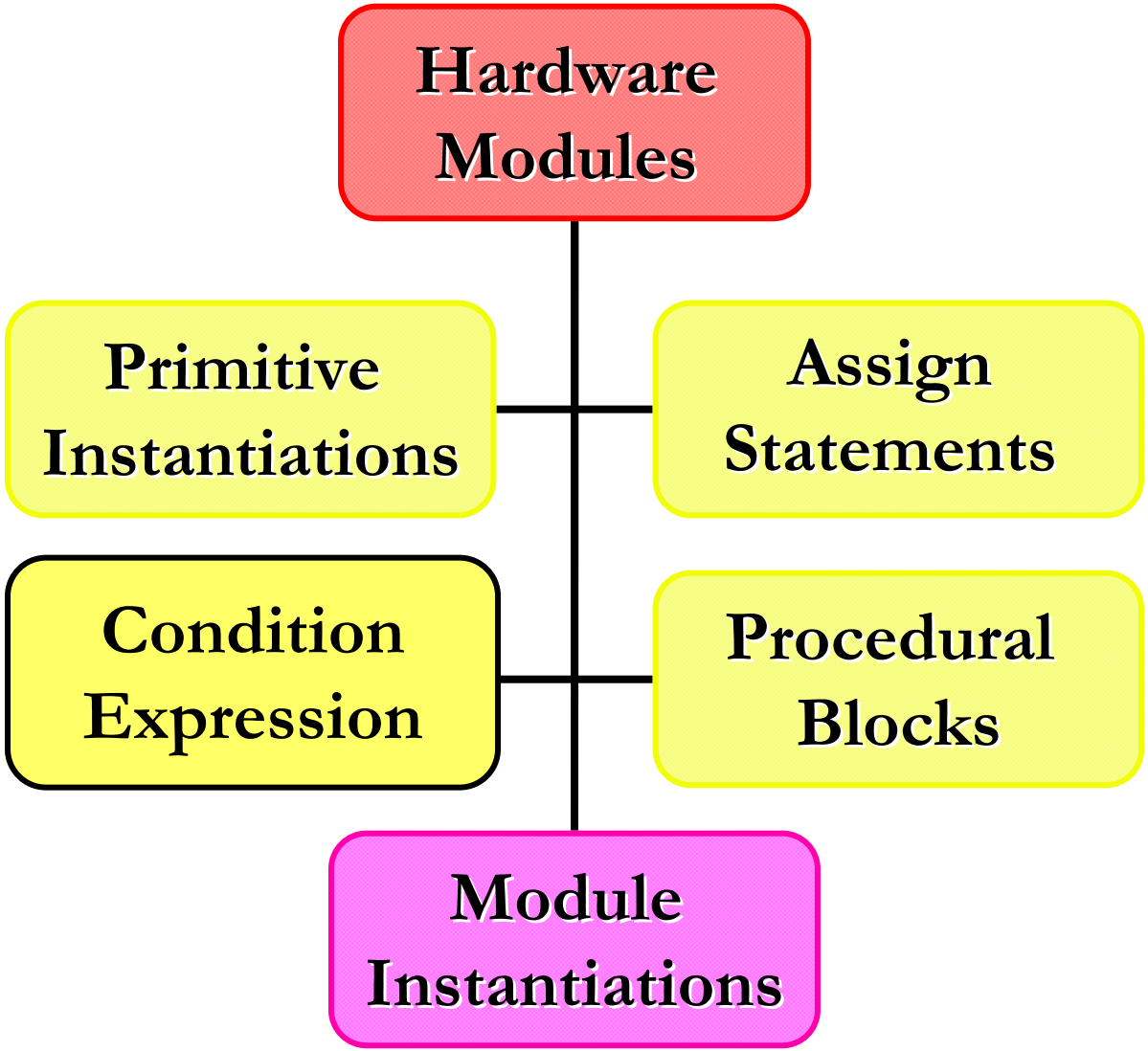
Continuously drives *w* with the right hand side expression

```
module MultiplexerB (input a, b, s, output w);
  assign w = (a & ~s) | (b & s);
endmodule
```

Using Boolean expressions to describe the logic

- Assign Statement and Boolean

Condition Expression



Condition Expression

Can be used when the operation of a unit is too complex to be described by Boolean expressions

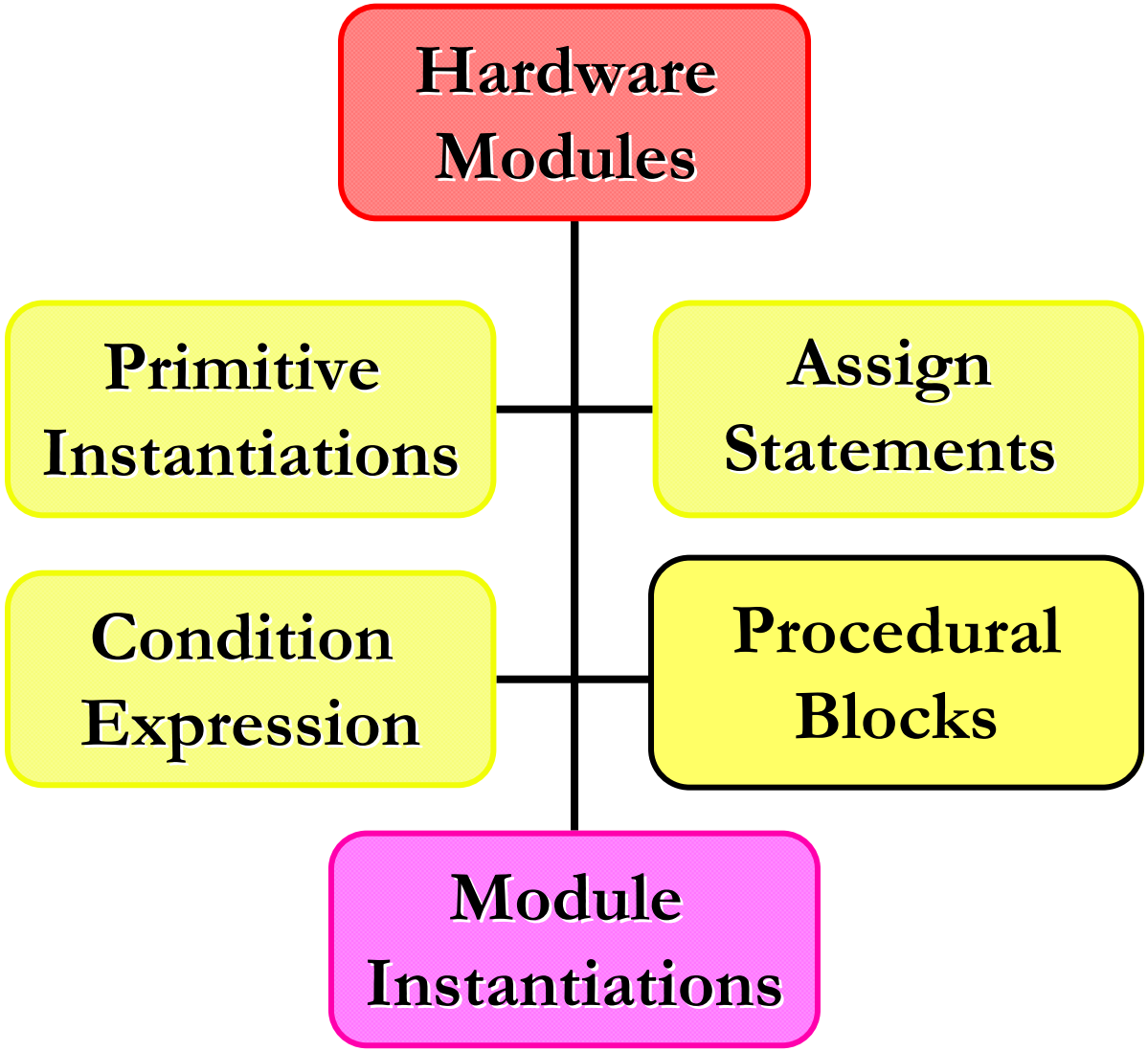
```
module MultiplexerC (input a, b, s, output w);
  assign w = s ? b : a;
endmodule
```

- Assign Statement and Condition Operator

Very Effective in describing complex functionalities

Useful in describing a behavior in a very compact way

Procedural Blocks



Procedural Blocks

always statement

Sensitivity list

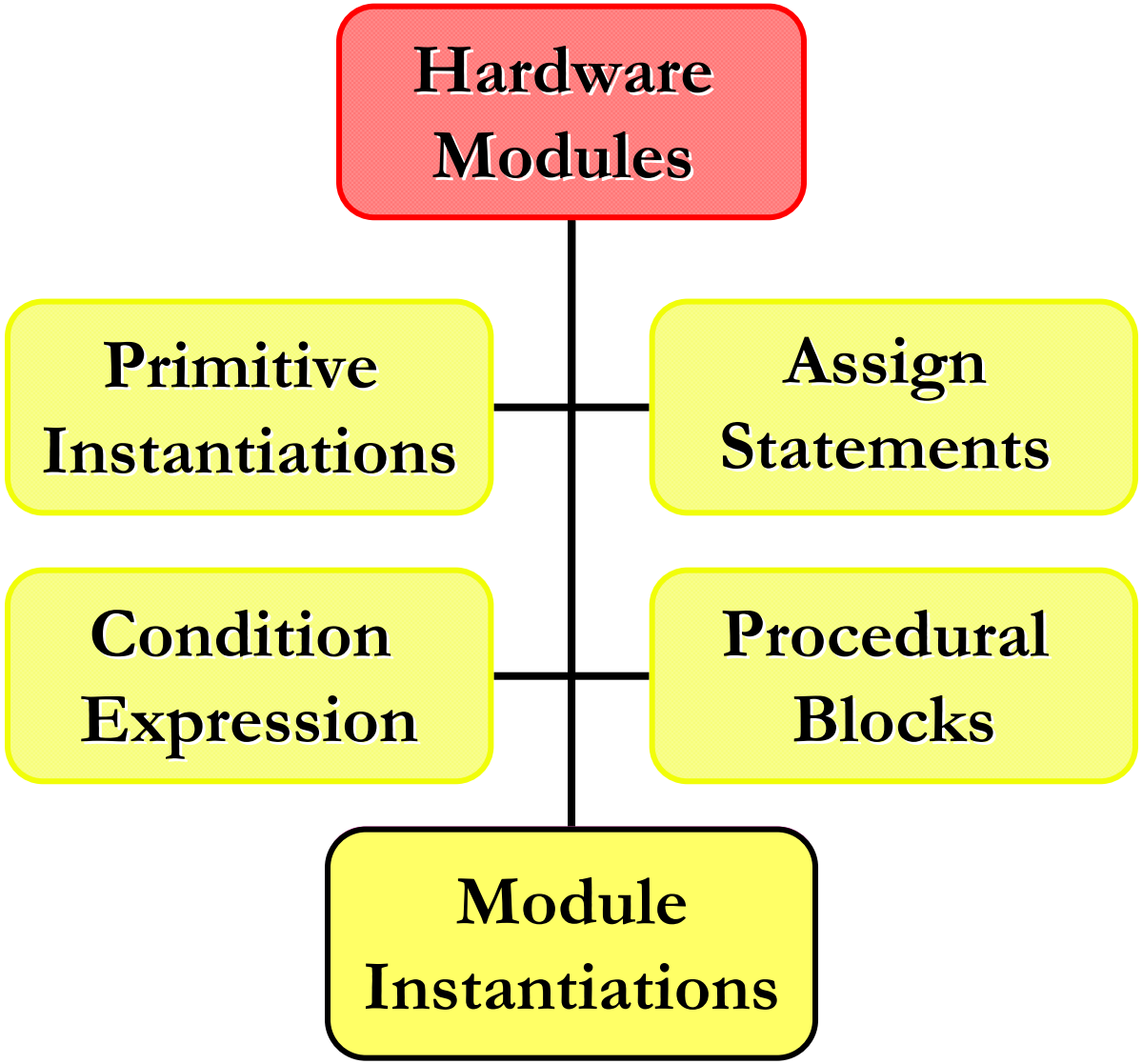
```
module MultiplexerD (input a, b, s, output w);
   reg w;
   always @(a, b, s) begin
      if (s) w = b;
      else w = a;
   end
endmodule
```

if-else statement

Can be used when the operation of a unit is too complex to be described by Boolean or conditional expressions

- Procedural Statement

Module Instantiations



Module Instantiations

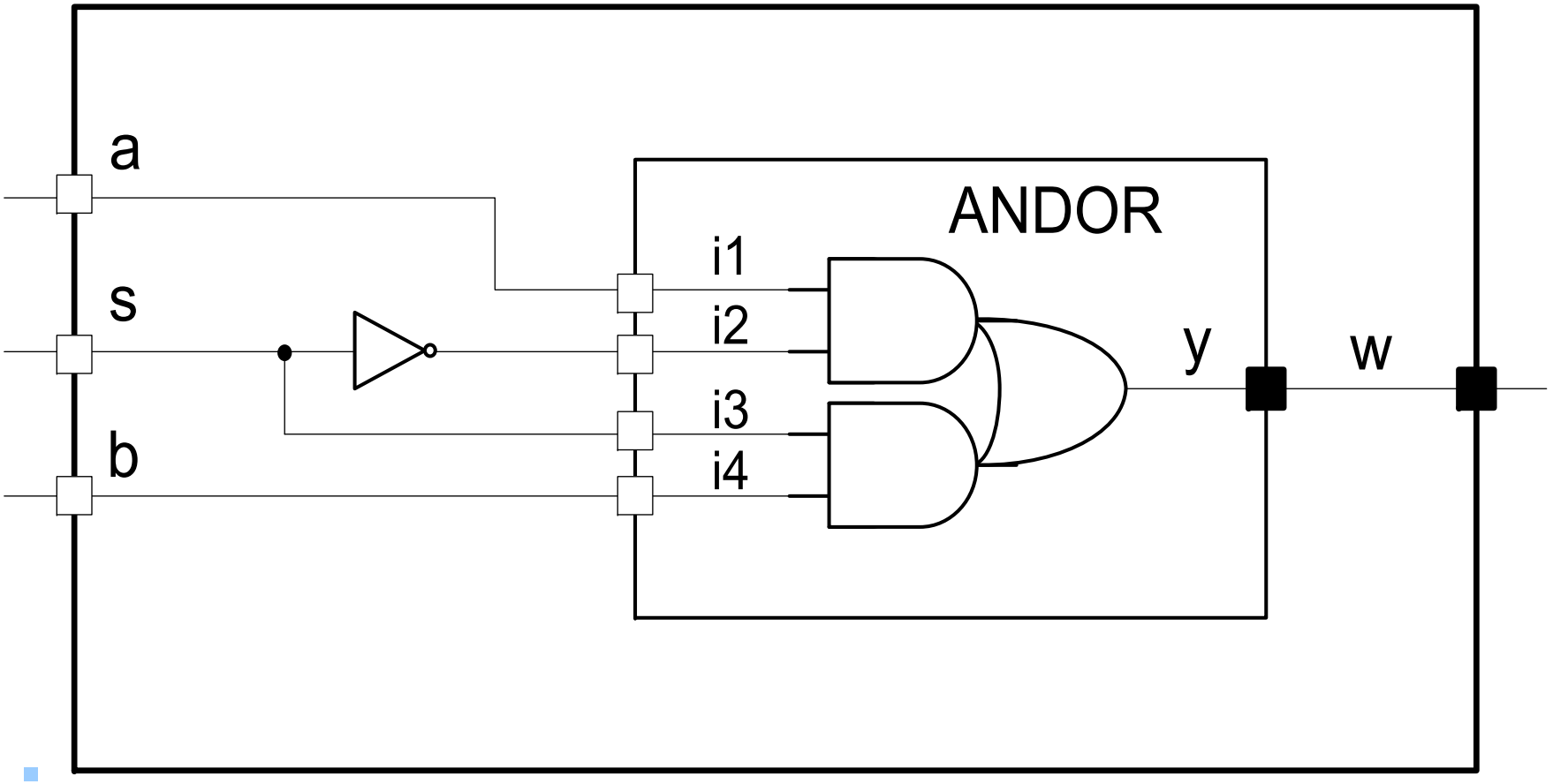
```
module ANDOR (input i1, i2, i3, i4, output y);
    assign y = (i1 & i2) | (i3 & i4);
endmodule
//
module MultiplexerE (input a, b, s, output  w);
    wire s_bar;
    not U1 (s_bar, s);
    ANDOR U2 (a, s_bar, s, b, w);
endmodule
```

ANDOR module is defined

ANDOR module is instantiated

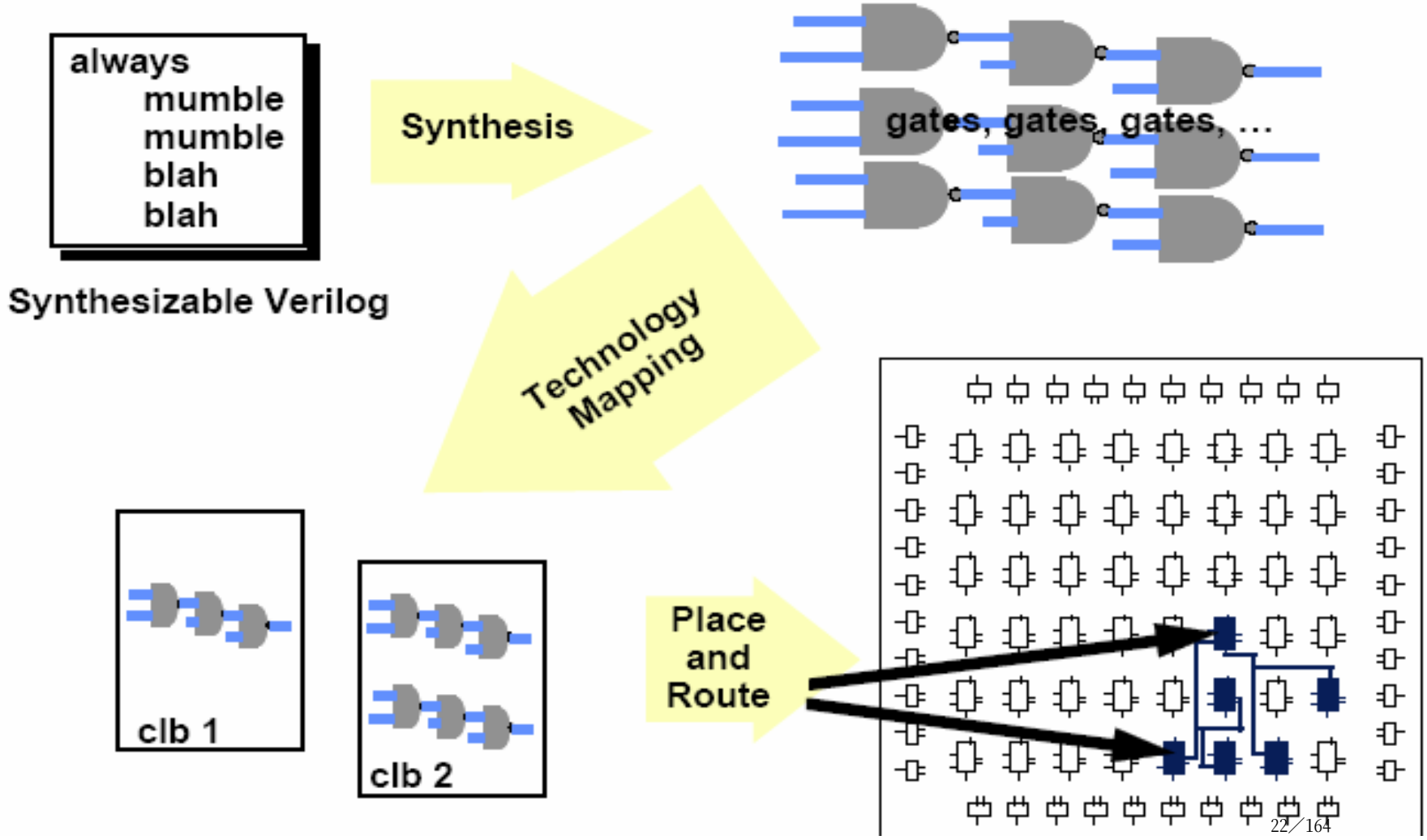
- Module Instantiation

Module Instantiations



Modern Design Methodology

Simulation and Synthesis are components of a design methodology



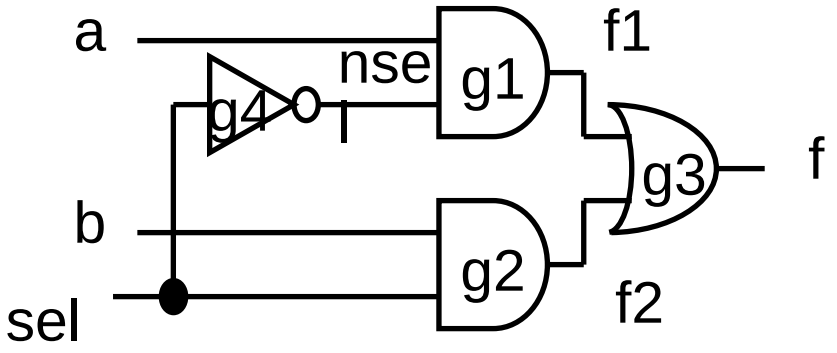
Verilog HDL

- **Serves as a vehicle for designing, verifying and synthesizing a circuit**
- Use a top-down methodology to partition a complex design unit smaller functional units
- Let designers create a design hierarchy of functional units

Multiplexer

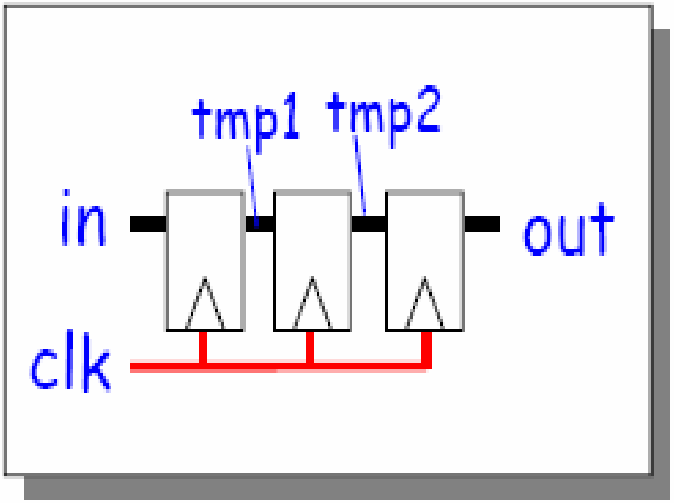
```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and  g1(f1, a, nsel),
     g2(f2, b, sel);
or   g3(f, f1, f2);
not  g4(nsel, sel);

endmodule
```

Identifiers not explicitly defined default to wires



Delayline Circuit

```
module DELAYLINE(out, in, clk);

output out;
input in, clk;
reg out;
reg tmp1,tmp2;

always @(posedge clk)
begin
    out <= tmp2;
    tmp2 <= tmp1;
    tmp1 <= in;
end

endmodule
```



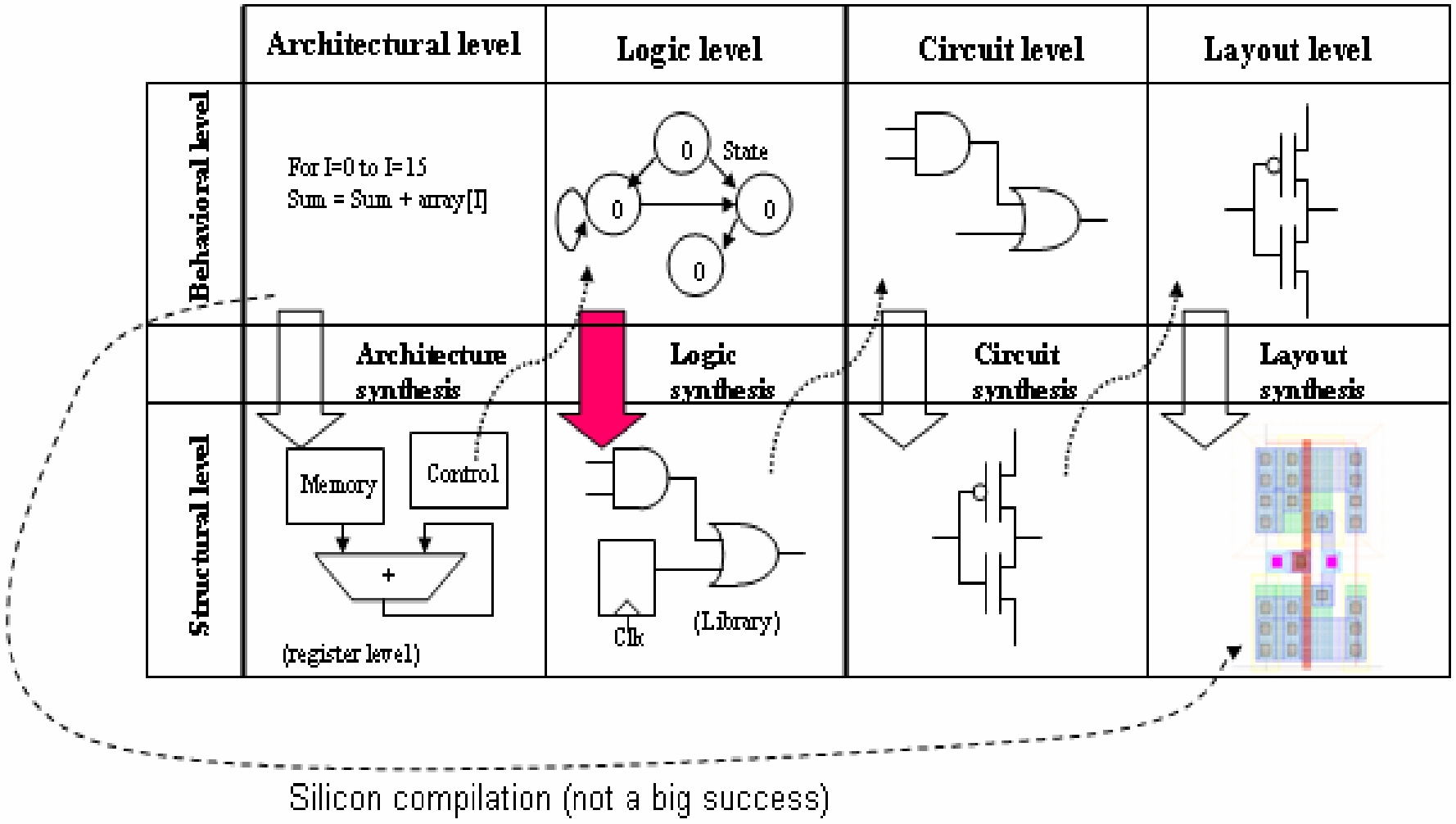
Summary of Verilog

- Systems described hierarchically
  - Modules with interfaces
  - Modules contain instances of primitives, other modules
  - Modules contain initial and always blocks
- Based on discrete-event simulation semantics
  - Concurrent processes with sensitivity lists
  - Scheduler runs parts of these processes in response to changes

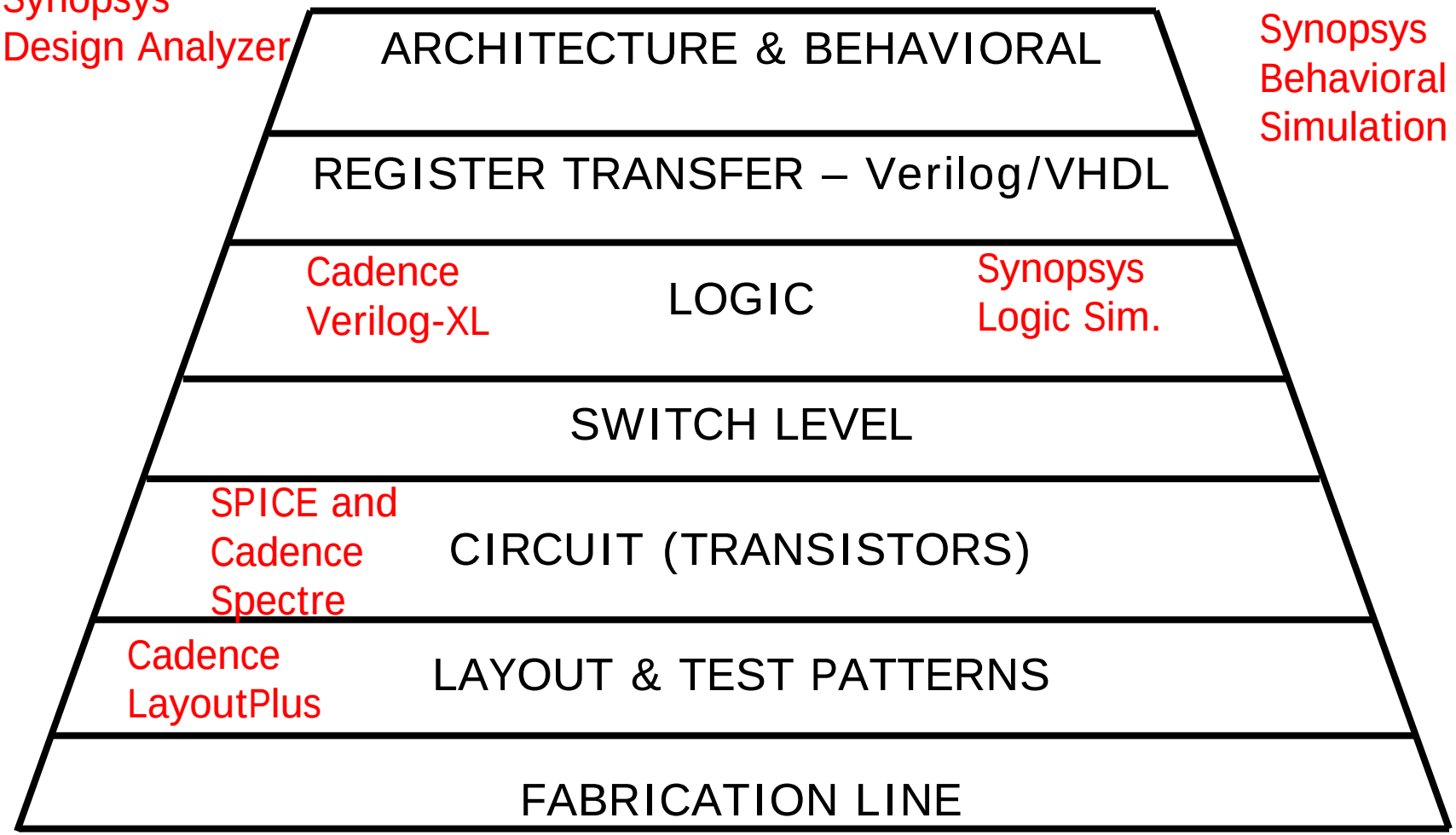
Modeling Tools

- **Switch-level primitives**
  - CMOS transistors as switches that move around charge
- **Gate-level primitives**
  - Boolean logic gates
- **User-defined primitives**
  - Gates and sequential elements defined with truth tables
- **Continuous assignment**
  - Modeling combinational logic with expressions
- **Initial and always blocks**
  - Procedural modeling of behavior

Abstraction levels and synthesis



Activities of Design at Levels of Design



Language Features

- **Nets (wires) for modeling interconnection**
  - Non state-holding
  - Values set continuously
- **Regs for behavioral modeling**
  - Behave exactly like memory for imperative modeling
  - Do not always correspond to memory elements in synthesized netlist
- **Blocking (=) vs. nonblocking (<=) assignment**
  - Blocking behaves like normal “C-like” assignment
  - Nonblocking updates later for modeling synchronous behavior

Language Uses

- **Event-driven simulation**
  - Event queue containing things to do at particular simulated times
  - Evaluate and update events
  - Compiled-code event-driven simulation for speed
- **Logic synthesis**
  - Translating Verilog (structural and behavioral) into netlists
  - Register inference: whether output is always updated
  - Logic optimization for cleaning up the result

Little-used Language Features

- **Switch-level modeling**
  - Much slower than gate or behavioral-level models
  - Insufficient detail for modeling most electrical problems
  - Delicate electrical problems simulated with a SPICE-like differential equation simulator
- **Delays**
  - Simulating circuits with delays does not improve confidence enough
  - Hard to get timing models accurate enough
  - Never sure you've simulated the worst case
  - Static timing analysis has taken its place

Compared to VHDL

- Verilog and VHDL are comparable languages
- **VHDL has a slightly wider scope**
  - System-level modeling
  - Exposes even more discrete-event machinery
- **VHDL is better-behaved**
- **VHDL is harder to simulate quickly**
- **VHDL has fewer built-in facilities for hardware modeling**
- **VHDL is a much more verbose language**

In Conclusion

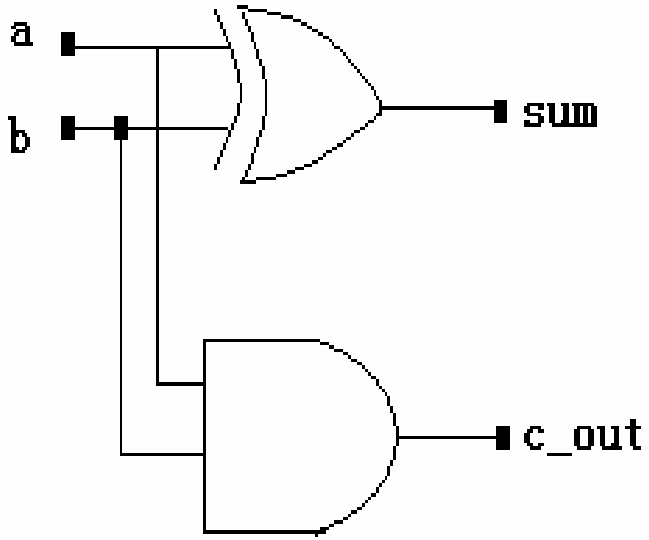
- **Verilog is a deeply flawed language**
  - Nondeterministic
  - Often weird behavior due to discrete-event semantics
  - Vaguely defined synthesis subset
  - Many possible sources of simulation/synthesis mismatch
- **Verilog is widely used because it solves problems**
  - Good simulation speed that continues to improve
  - Designers use a well-behaved subset of the language
  - Makes a reasonable specification language for logic synthesis
  - Logic synthesis: one of the great design automation success stories

4.1 Structure models of combinational logic

- A Verilog model encapsulates a **description of its functionality as a structural or behavioral view of its I/O relationship**
- A structural view could be a netlist of gates or a high-level architectural partition of the circuit
- Structural design is similar to creating a schematic

A structure design : a half adder

```
module Add_half (sum, c_out, a, b);
    input a, b;
    output c_out, sum;
    xor (sum, a, b);
    and (c_out, a, b);
endmodule
```



A schematic consists of

Icons ***(symbols)*** of logic gates

Lines representing wires that connect gates

Labels of relevant signal names at I/O pins and internal nodes

An HDL structural model

- An HDL structural model consists of a list of declarations that specify the inputs and outputs of the unit
- **Structure Modeling**
  - Primitive (Basic Unit):

    Logic Gates ,Switches…
  - UDP (User Defined Primitives)
  - Module

4.1.1 Verilog Primitives and Design encapsulation

- Verilog includes a set of 26 primitives predefined functional models of common combinational logic gates
- **Primitives are the most basic functional objects that can be used to compose a design.**
- Primitives have ports(terminals) that connect to its environment
- **The output port(s) of a primitive must be first in the list of ports ,followed by the primitive's input port(s)**

Verilog Primitives

- **Logic Gates**
- and (Output, Input,...)
- nand (Output, Input,...)
- or (Output, Input,...)
- nor (Output, Input,...)
- xor (Output, Input,...)
- xnor (Output, Input,...)
- **Buffer and Inverter Gates**
- buf (Output,..., Input)
- not (Output,..., Input)
- **Tristate Logic Gates**
- bufif0 (Output, Input, Enable)
- bufif1 (Output, Input, Enable)
- notif0 (Output, Input, Enable)
- notif1 (Output, Input, Enable)
- **MOS Switches**
- nmos (Output, Input, Enable)
- pmos (Output, Input, Enable)
- rnmos (Output, Input, Enable)
- rpmos (Output, Input, Enable)

- **CMOS Switches**
- cmos (Output, Input, NEnable, PEnable)
- rcmos (Output, Input, NEnable, PEnable)
- **Bidirectional Pass Switches**
- tran (Inout1, Inout2)
- rtran (Inout1, Inout2)
- **Bidirectional Pass Switches with Control**
- tranif0 (Inout1, Inout2, Control)
- tranif1 (Inout1, Inout2, Control)
- rtarnif0 (Inout1, Inout2, Control)
- rtranif1 (Inout1, Inout2, Control)
- **Pullup and Pulldown Sources**
- pullup (Output)
- pulldown (Output)

A simulator & primitives

- A simulator uses built-in truth tables to form the outputs of primitives during simulation

bufif0		Enable			
		0	1	X	Z
Data	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

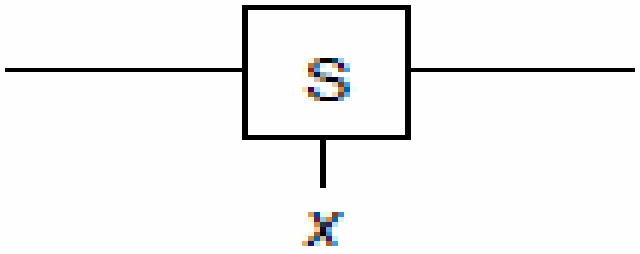
bufif1		Enable			
		0	1	X	Z
Data	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

notfif0		Enable			
		0	1	X	Z
Data	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

notif1		Enable			
		0	1	X	Z
Data	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

Transistors as Switches: Binary Switch

(a) Two states of a switch

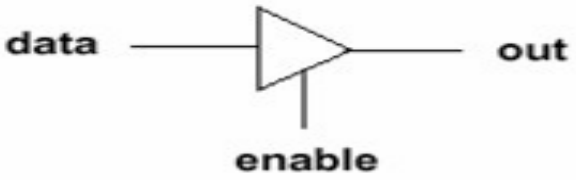


(b) Symbol for a switch

Buffer

基本单元名称	功能
bufif1	条件缓冲器，逻辑 1 使能
bufif0	条件缓冲器，逻辑 0 使能
notif1	条件反相器，逻辑 1 使能
notif0	条件反相器，逻辑 0 使能

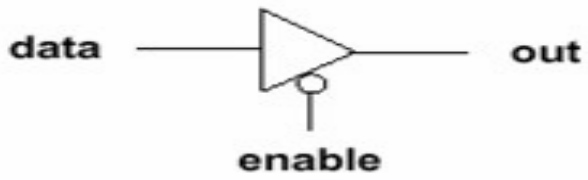
bufif1



bufif1 (out, data, enable)

	bufif1	enable 0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

bufif0



bufif0 (out, data, enable)

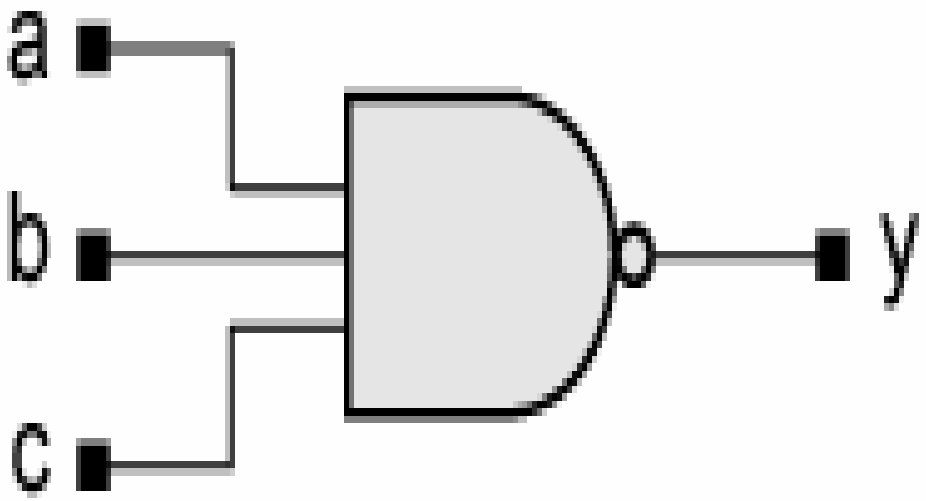
	bufif0	enable 0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

figure 4-2:
a three-input *nand* gate

...

nand (y, a, b, c);

...



Verilog module

- **The functionality of a design and its interface to the environment are encapsulated in a Verilog module.**
- **Verilog module describes**
- (1) its functionality
- (2) its ports to/from the environment
- (3) its timing and other attributes of the design

4.1.2 Verilog Structural Models

A verilog module consists of:

(1) a module name accompanied by its ports

(2) a list of operational modes of the ports (e.g., input)

(3) an optional list of internal wires and/or other variables used by the model, and

(4) a list of interconnected primitives and/or other modules

The text format of a Verilog module

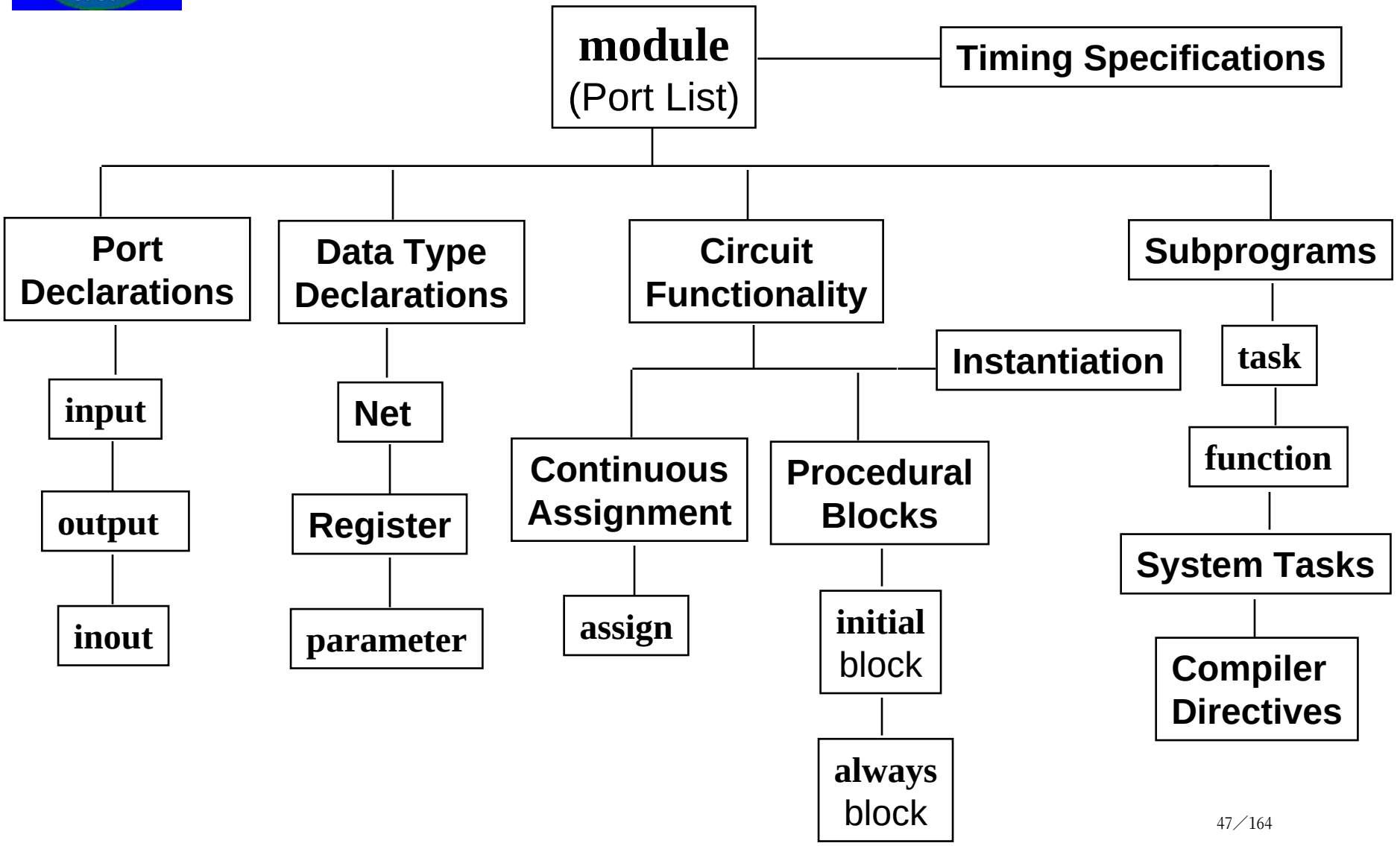
module *module_name (port_list);*

- *port declarations*
- *data type declarations*
- *circuit functionality*
- *timing specifications*

endmodule

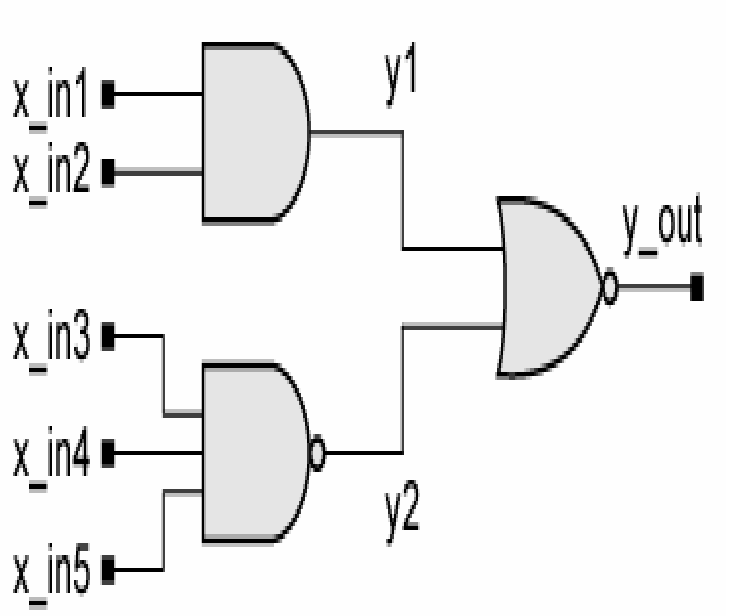
- **CASE**-sensitve
- All **keywords** are lowercase
- **Whitespace** is used for readability.
- **Semicolon** is the statement terminator
- **Single** line comment: ***//***
- **Multi**-line comment: ***/* */***
- **Timing specification** is for simulation

Components of a Verilog Module



A complete structural model of a five-input AOI(And-Or- Invert) circuit

```
module AOI_str
    (y_out,x_in1,x_in2,x_in3,x_in4,
    x_in5);
    output y_out;
    input
    x_in1,x_in2,x_in3,x_in4,x_in5;
    wire y1, y2;
    nor (y_out, y1, y2);
    and (y1, x_in1, x_in2);
    nand (y2, x_in3, x_in4, x_in5);
endmodule
```



4.1.3 Module ports

- **The ports of a module define its interface to the environment in which it is used.**
- The mode of a port determines the direction that information (signal values) may flow through the port.
- **Port's mode:**

(1)unidirectional (***input, output***)

(2)bidirectional (***inout***)

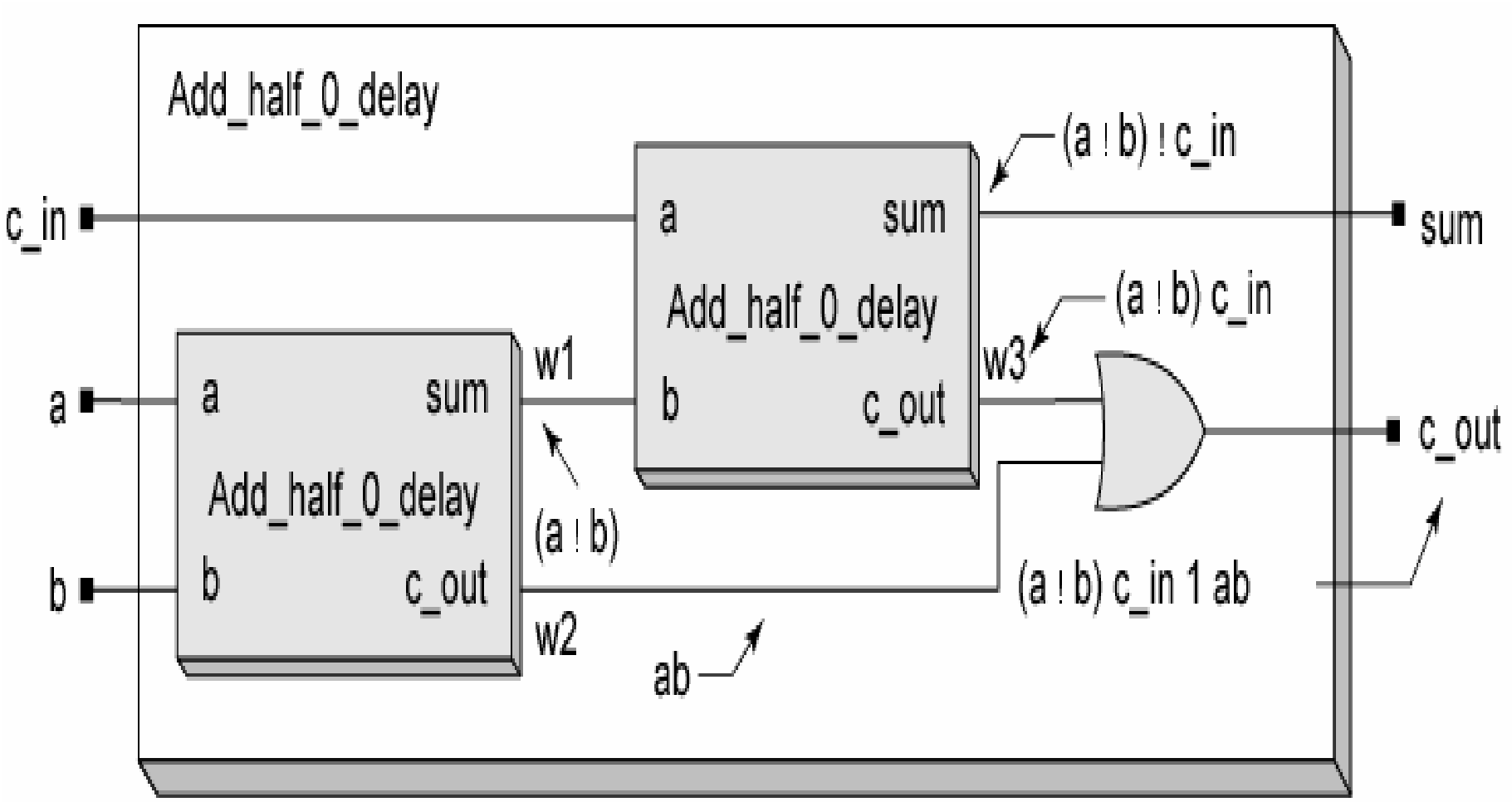
4.1.4 Some language rules

- Verilog is case-sensitive (*a* and *A* is different)
- ***Identifier*** *(name)* is composed of:
  - **case-sensitive sequence of upper and lower case letters**
  - Digits
  - Underscore (_)
  - The $ symbol
  - Not begin with a digit or $
  - Be up to 1024 characters long
- Use a pair of //, or symbol pair /* and */ to imbed comments
- Multiline comments may not be nested

4.1.5 Top-down design and nested modules

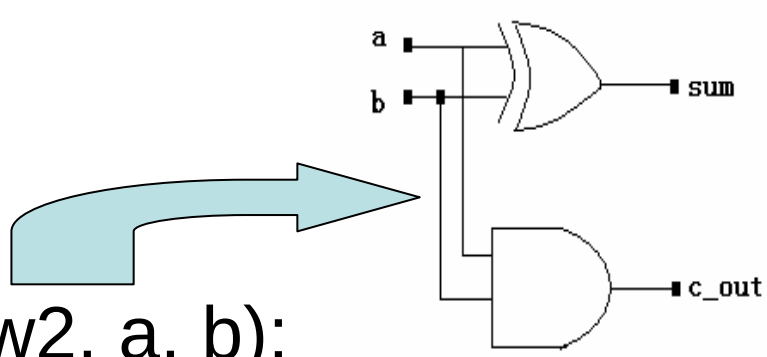
- Top-down design is used in the most modern and sophisticated design methodologies that integrate entire systems on a chip (SoC)
- Nested modules are the Verilog mechanism supporting top-down design

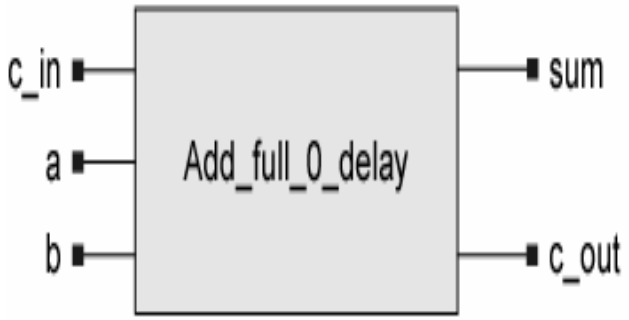
A full adder by combining two half adders (Gate-Level Schematic)



Example 4.1 forming a full adder by combining two half adders

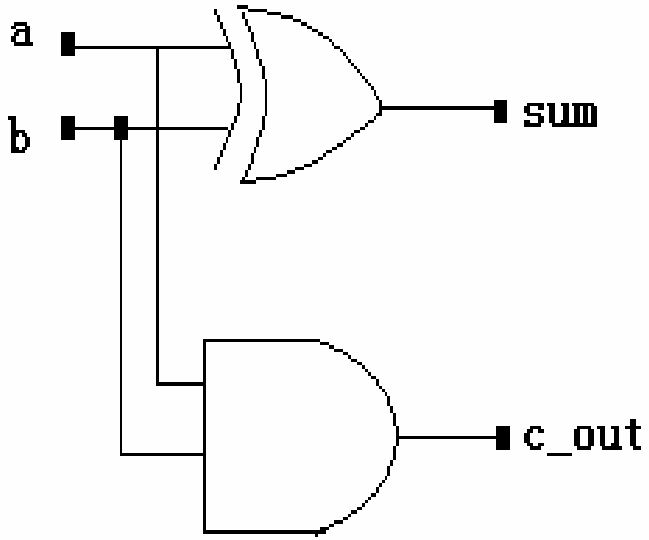
```
module Add_full_0_delay (sum, c_out, a, b, c_in);
    input a, b, c_in;
    output sum, c_out;
    wire w1, w2, w3;
    Add_half_0_delay M1 (w1, w2, a, b);
    Add_half_0_delay M2 (sum, w3, c_in, w1);
    or (c_out, w2, w3);
endmodule
```





A half adder

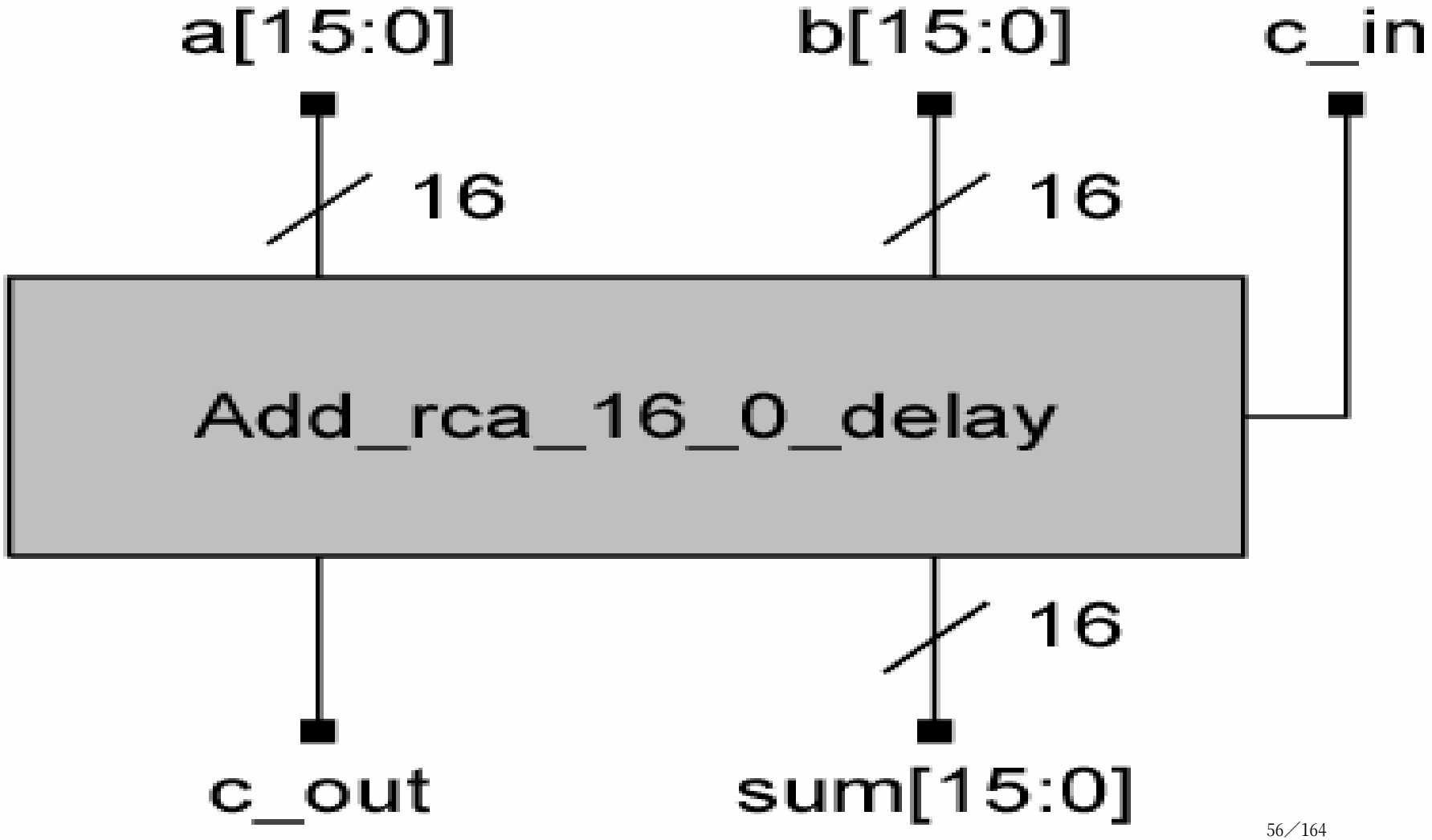
```
module Add_half (sum, c_out, a, b);
    input a, b;
    output c_out, sum;
    xor (sum, a, b);
    and (c_out, a, b);
endmodule
```



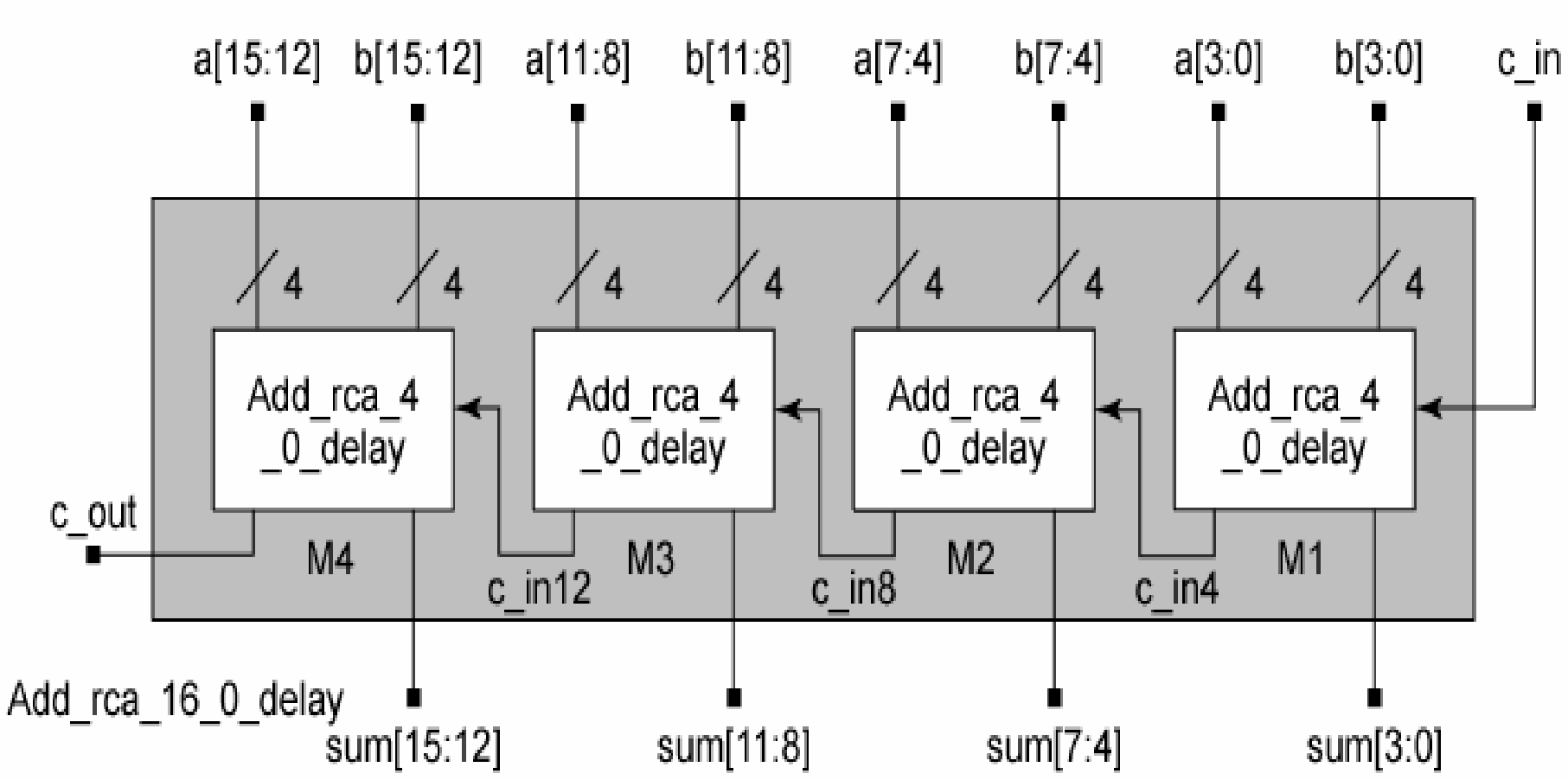
Example 4.2

- A 16-bit ripple-carry adder can be formed by cascading four 4-bit ripple-carry adders in a chain
- In which the carry generated by a unit is passed to the carry input port of its neighbor, beginning with the least significant bit
- Each 4-bit adder is declared as a cascade of full adders

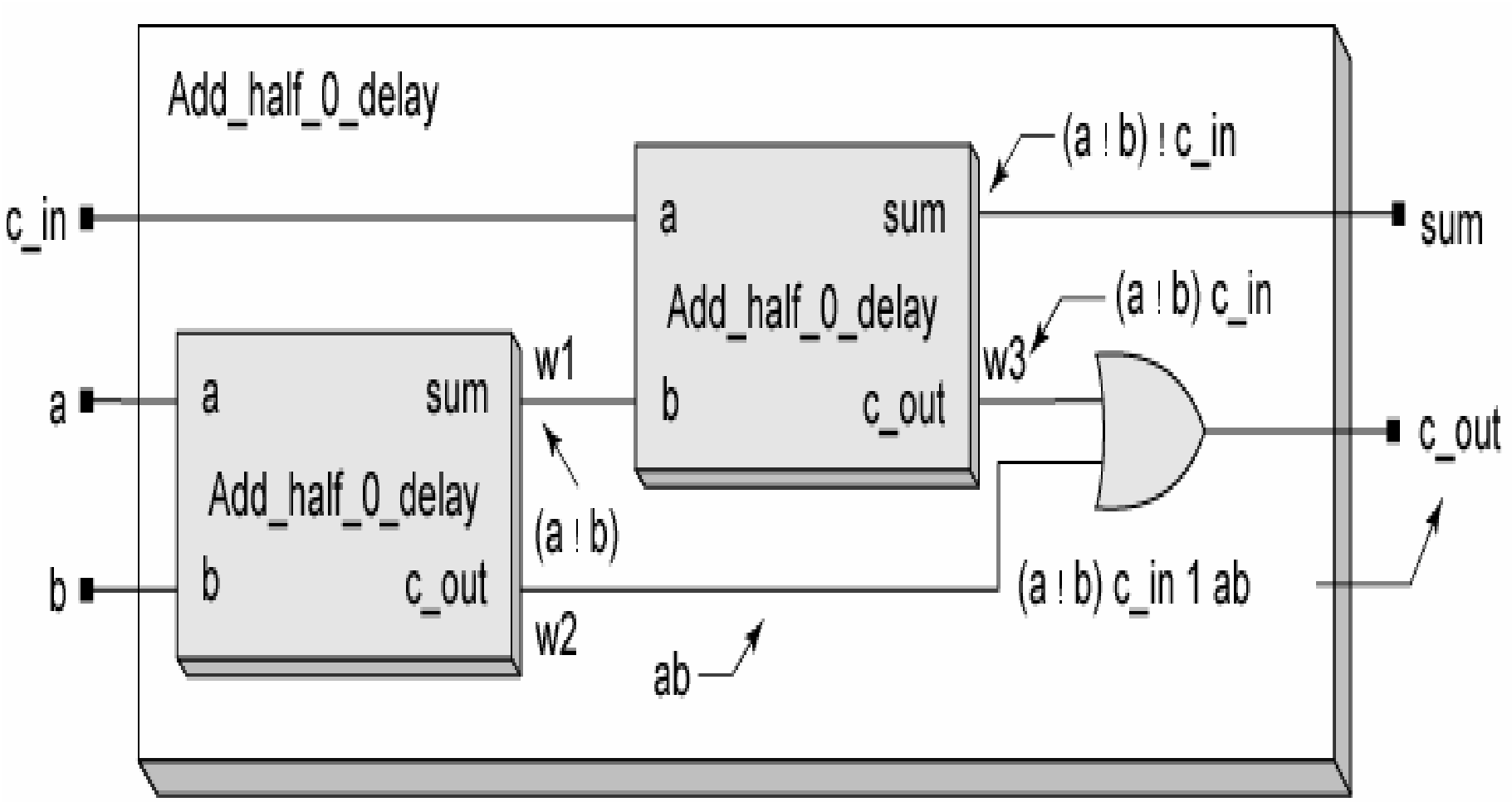
Top-Level Schematic Symbol (Fig 4-7-a)



Decomposition into four 4-bit adders (Fig 4-7-b)

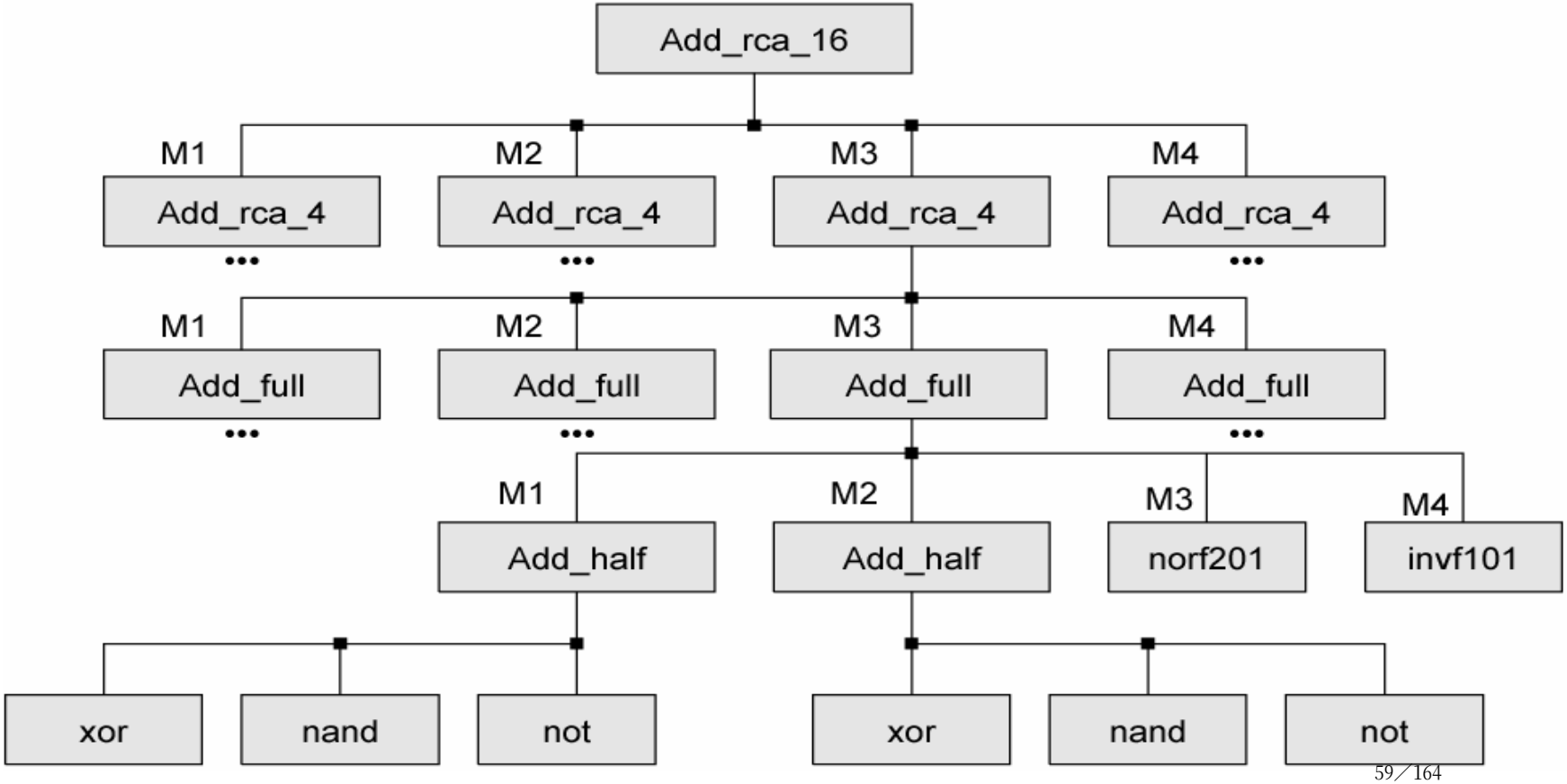


A full adder by combining two half adders



4.1.6 Design hierarchy and source-code organization :

Design hierarchy of a 16-bit ripple-carry adder



Design hierarchy and source-code organization

- All of the modules that compose a design **must be placed in one or more text files** that, when compiled together, completely describe the functionality of the top-level module.
- It does not matter how the modules are distributed across multiple source code files, as long as their individual descriptions reside in a single file.

4.1.7 Vectors in Verilog

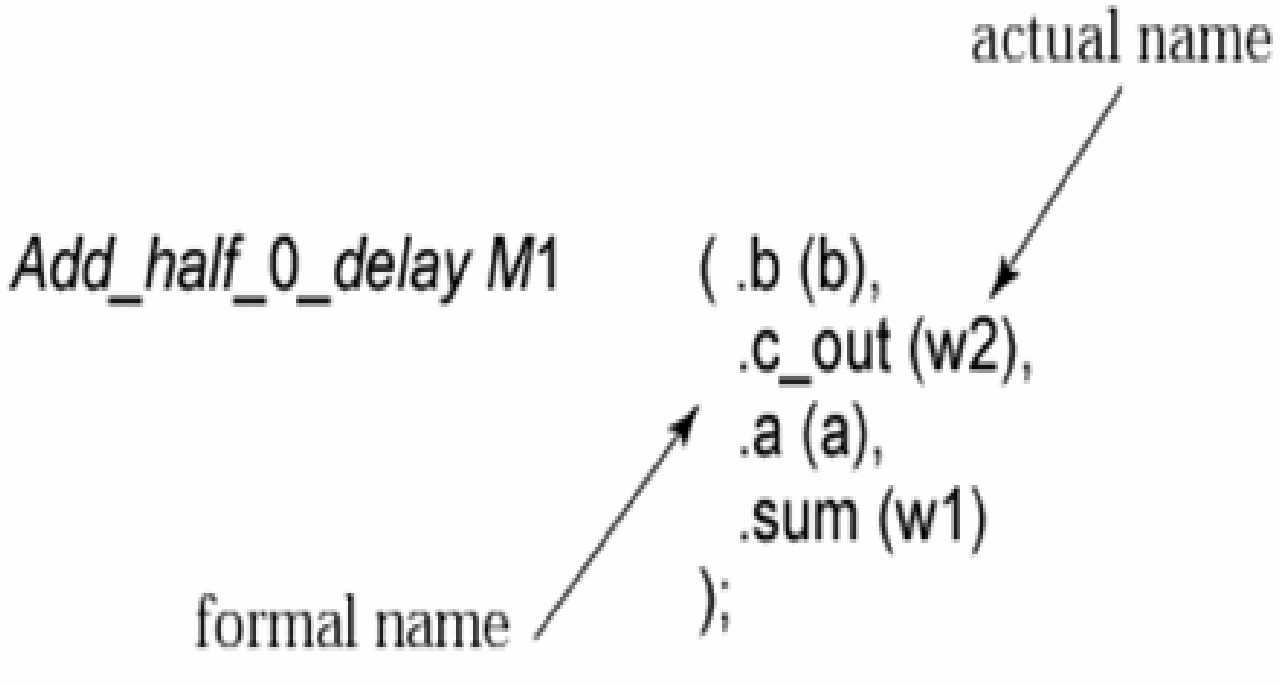
- A vector in Verilog is denoted by square brackets, enclosing a contiguous range of bits
- e.g., sum[3:0] represents four bits from sum
- An expression can be the index of a part-select.
- e.g., if an 8-bit word vect_word[7:0] has a stored value of decimal 4, then
- vect_word[2] has a value of 1;
- vect_word[3:0] has a value of 4;
- vect_word[5:1] has a value of 2.

4.1.8 Structural connectivity

- Wires in Verilog establish connectivity between design objects
- They connect primitives to other primitives and/or modules, and connect modules to other modules and/or primitives
- By themselves, wires have no logic
- Use nets to establish structure connectivity
- **An undeclared identifier is treated by default as a wire**

Example 4.3 The instantiation of Add_half_0_delay in Add_full_0_delay

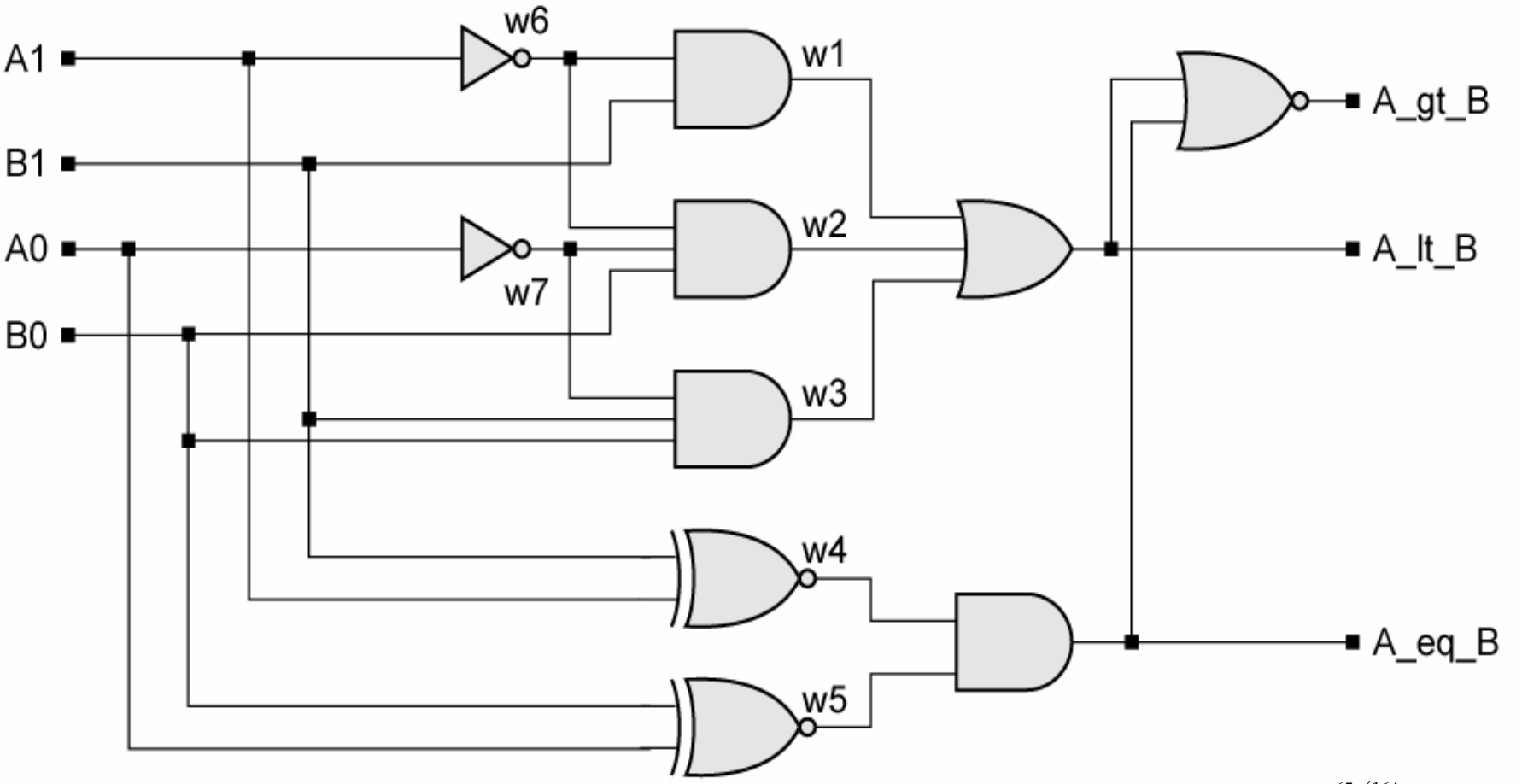
- Using port name association as shown in Figure 4-9



Example 4.4 A 2-bit comparator

```
module compare_2_str (A_gt_B,A_lt_B,A_eq_B,A0,A1,B0,B1);
  output A_gt_B,A_lt_B,A_eq_B;
  input A0,A1,B0,B1;
                    //  A undecleared Indetifiers  default as a wire
  nor (A_gt_B,A_lt_B,A_eq_B);
  or (A_lt_b,w1,w2,w3);
  and (A_eq_B,w4,w5);
  and (w1,w6,B1);
  and (w2,w6,w7,B0);
  and (w3,w7,b0,B1);
  not (w6,A1);
  not (w7,A0);
  xnor (w4,A1,B1);
  xnor (w5,A0,B0);
endmodule
```

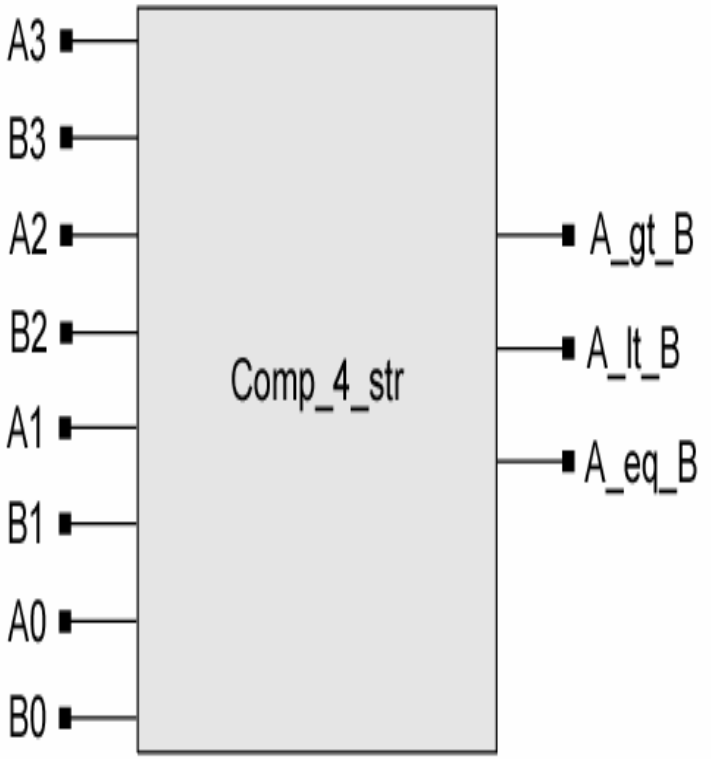
Ex. schematic of a 2-bit binary comparator(Fig 4.10)

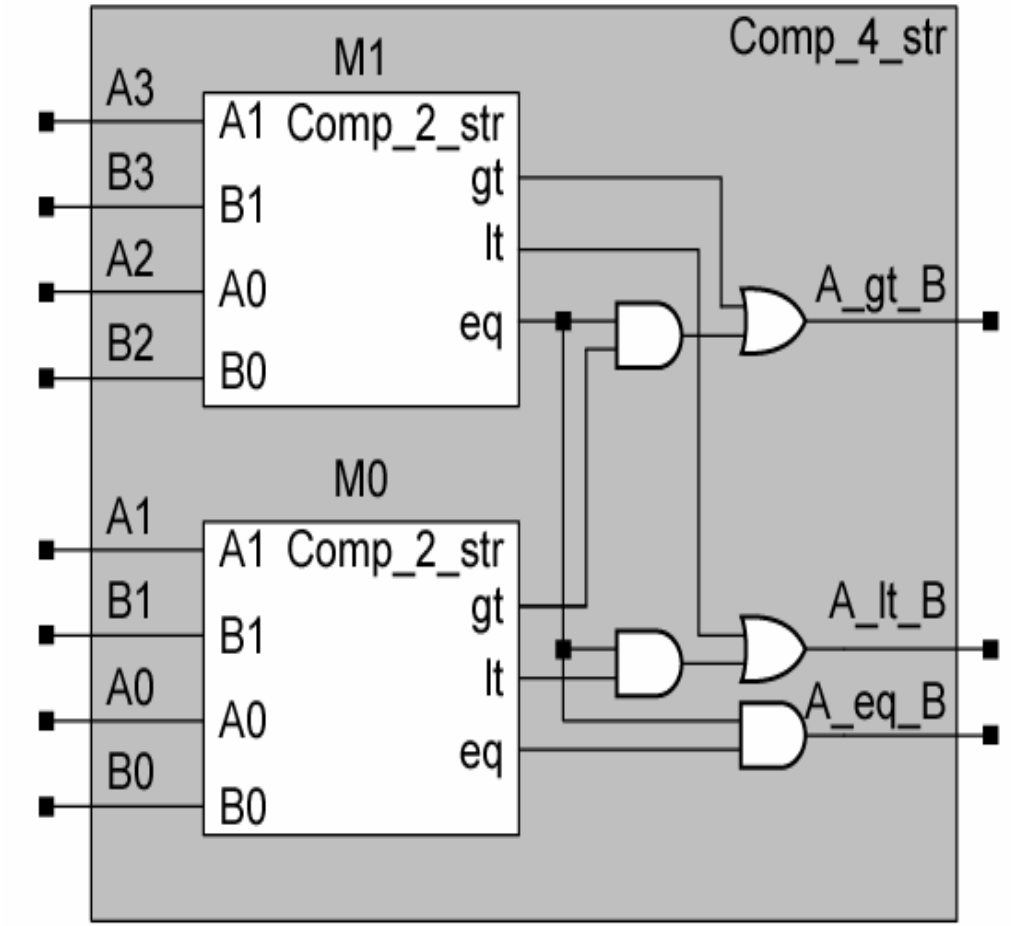


Ex 4.5 A 4-bit comparator

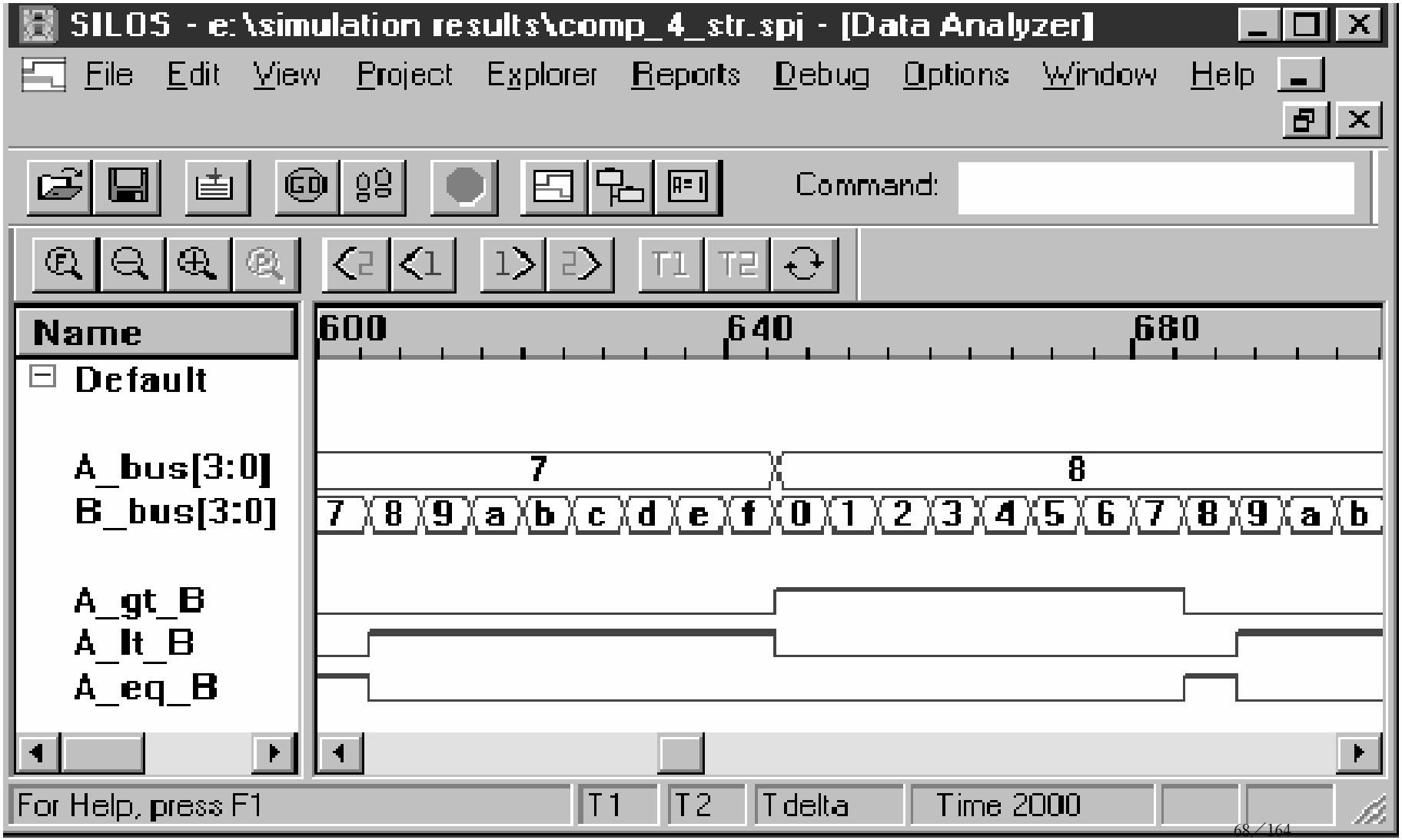
```
module Comp_4_str (A_gt_B,A_lt_B,A_eq_B,A3,A2,A1,A0,B3,B2,B1,B0);
    output A_gt_B,A_lt_B,A_eq_B;
    input A3,A2,A1,A0,B3,C2,B1,B0;
    wire w1, w0;
    Comp_2_str M1(A_gt_B_M1,A_lt_B_M1,A_eq_B_M1,A3,A2,B3,B2);
    Comp_2_str M2 (A_gt_B_M0,A_lt_B_M0,A_eq_B_M0,A1,A0, B1,B0);
    or (A_gt_B,A_gt_B_M1,w1);
    and (w1,A_eq_B_M1,A_gt_B_M0);
    and (A_eq_B,A_eq_B_eq_M1,A_eq_B_M0);
    or (A_lt_B,A_lt_B_M1, w0);
    and (w0,A_eq_B_M1,A_lt_B_M0);
endmodule
```

Ex.4-5 block diagram and hierarchical structure





Simulation result



Module instantiations

```
module ANDOR (input i1, i2, i3, i4, output y);
    assign y = (i1 & i2) | (i3 & i4);
endmodule
//
module MultiplexerE (input a, b, s, output w);
    wire s_bar;
    not U1 (s_bar, s);
ANDOR U2 (a, s_bar, s, b, w);
endmodule
```

4.2 Logic System, Design Verification, and Test Methodology

- Verify models to assure their functionality conforms to the specification for the design
- Two methods of verification:

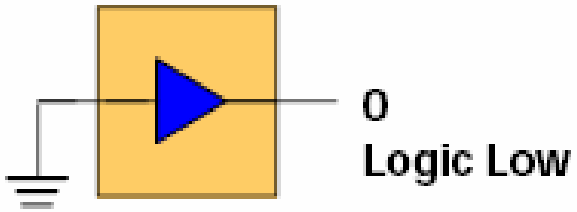
  logic simulation and formal verification
- **Logic simulation**: apply stimulus and monitor circuit's simulated behavior
- **Formal verification** :uses elaborate mathematical proofs to verify a circuit's functionality without having to apply stimulus patterns

4.2.1 Four-Value Logic and Signal Resolution in Verilog

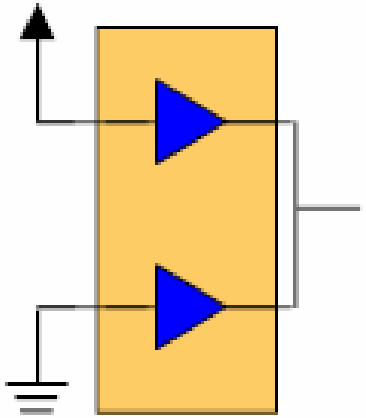
- Verilog uses a four-valued system: 0,1,*x*,*z*
- *The value **x** represents a condition of ambiguity in which the simulator cannot determine whether the value of the signal is 0 or 1*
- The logic value z denotes a three-state condition in which a wire is disconnected from its driver

Four-Value Logic

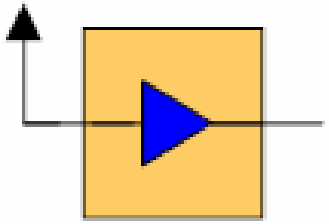
4-value logic system in Verilog



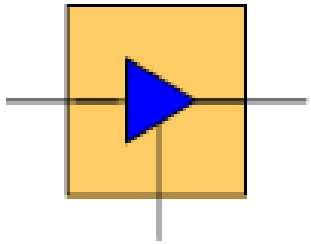
0
Logic Low



x
Unknown



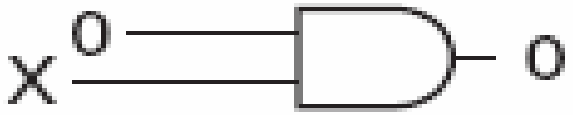
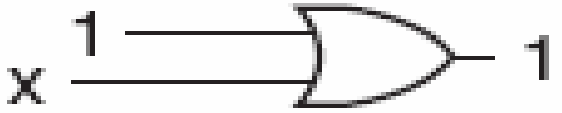
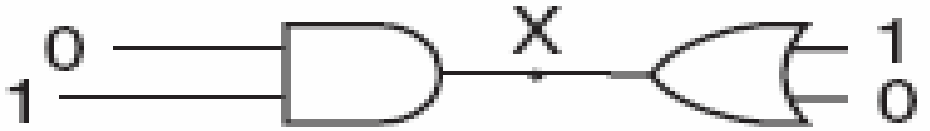
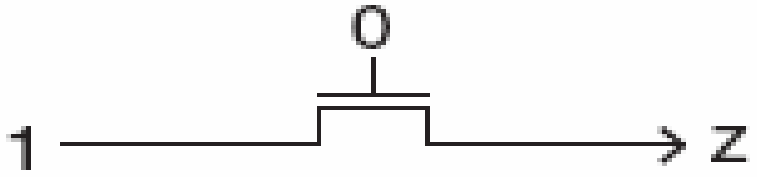
1
Logic High



z
High Impedance

0

Logic Values and Examples

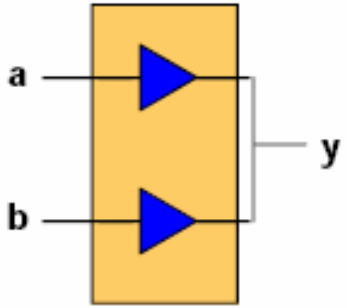
0 :	
1 :	
X or x:	
Z or z :	

Major Data Type Class

- **Nets**
  - Physical connection between devices
- **Registers**
  - Represent abstract storage elements
- **Parameters**
  - Configure module instances

Type of Nets

- The default type of net is ***wire***.
- Logic conflict resolution with net data types



Wire/Tri

a \ b	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Wand/Triand

a \ b	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Wor/Trior

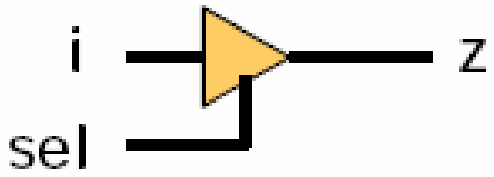
a \ b	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Type of Nets (…,wor,trior,wand…)

Net Types	Functionality
wrie,tri	For standard connection wires (default)
wor,trior	For multiple drivers that are Wired-OR
wand,trand	For multiple drivers that are Wired-AND
trireg	For nets with capacitive storage
tri0	For nets with weak pull down device
tri1	For nets with weak pull up device
supply0	Ground net
supply1	Power net

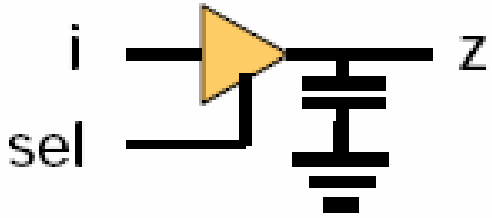
Type of Nets (…,wor,trior,wand…)

wire z:



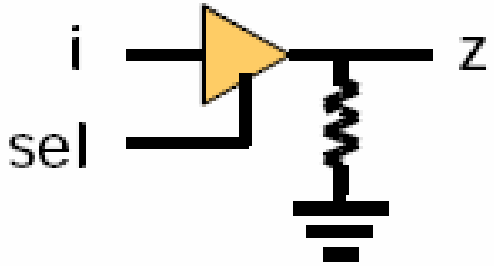
If sel=1, z=i;
If sel=0, z=high impedance;

trireg z:



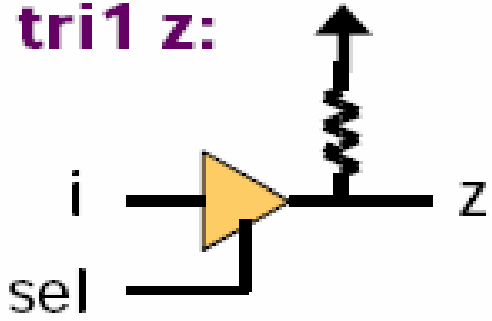
If sel=1, z=i;
If sel=0, z stores its last value;

tri0 z:



If sel=1, z=i;
If sel=0, z=0;

tri1 z:



If sel=1, z=i;
If sel=0, z=1;

Fig 4.14 The waveforms of a simulator

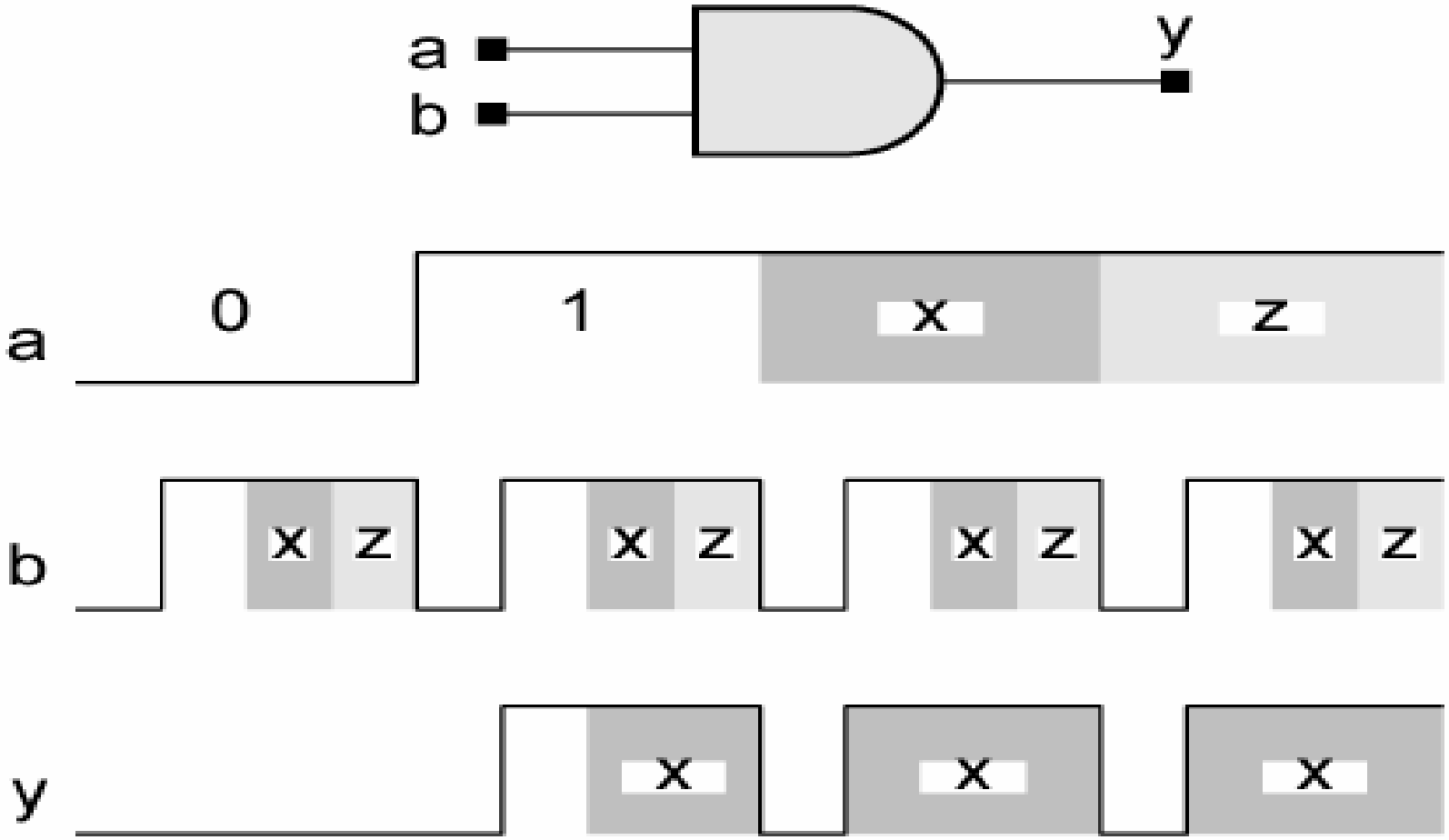
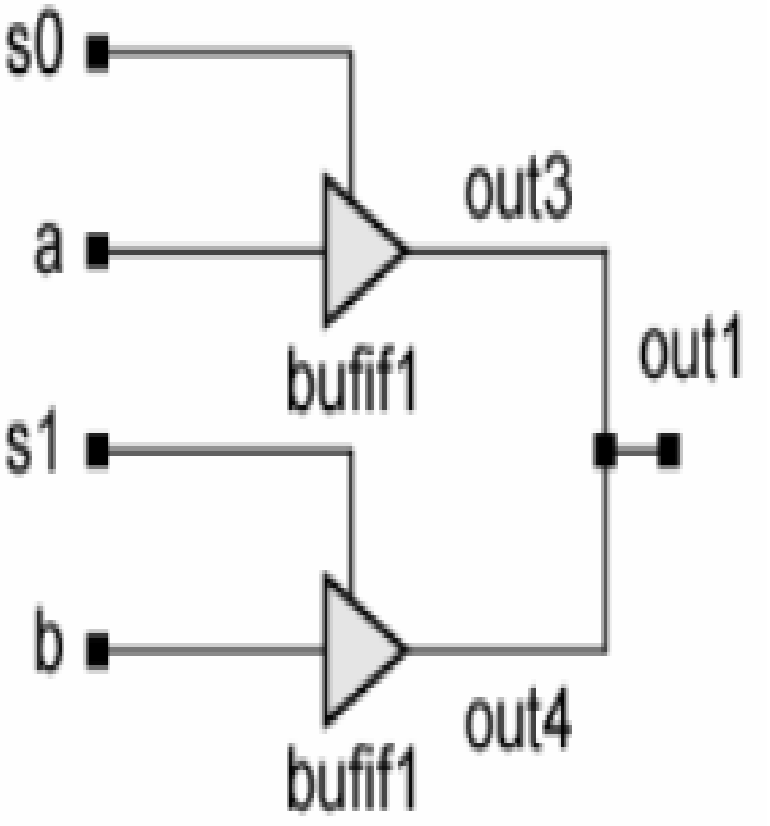


Fig 4.15(1) a Verilog simulator resolves multiple drivers on a net



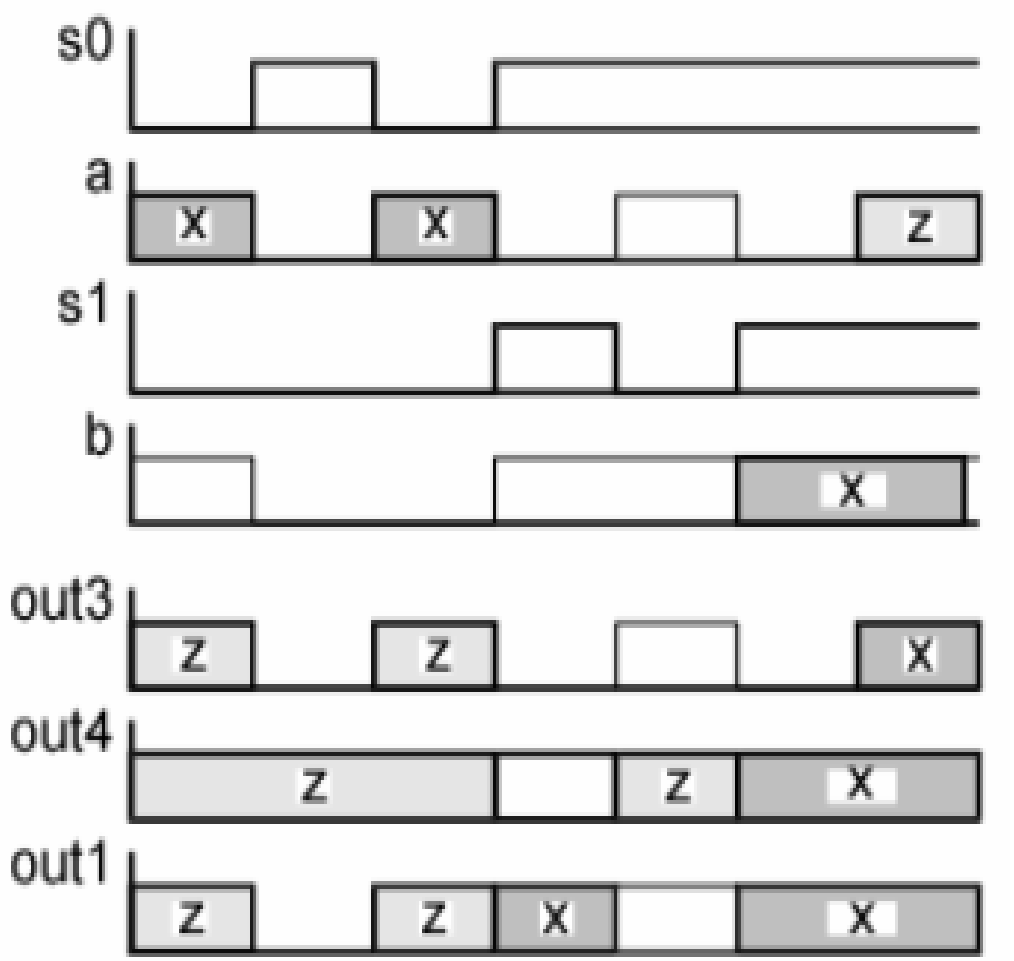
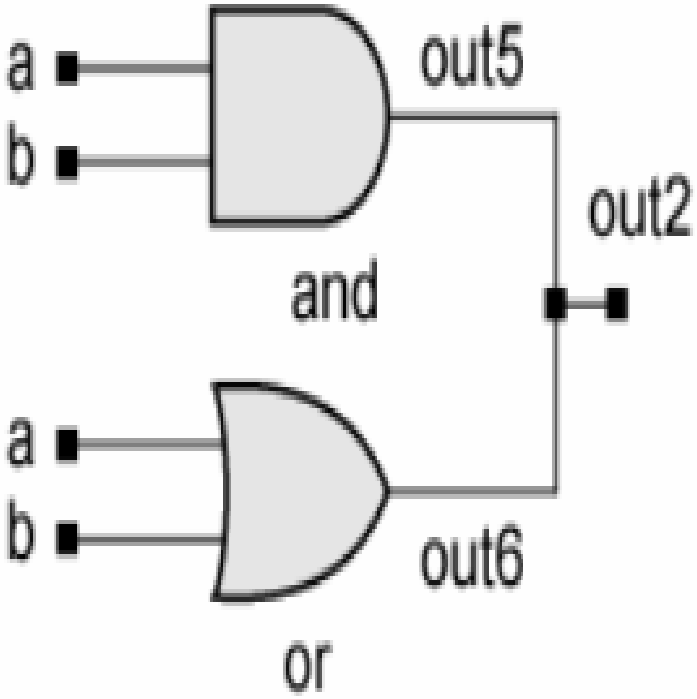
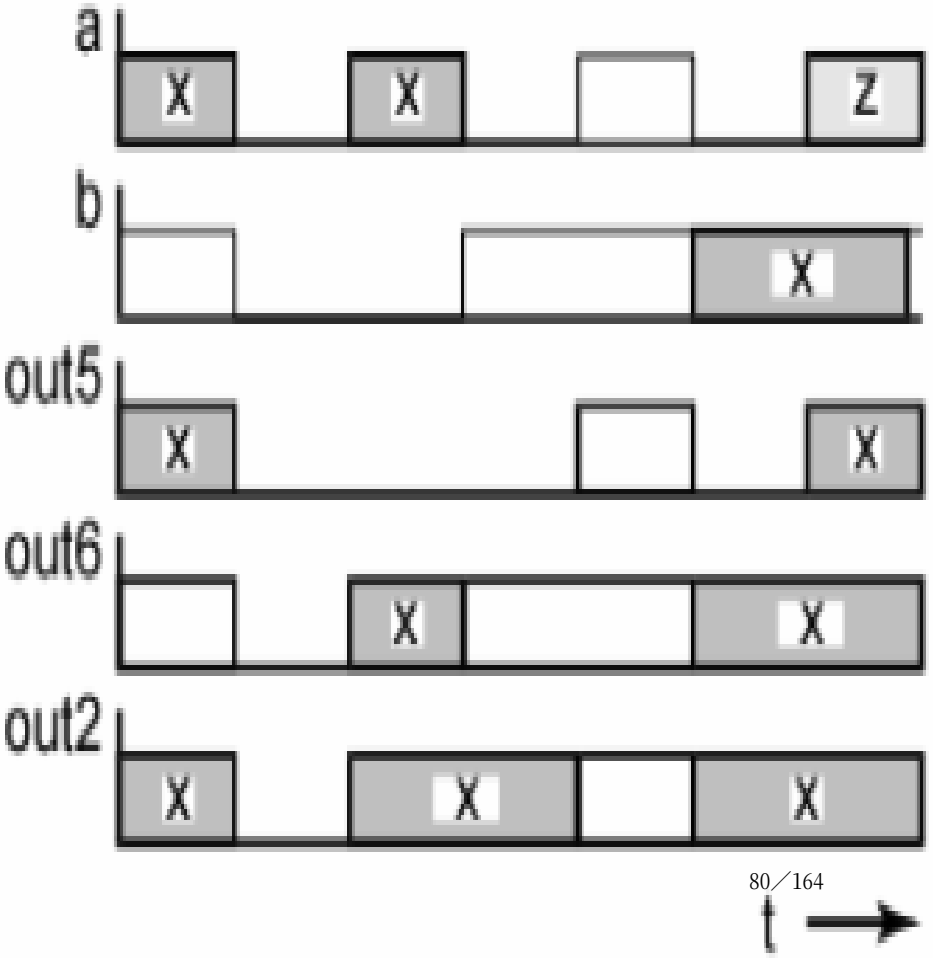


Fig 4.15(2) a Verilog simulator resolves multiple drivers on a net

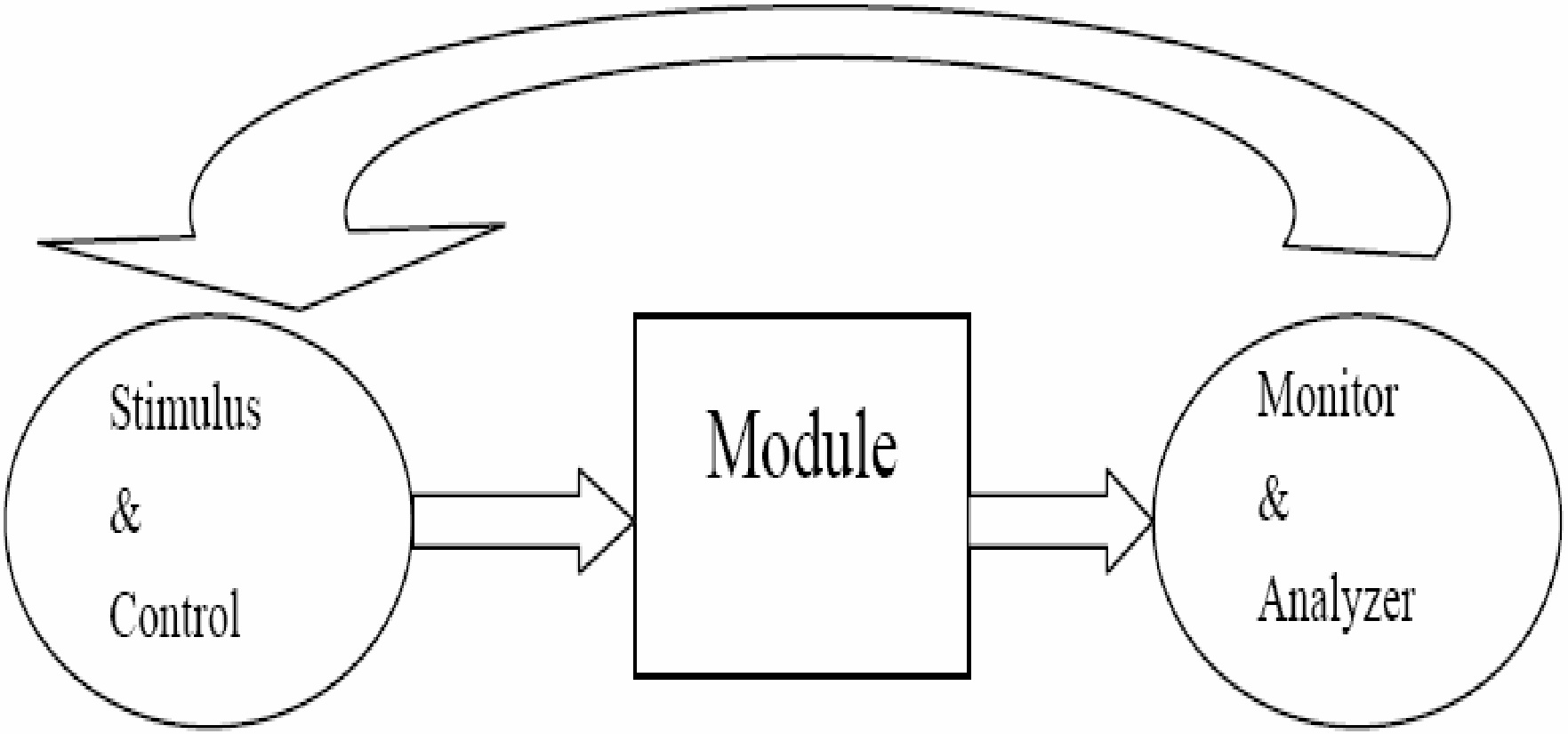




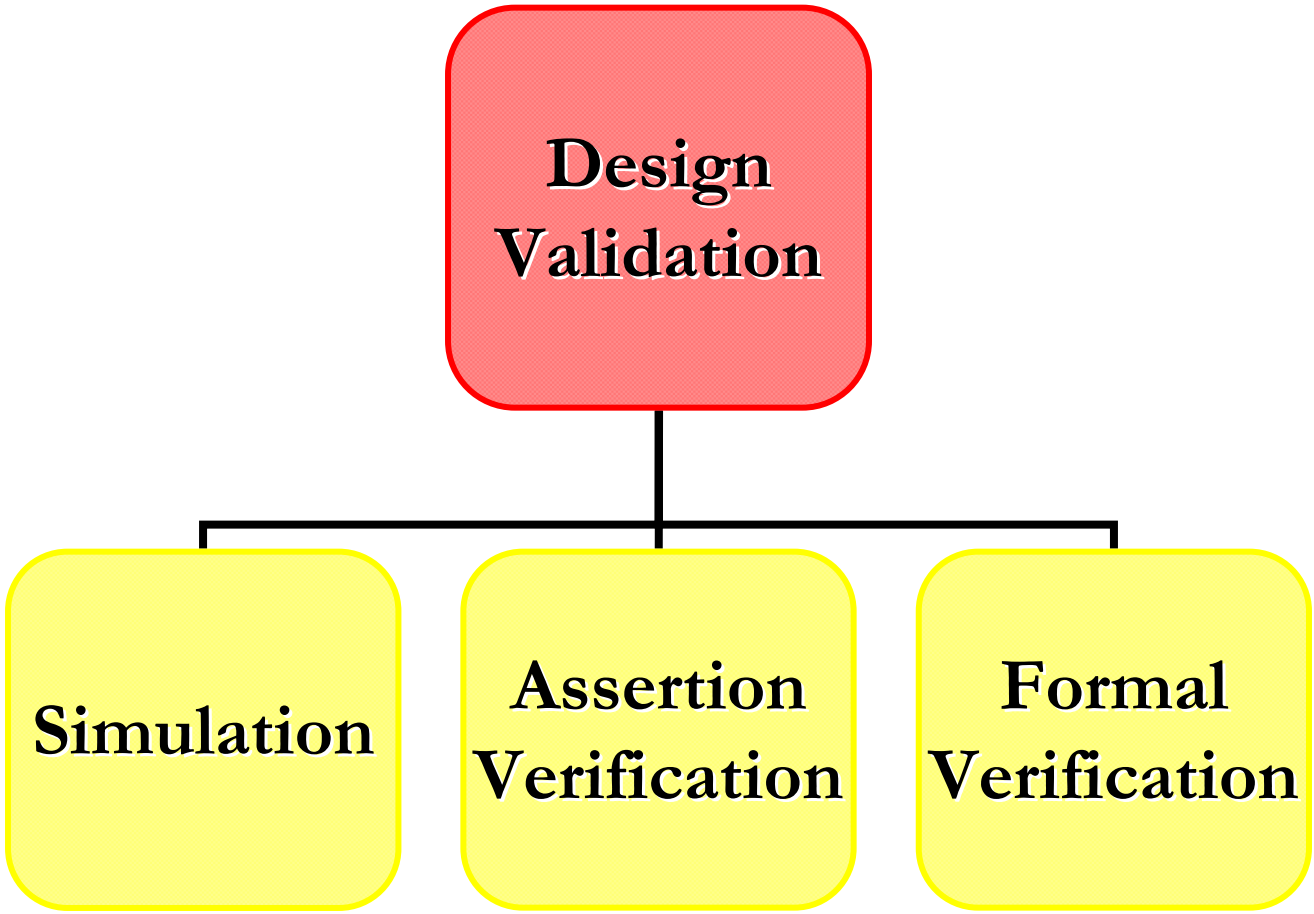
4.2.2 Test Methodology

- A large **circuit must be tested and verified** systematically to ensure that all of its logic has been exercised and found to be functionally correct.
- A haphazard test methodology makes debugging very difficult, and **risks enormous loss should a product fail as a result of an untested area of logic**.
- In practice, design teams write an elaborate test plan that specifies the features that will be tested and the process by which they will be tested.

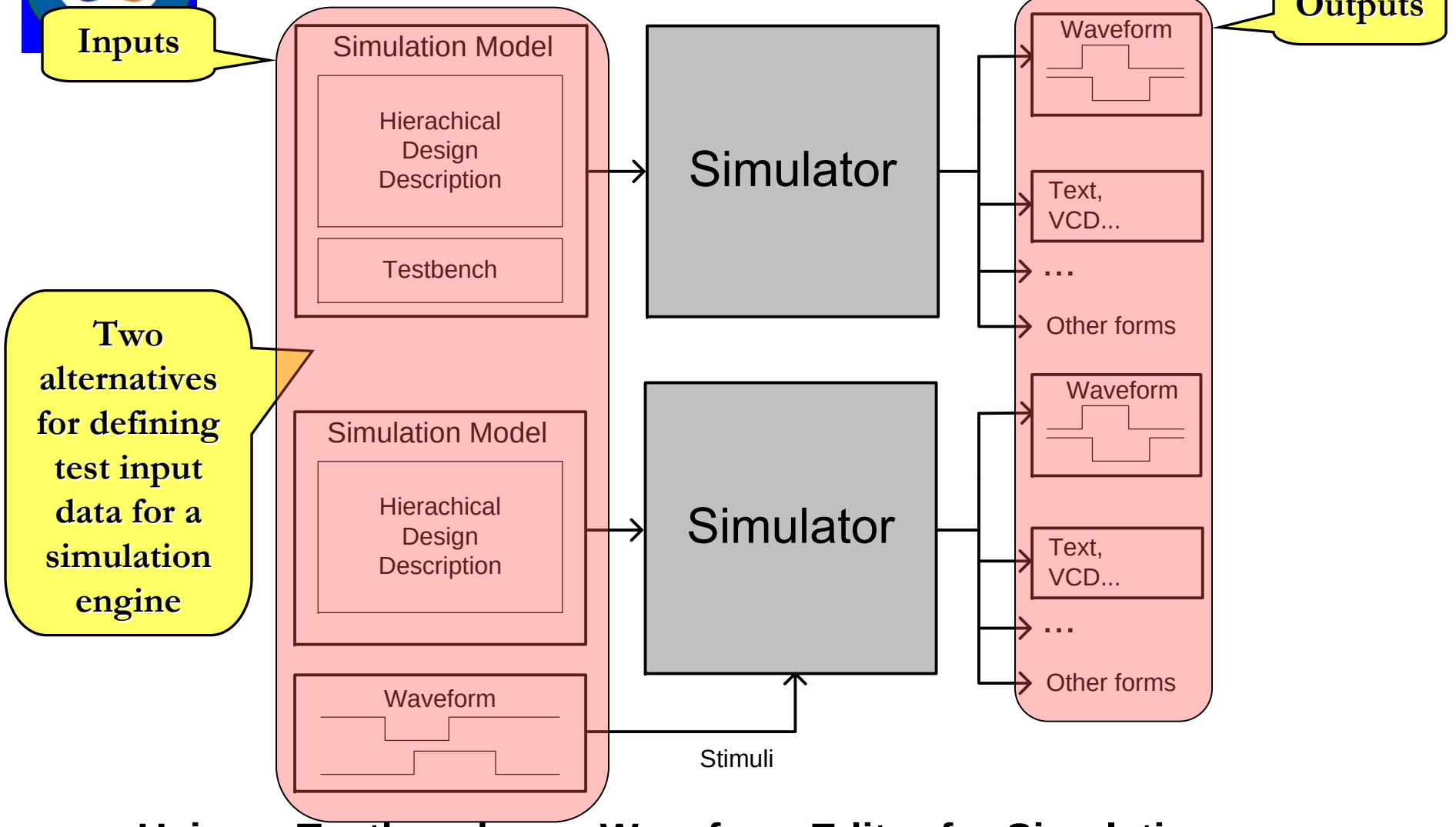
What is Verification?



Design Validation



Simulation



- **Using a Testbench or a Waveform Editor for Simulation**

Simulation

Testbench for the Counter Circuit

```
`timescale 1 ns / 100 ps
module Chap1CounterTester ();
  reg Clk=0, Reset=0;
  wire [3:0] Count;
  initial begin
    Reset = 0; #5 Reset = 1; #115 Reset = 0;
    #760 $stop;
  end
  always #26.5 Clk = ~ Clk;
  Chap1Counter U1 (Clk, Reset, Count);
endmodule
```

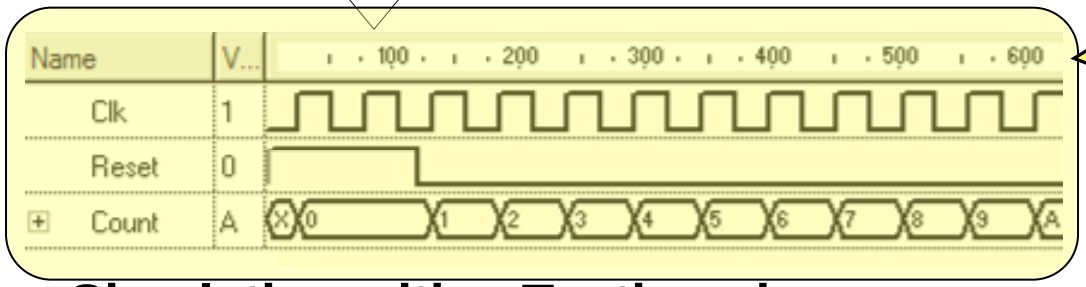
```
module Chap1Counter (Clk, Reset, Count);
  input Clk, Reset;
  output [3:0] Count;
  reg [3:0] Count;
  always @(posedge Clk) begin
    if (Reset) Count = 0;
    else Count = Count + 1;
  end
endmodule
```

Verilog Code of a Counter Circuit

Simulator

Testbench

Design to Simulate



The simulation results in form of a waveform

- **Verilog Simulation with a Testbench**

Simulation

The testbench instantiates the design under test, and as part of the code of the testbench it applies test data to the instantiated circuit.

```
`timescale 1 ns / 100 ps
module Chap1CounterTester ();
  reg Clk=0, Reset=0;
  wire [3:0] Count;
  initial begin
    Reset = 0; #5 Reset = 1; #115 Reset = 0;
    #760 $stop;
  end
  always #26.5 Clk = ~ Clk;
  Chap1Counter U1 (Clk, Reset, Count);
endmodule
```

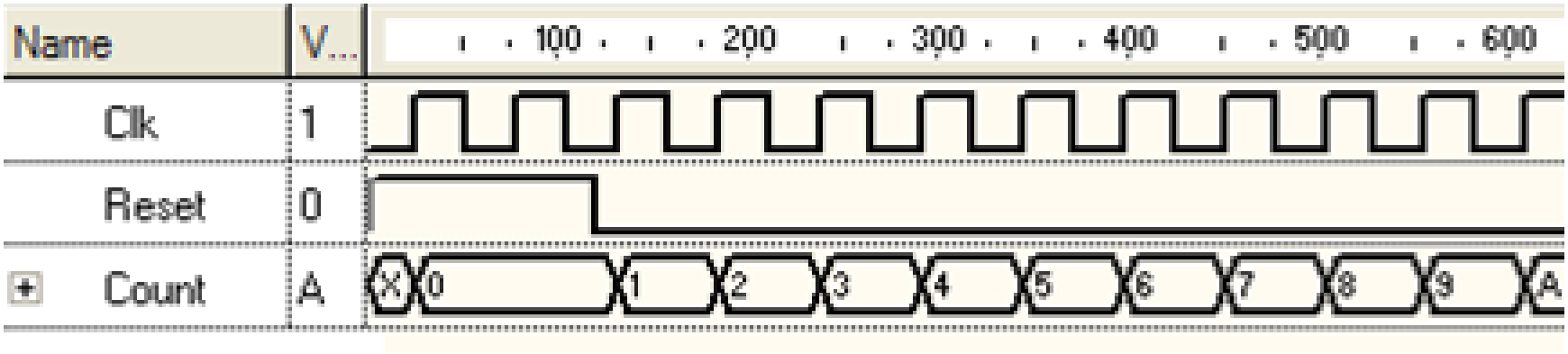
```
module Chap1Counter (Clk, Reset, Count);
  input Clk, Reset;
  output [3:0] Count;
  reg [3:0] Count;
  always @(posedge Clk) begin
    if (Reset) Count = 0;
    else Count = Count + 1;
  end
endmodule
```

- **Verilog Simulation with a Testbench (Continued)**

Simulation

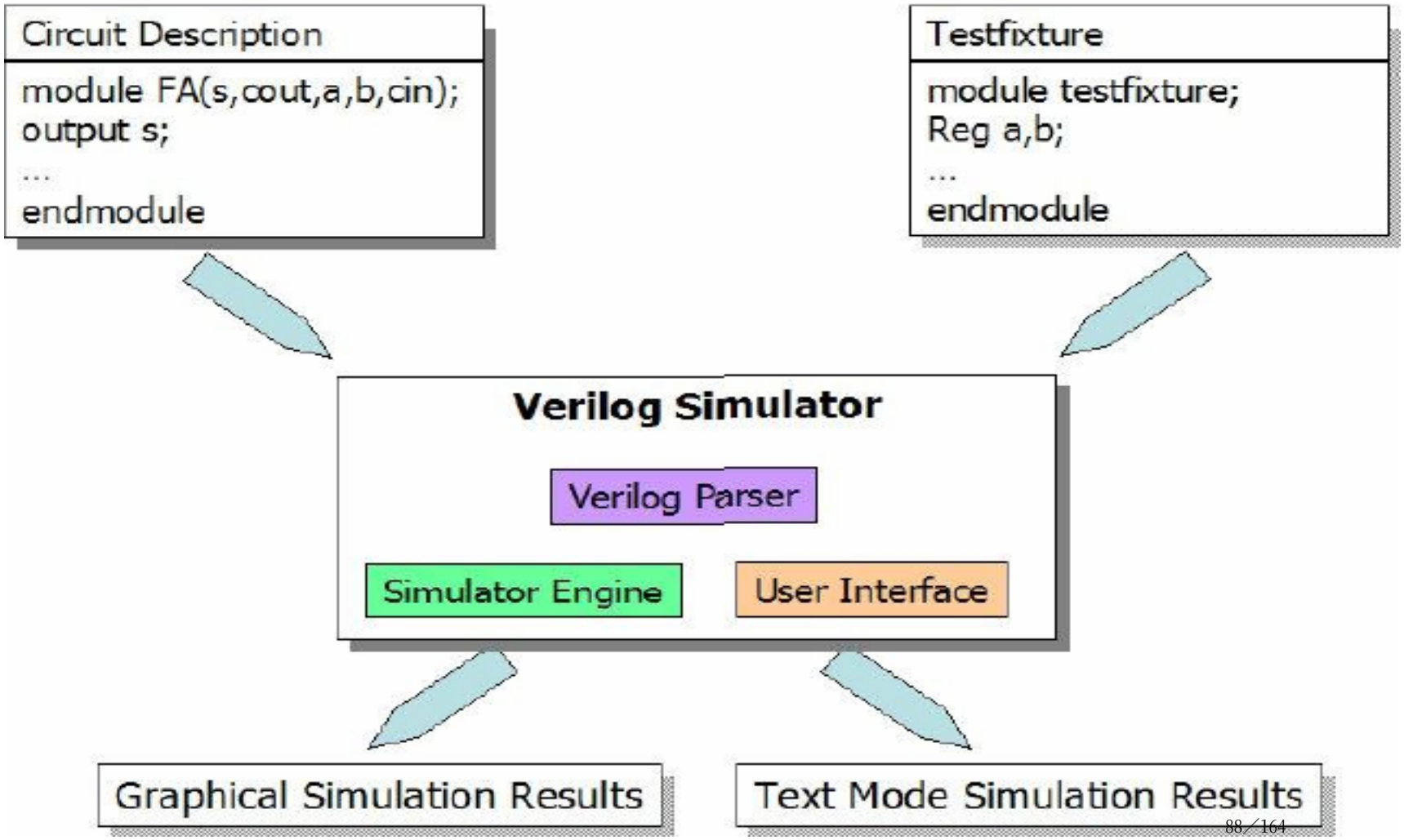
Simulator

Validates the functionality of the counter circuit being tested, Regardless of clock frequency

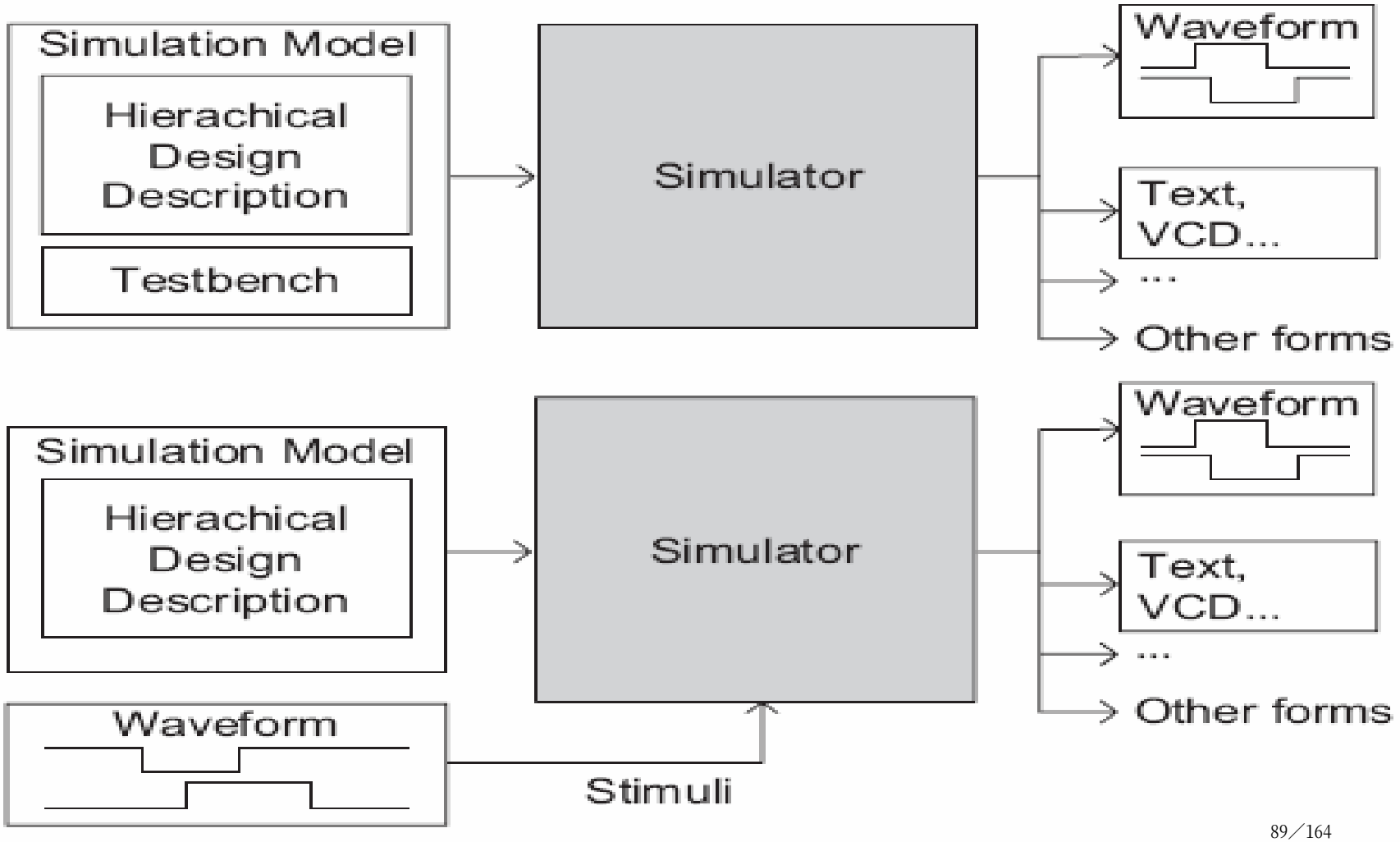


- **Verilog Simulation with a Testbench (Continued)**

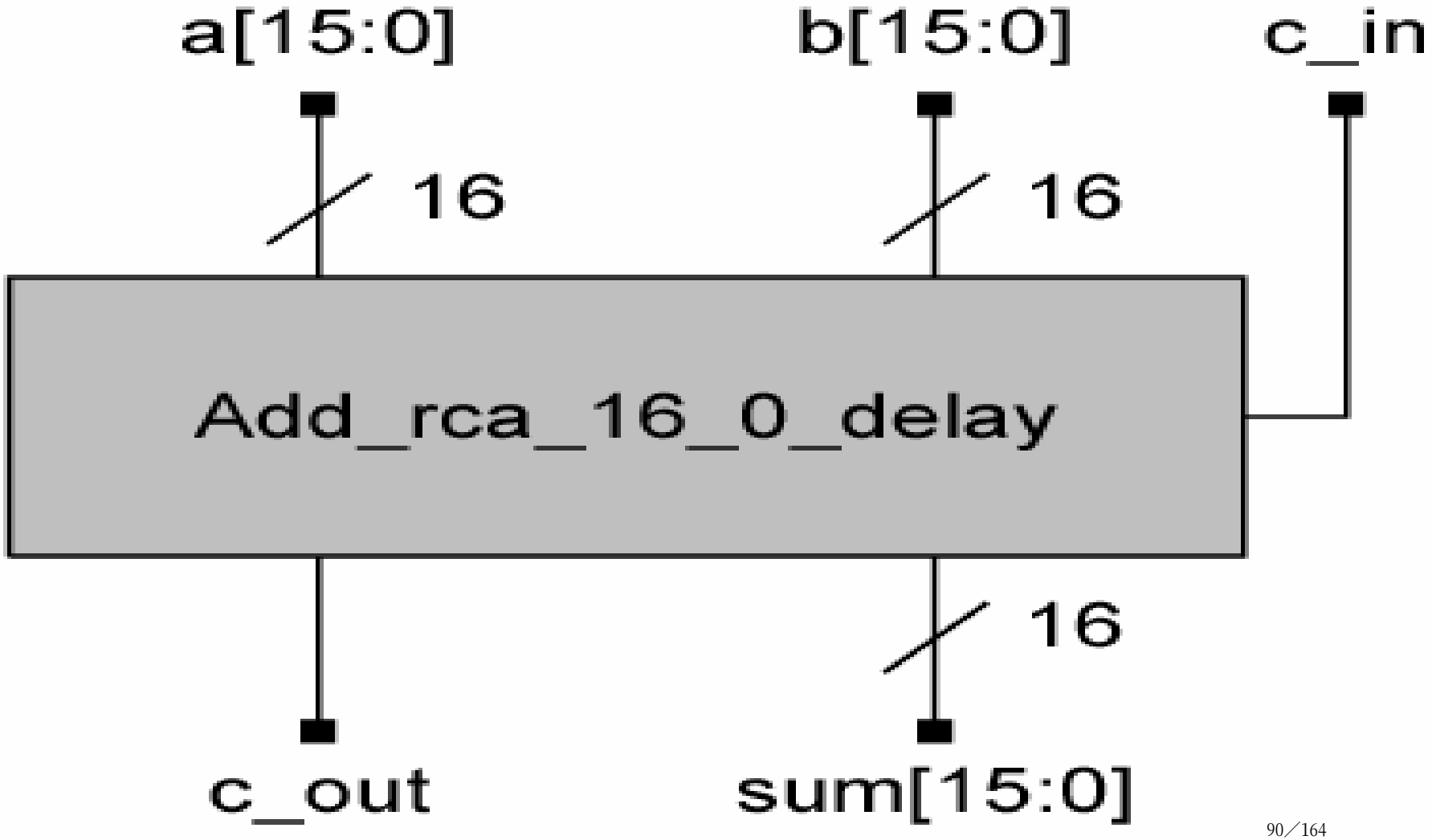
Verilog Simulation



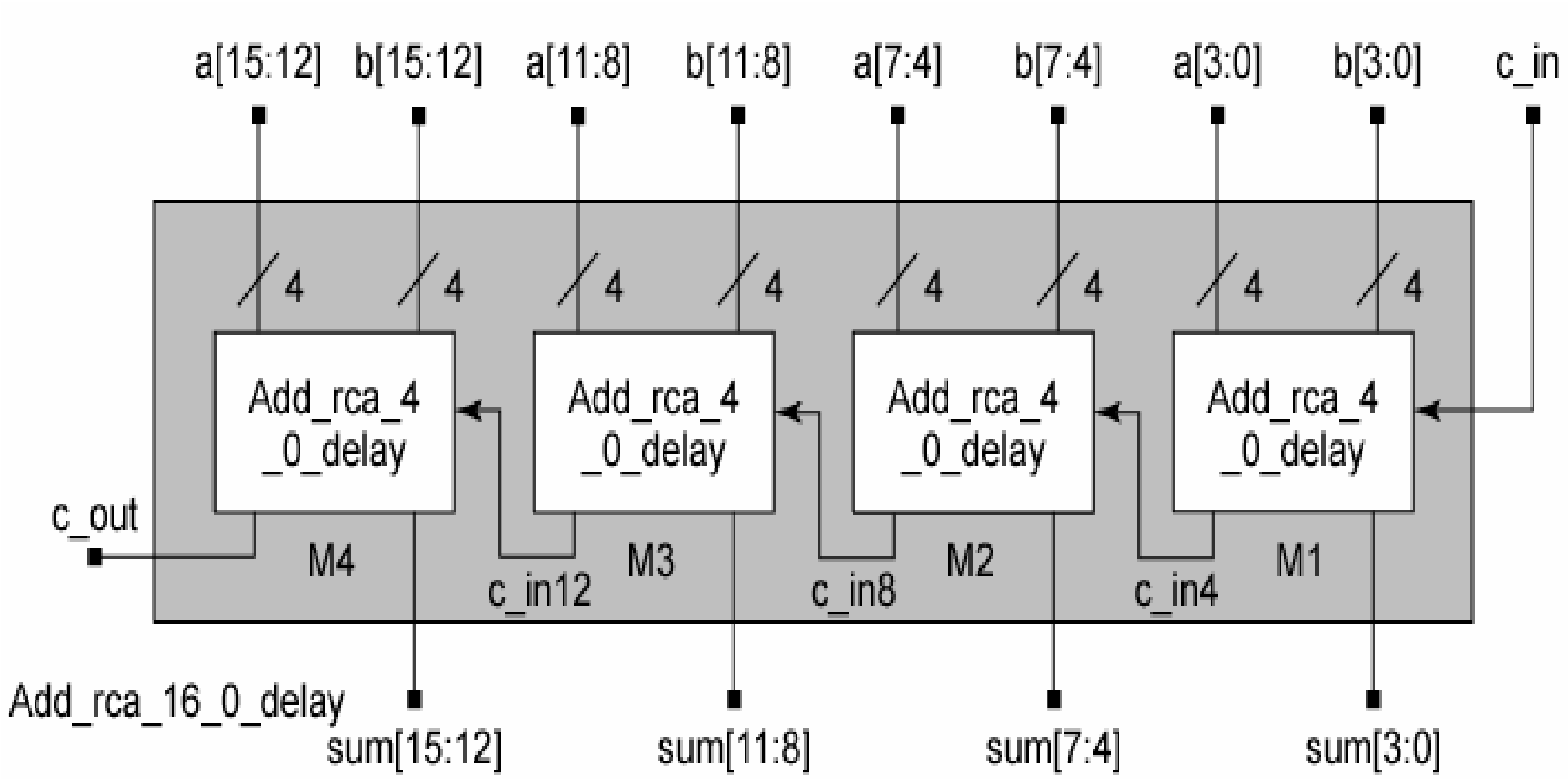
Using a Testbench or a Waveform Editor for Simulation



Top-Level Schematic Symbol (Fig 4-7-a)



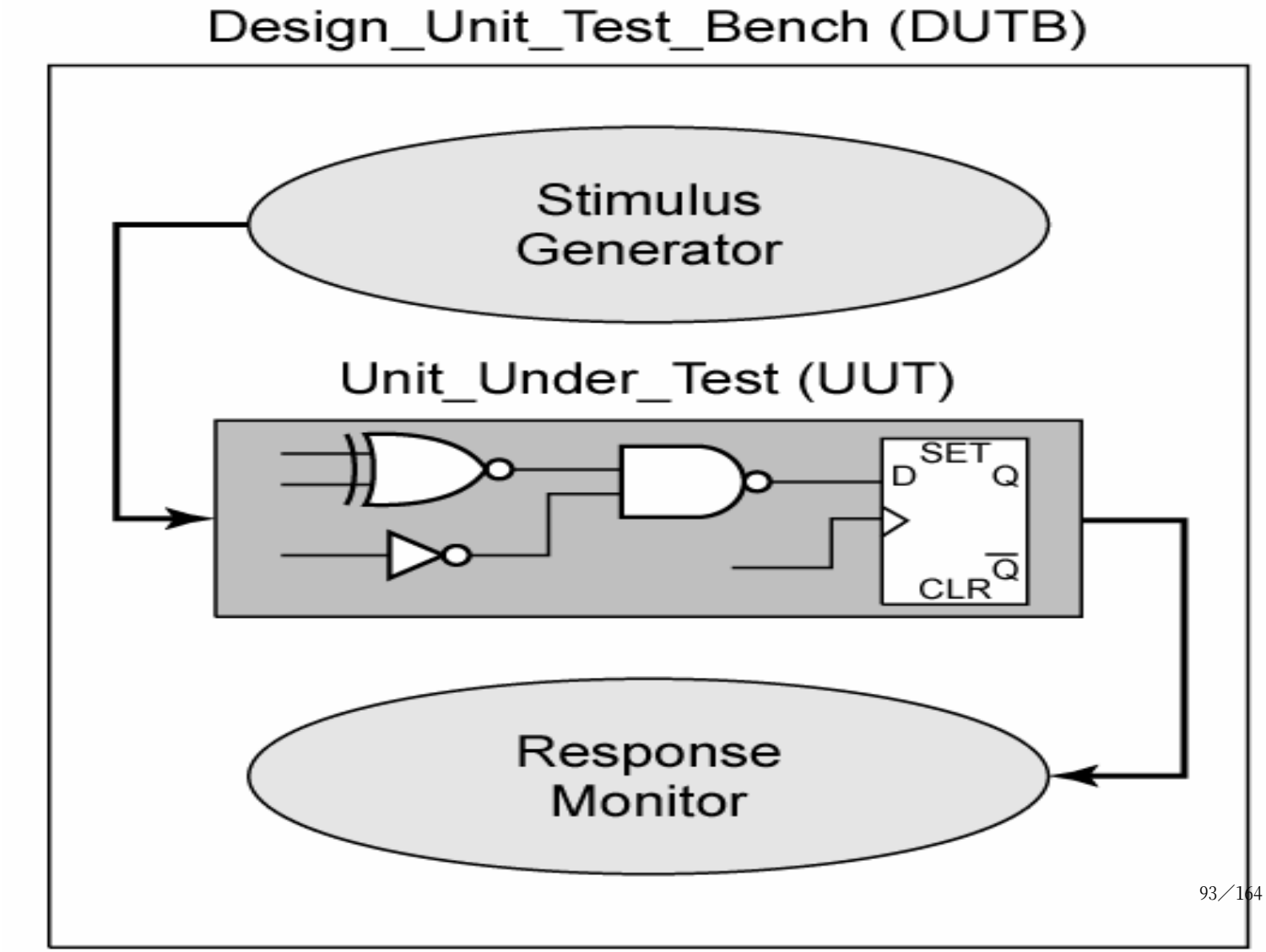
Example 4.6
The 16-bit adders



In the example 4.6

- The partition of a 16-bit ripple-carry adder into four bit-slices of 4 bits
- Applying 16-bit stimulus patterns requires 2^{33} input patterns
- First to verify the half-adder and full-adder each work correctly
- Then the 4-bit slice unit can be verified by applying 2^{9} patterns

Fig 4.16 A testbench's basic organization



Example 4.7 A testbench for verifying Add_half_0_delay

```
module t_add_half();
   wire sum, c_out;
   reg a, b;
   add_half_0_delay M1 (sum, c_out, a, b);      // UUT
   initial begin                                // time out
     #100 $finish;                               //system task
   end
   initial   begin                              //stimulus
     #10 a=0;b=0;
     #10 b=1;
     #10 a=1;
     #10 b=0;
   end
endmodule
```

System Functions

- **$finish**
- $finish[(N)]; {*N* is 0, 1, or 2}
- Causes the simulator to exit, passing control back to the operating system.

Delay description

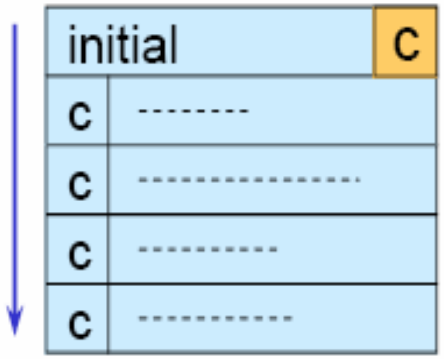
- Delays may be specified for instances of UDPs and gates, for continuous assignments, and for nets. These delays model the propagation delay of components and connections in a netlist
- # delaytime
- # (d1,d2)
- # (d1,d2,d3)
- d1:rise; d2:fall; d3:z

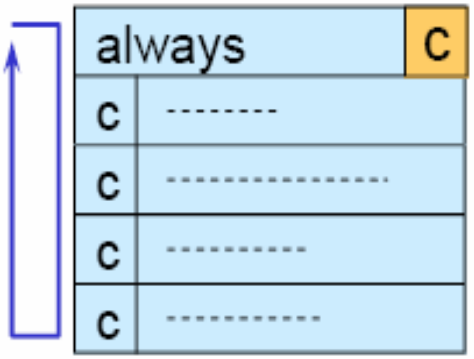
begin-end block

- Used to group statements, so that they execute in sequence.
- The Verilog syntax often requires exactly one statement, for example in an **always**.
- If more than one statement is needed, the statements may be included in a begin-end block.

Procedural Blocks

- In verilog, procedural blocks are basis of behavior modeling
- Procedural blocks are of two types
  - initial procedural block, which execute only once
  - always procedural block, which execute in a loop

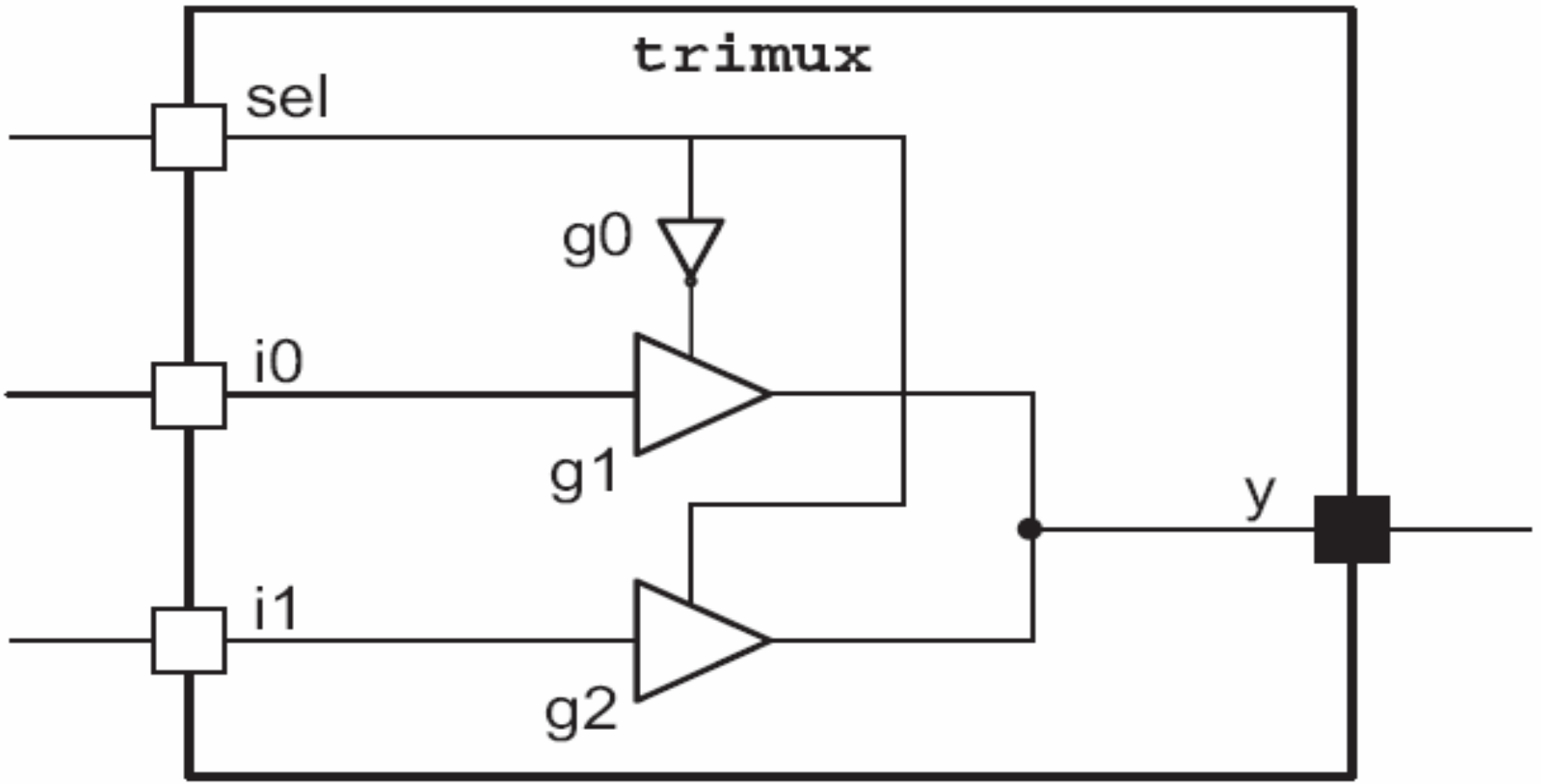




Initial Block

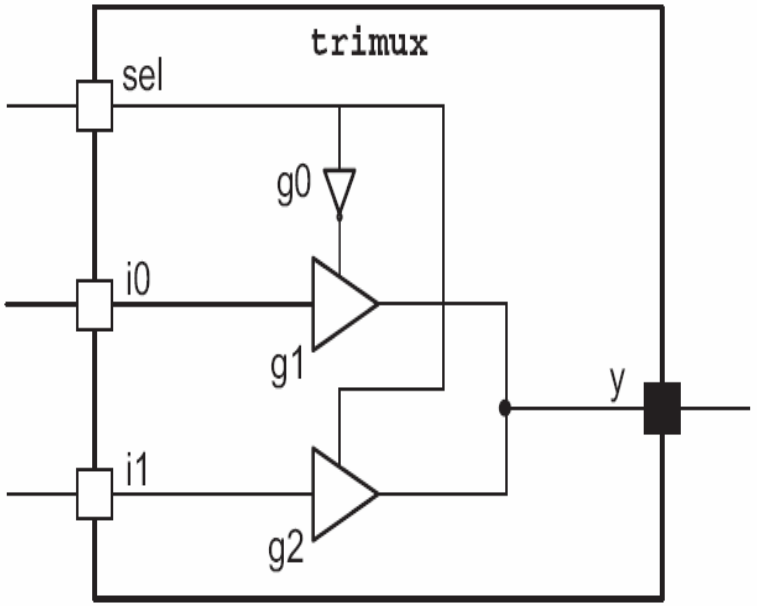
- Used for initialization, monitoring, waveforms and other processes that must be executed only once during simulation
  - ⇨An **initial** block starts at time 0, **executes only once during simulation,** and then does **not** execute again.
  - ⇨Behavioral statements inside an initial block execute sequentially.
  - ⇨Therefore, **order of statements does matter**

Example: Multiplexer Using Tri-State Buffers



Verilog Code for a Multiplexer with Tri-State Buffers

```
`timescale 1ns/100ps
module TriMux (input i0, i1, sel, output y);
wire sel_ ;
    not #5 g0 (sel_, sel);
    bufif1 #4
      g1 (y, i0, sel_),
      g2 (y, i1, sel );
endmodule
```



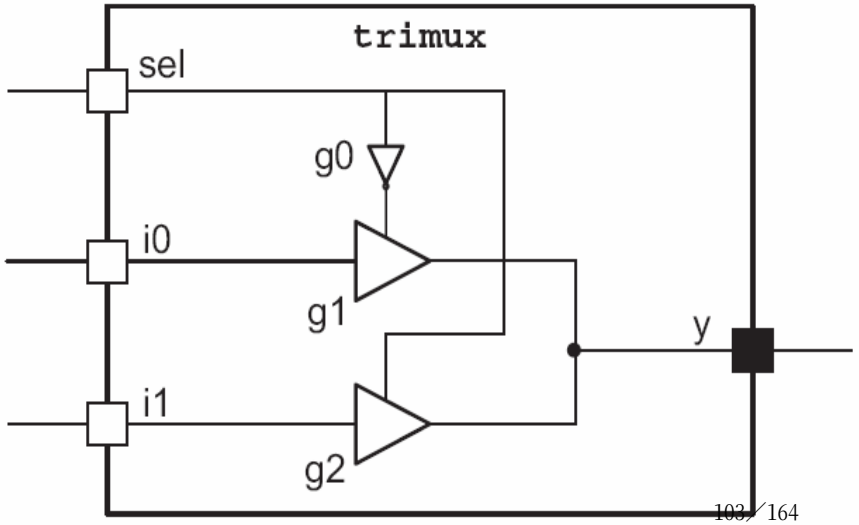
Compiler Directives: `timescale

- **`timescale**
- Defines the time units and simulation precision (smallest increment).
- **Syntax**
- `timescale TimeUnit / PrecisionUnit
- TimeUnit = Time Unit
- PrecisionUnit = Time Unit
- Time = {either} 1 10 100
- Unit = {either} s ms us ns ps fs
- Ex: **`timescale** 1ns/100ps

A Testbench for *TriMux*

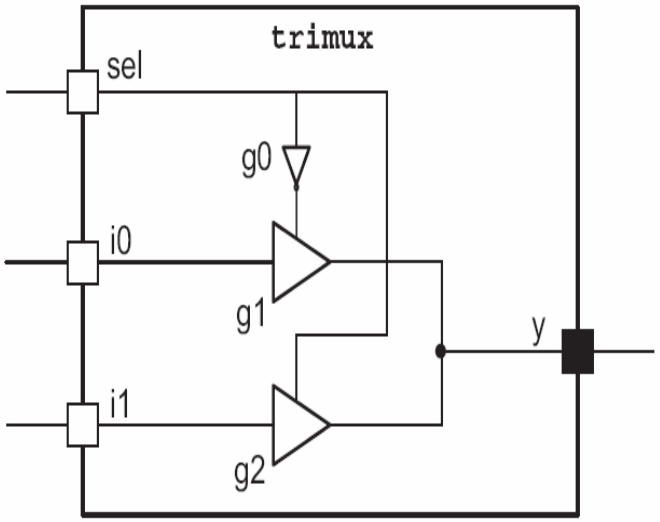
```
`timescale 1ns/100ps
    module TriMuxTest;
    reg i0=0, i1=0, s=0;
    wire y;
    TriMux MUT (i0, i1, s, y);
    initial begin
        #15 i1=1'b1;
        #15 s=1'b1;
        #15 s=1'b0;
        #15 i0=1'b1;
        #15 i0=1'b0;
        #15 $finish;
    end
endmodule
```

```
`timescale 1ns/100ps
module TriMux (input i0, i1,
                    sel, output y);
wire sel_ ;
      not #5 g0 (sel_, sel);
      bufif1 #4
        g1 (y, i0, sel_),
        g2 (y, i1, sel );
endmodule
```

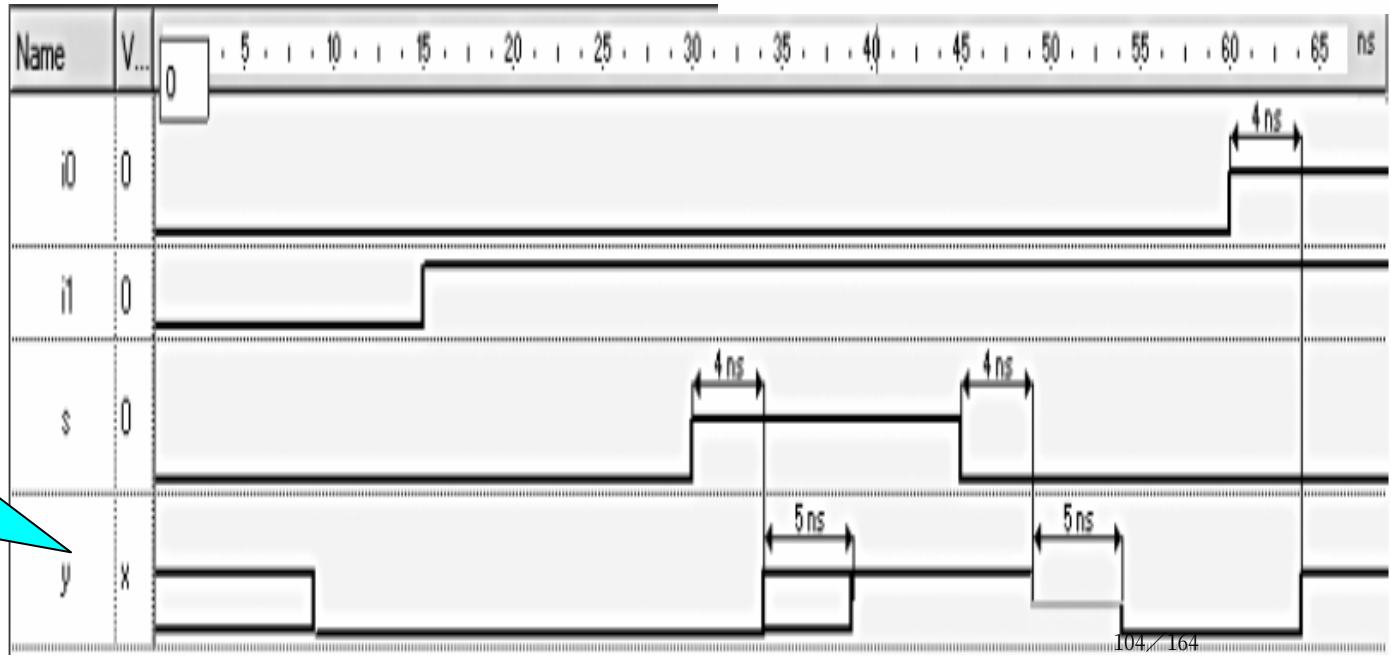


A Testbench for *TriMux*

```
`timescale 1ns/100ps
module TriMux (input i0, i1,
                    sel, output y);

wire sel_ ;
      not #5 g0 (sel_, sel);
      bufif1 #4
        g1 (y, i0, sel_),
        g2 (y, i1, sel );
endmodule
```




```
`timescale 1ns/100ps
      module TriMuxTest;
      reg i0=0, i1=0, s=0;
      wire y;
      TriMux MUT (i0, i1,
      s, y);
      initial begin
            #15 i1=1'b1;
            #15 s=1'b1;
            #15 s=1'b0;
            #15 i0=1'b1;
            #15 i0=1'b0;
            #15 $finish;
      end
endmodule
```



The initial value of a driven net is X and that of an undriven net is Z;s-g0-g1=9ns

The property of nets

- **The initial value of a driven net is X and that of an undriven net is Z.**
- Note that initially *y* is **X** until the value of *i0* propagates to this output at time 9 ns.
- The 9 ns delay is due to a **0** propagating to *sel_* after 5 ns, and then *i0* propagating to *y* after 4 ns.
- At time 34 ns, both *g1* and *g2* conduct. *g1* is still conducting because it takes the **not** gate 5 ns to change the value of *sel_* and **bufif1** an extra 4 ns to stop *g1* from conducting. This causes the value **X** to appear on *y*.
- At time 45 ns when *sel* becomes **0**. Because of this change, after a 4 ns delay,neither *g1* nor *g2* conduct, causing a **Z** (high impedance) to appear on *y* for a period of 5 ns (inverter delay).

4.2.3 Signal Generators for Testbenches

- **Initial** declares a single-pass behavior that begins executing when the simulator is activated
- The statements associated with the behavior are listed with the ***begin…end*** block keywords, and are called procedural statements
- **Procedural assignment operator: =**
  - Inside an initial or always block
  - **Procedural assignments are used to describe stimulus patterns in a testbench**
- **Continuous Assignment**

Procedural Assignment

- **Inside an initial or always block:**

  sum = a + b + cin;
- RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg
  - Primitives or cont. assignment may set wire values
  - **Procedural Assignment drives Reg.**

Continuous Assignment

- Another way to describe combinational function
- Convenient for logical or datapath specifications

Define bus widths

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes

Procedural Assignment vs Continuous Assignment

- **Continuous Assignment**

  assign w = m | p;

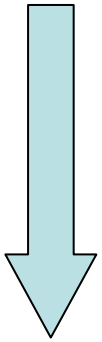
  assign #2 w = m | p;

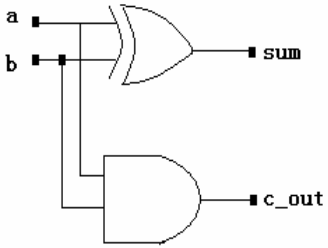
  used only in concurrent Verilog bodies.

- **Procedural Assignment**
  - take place in the **initial** and **always**
  - take place in in procedural bodies

Example 4.7 A testbench for verifying Add_half_0_delay

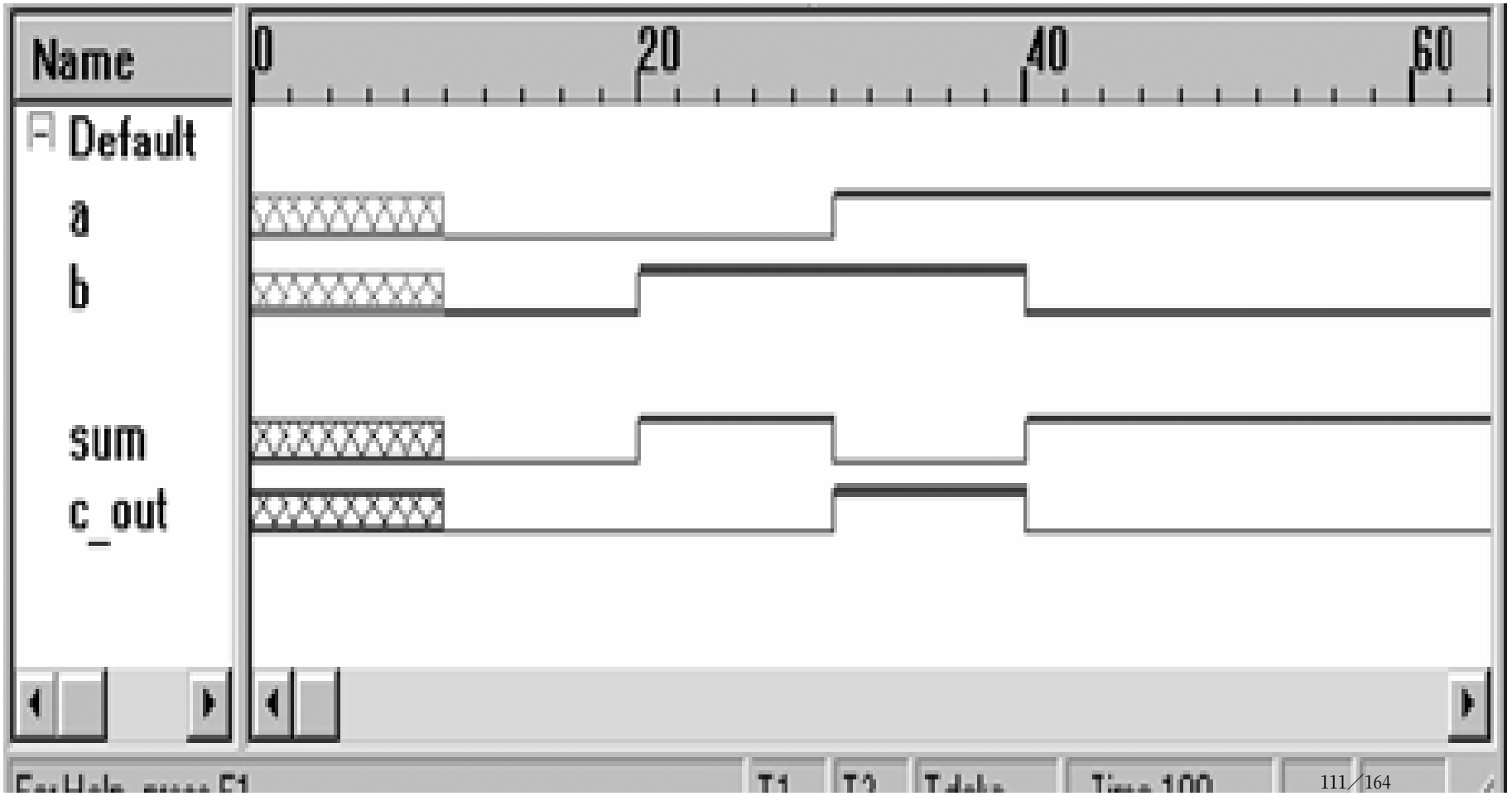
```
module t_add_half();
   wire sum, c_out;
   reg a, b;
   add_half_0_delay M1 (sum, c_out, a, b);      // UUT
   initial begin  // time out
     #100 $finish; //system task
   end
   initial begin  //stimulus
    #10 a=0;b=0;
    #10 b=1;
    #10 a=1;
    #10 b=0;
   end
endmodule
```





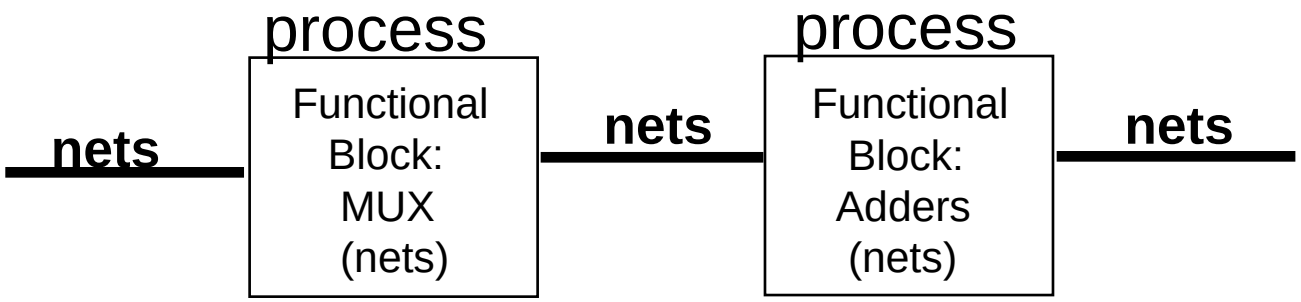

```
module Add_half (sum, c_out, a, b);
    input a, b;
    output c_out, sum;
    xor (sum, a, b);
    and (c_out, a, b);
endmodule
```

Fig 4.17 Simulation result of Add_half_0_Delay

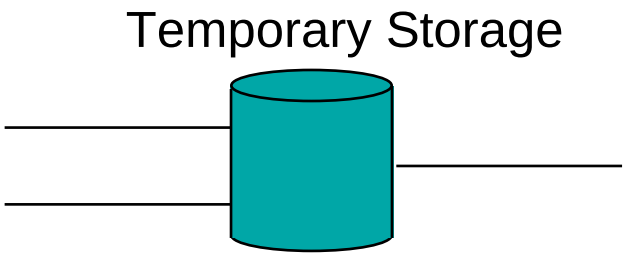


Variable of Net & *reg* type (Data Types)

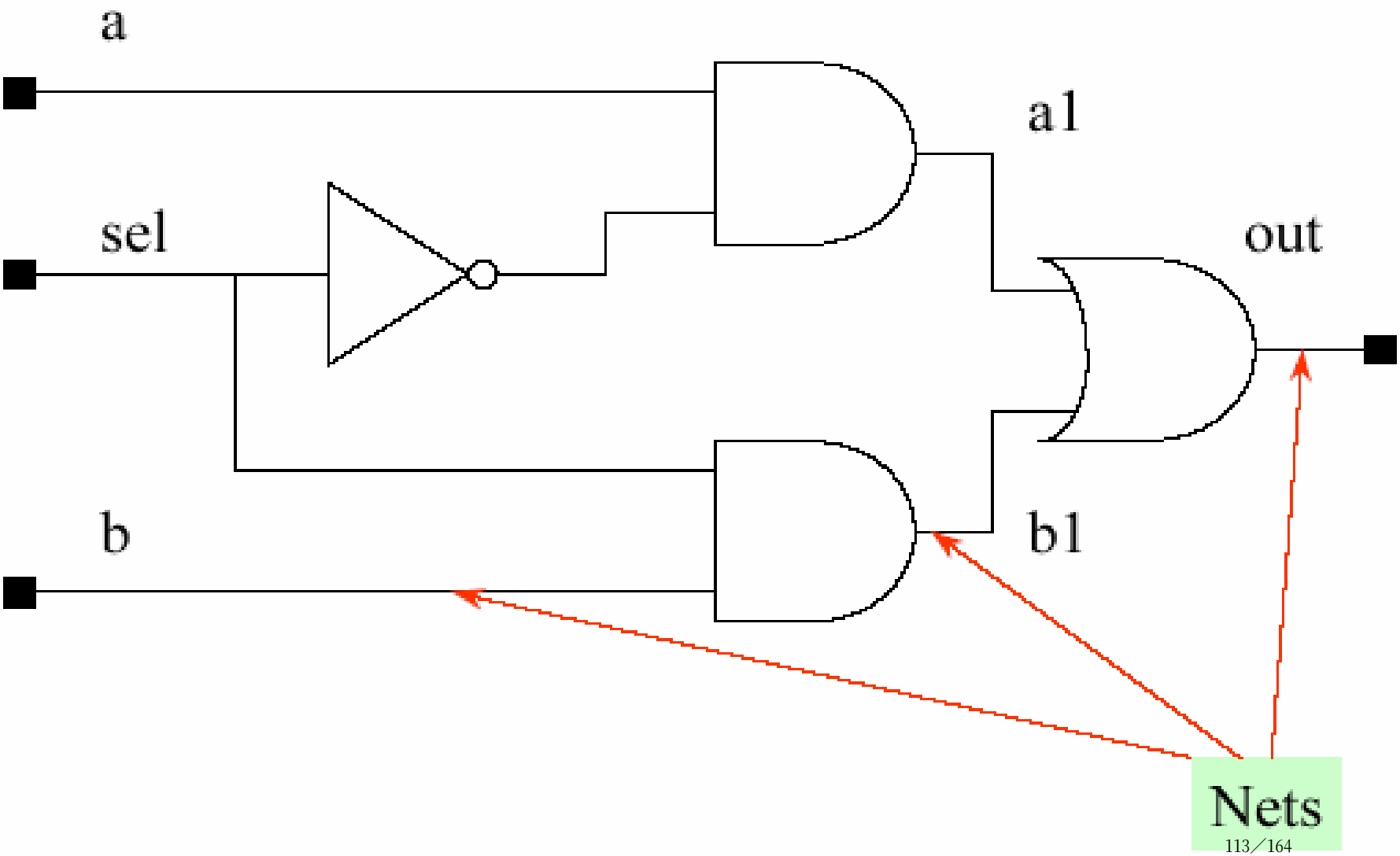
- **Net Data Type** - represent physical interconnect between processes (activity flows)



- **Register Data Type** - represent variable to store data temporarily
  - It does not represent a physical (hardware) register

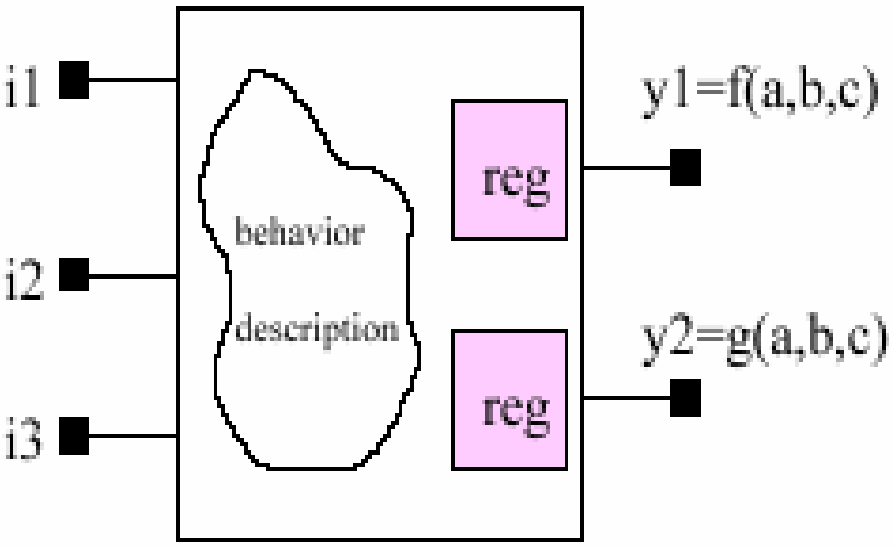


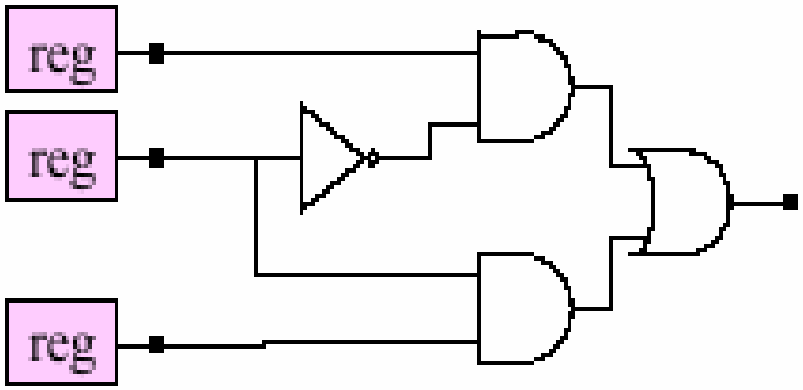
Nets



Registers

- Registers represent abstract storage elements.
- A register holds its value until a new value is assigned to it.
- Registers are used extensively in behavior modeling and in applying stimuli.





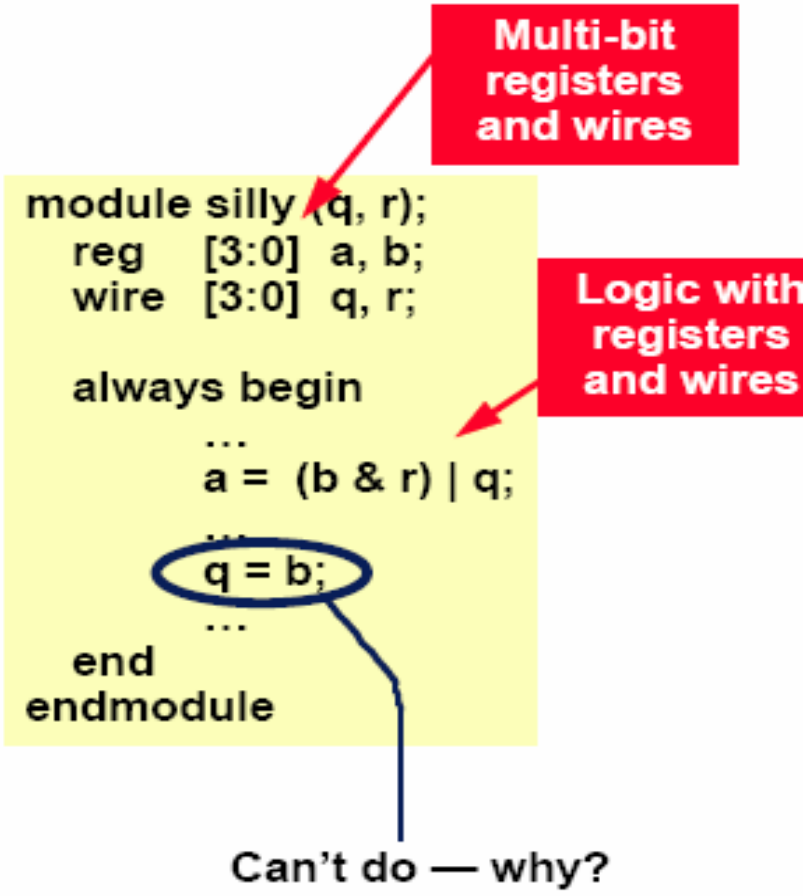
Register Data Types

- **reg** - unsigned variable of any bit size
- **integer** - signed variable (usually 32 bits)
- Bus Declarations:
  - **<data_type>** [*MSB* : *LSB*] <*signal name*> ;
  - **<data_type>** [*LSB* : *MSB*] <*signal name*> ;

- Examples:
  - **reg** <*signal name*> ;
  - **reg** [7 : 0] out ;

Net & Register

- **Registers**
  - Define storage, can be more than one bit
  - Can only be changed by assigning value to them on the left-hand side of a behavioral expression.
- **Wires (actually "nets")**
  - Electrically connect things together
  - Can be used on the right-hand side of an expression
    - Thus we can tie primitive gates and behavioral blocks together!
- **Statements**
  - left-hand side = right-hand side
  - left-hand side must be a register
  - Four-valued logic



Nets and Registers

- Wires and registers can be bits, vectors, and arrays

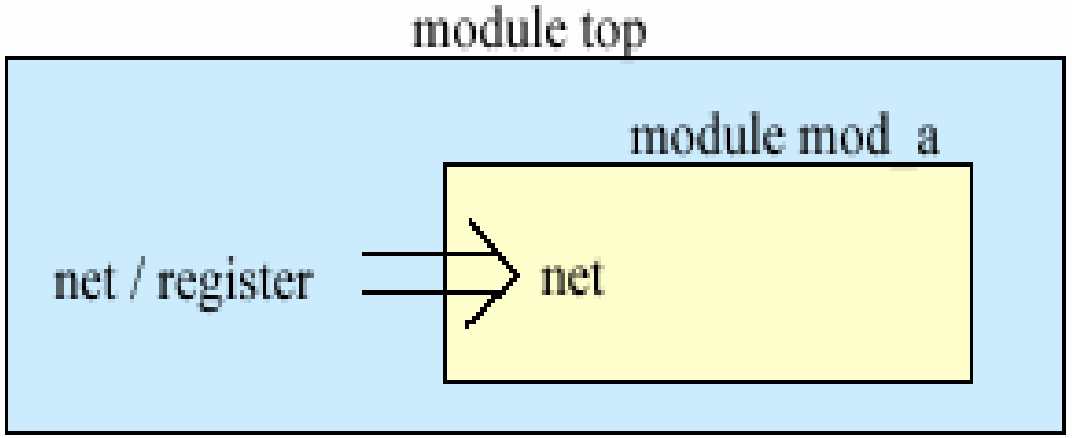
```
wire a;                          // Simple wire
tri [15:0] dbus;                 // 16-bit tristate bus
tri #(5,4,8) b;                      // Wire with delay
reg [-1:4] vec;                   // Six-bit register
integer imem[0:1023];  // Array of 1024 integers
reg [31:0] mymem[0:63];   // A 32-bit memory
```

Common Mistakes

- **An input port can be driven by a net or a register, but it can only drive a net.**
- **Example**

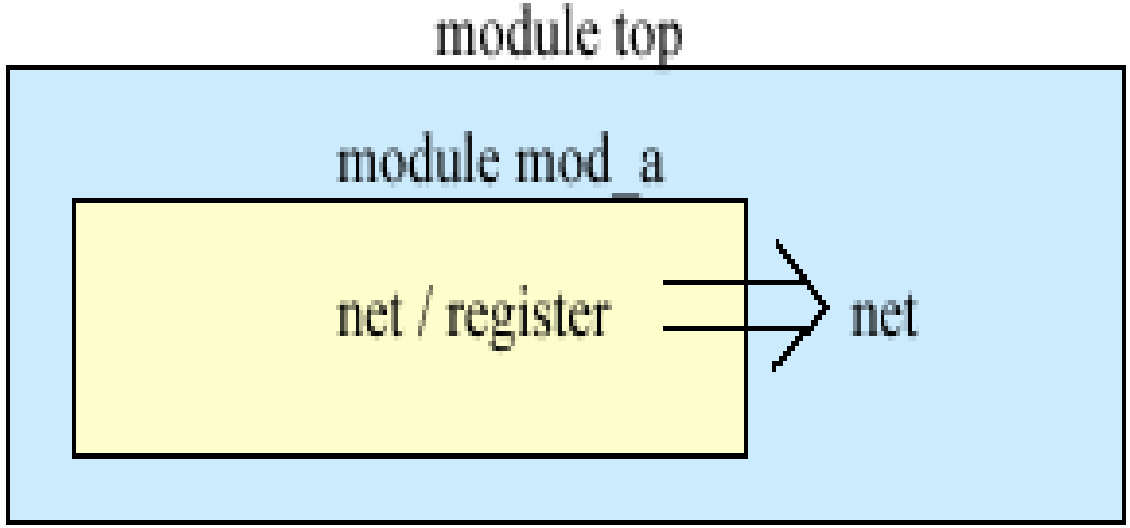
```
module mod_a(out, in);
    output out;
    input  in;
    reg    in;
.....
```

You cannot use input port to drive a register.



Common Mistakes

An output port can be driven by a net or a register, but it can only drive a net.



Example

```
module top (...);
  ....
  reg  rega, regb;
  mod_a U1(rega,  regb);
.....
```

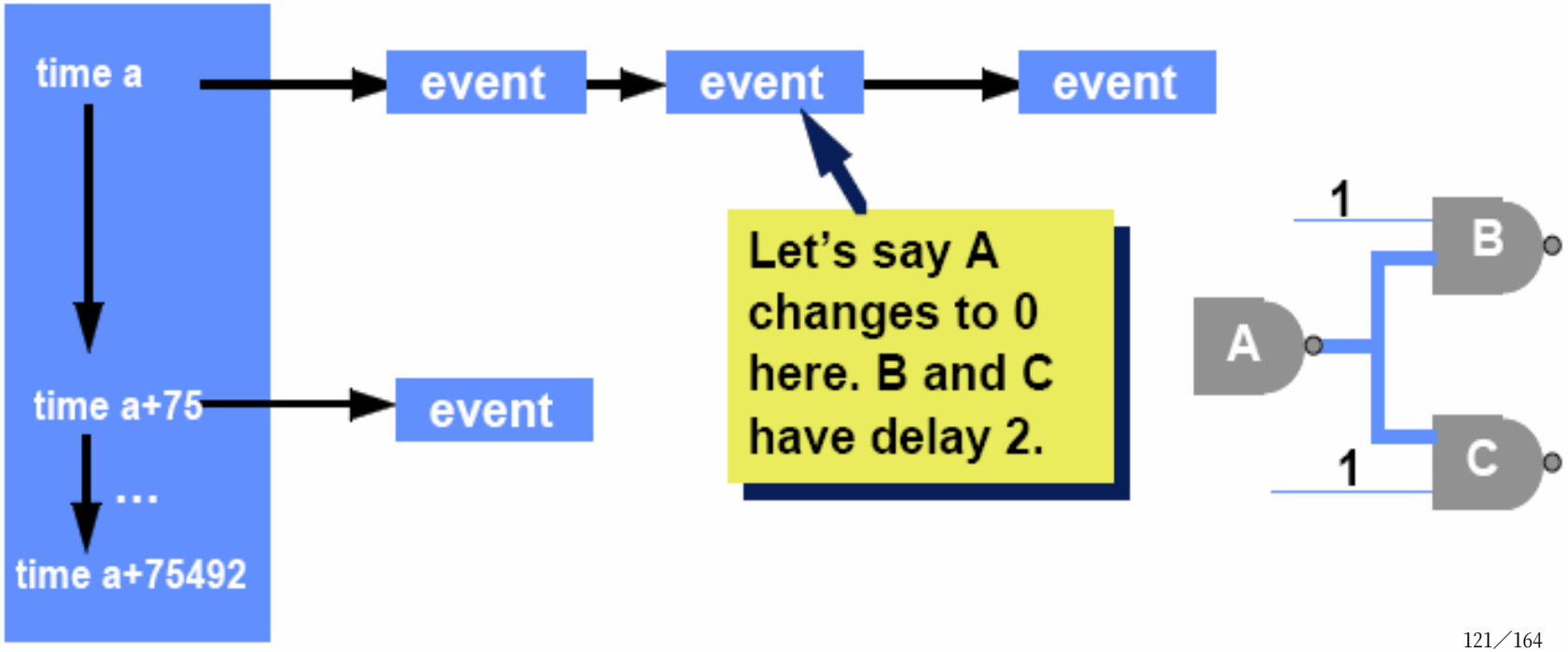
instance *U1*'s output port connect to a register *rega*.

4.2.4 Event-driven simulation

- **An "event":** a change in the value of a signal (variable) during simulation
- Event-driven simulators **are inactive during the interval between events**
- System create a schedule for the events of the outputs by knowing schedule of inputs and structure of a circuit
- Simulator's job is to detect events and schedule any new events that result from their occurrence

Event-driven simulation

- Events are stored in an event list (actually a 2-D list) ordered by time
- Events *execute* at a time and possibly *schedule* their output to change at a later time (a new event)
- When no more events for the current time, move to the next
- Events within a time are executed in arbitrary order

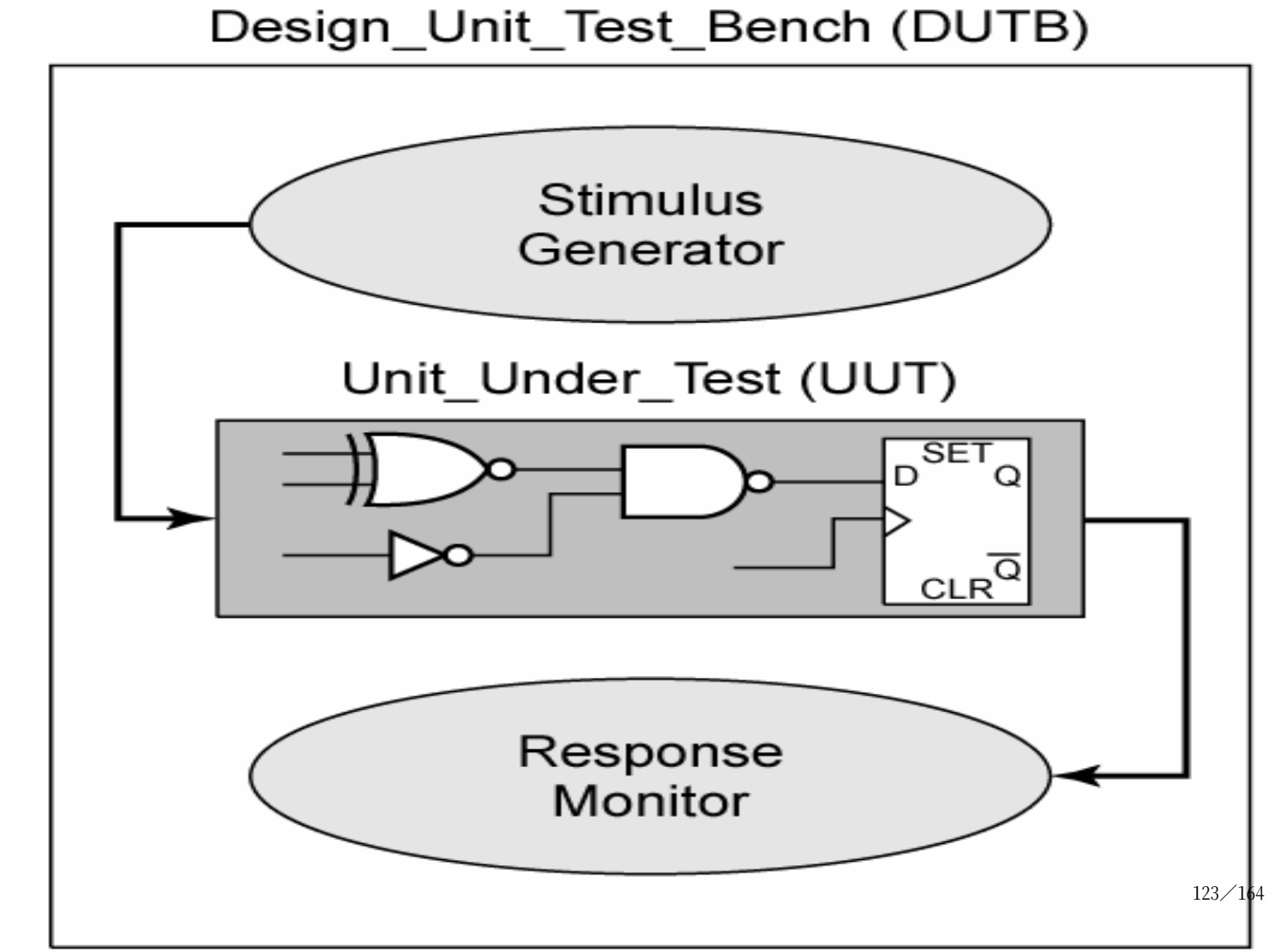


4.2.5 Testbench Template

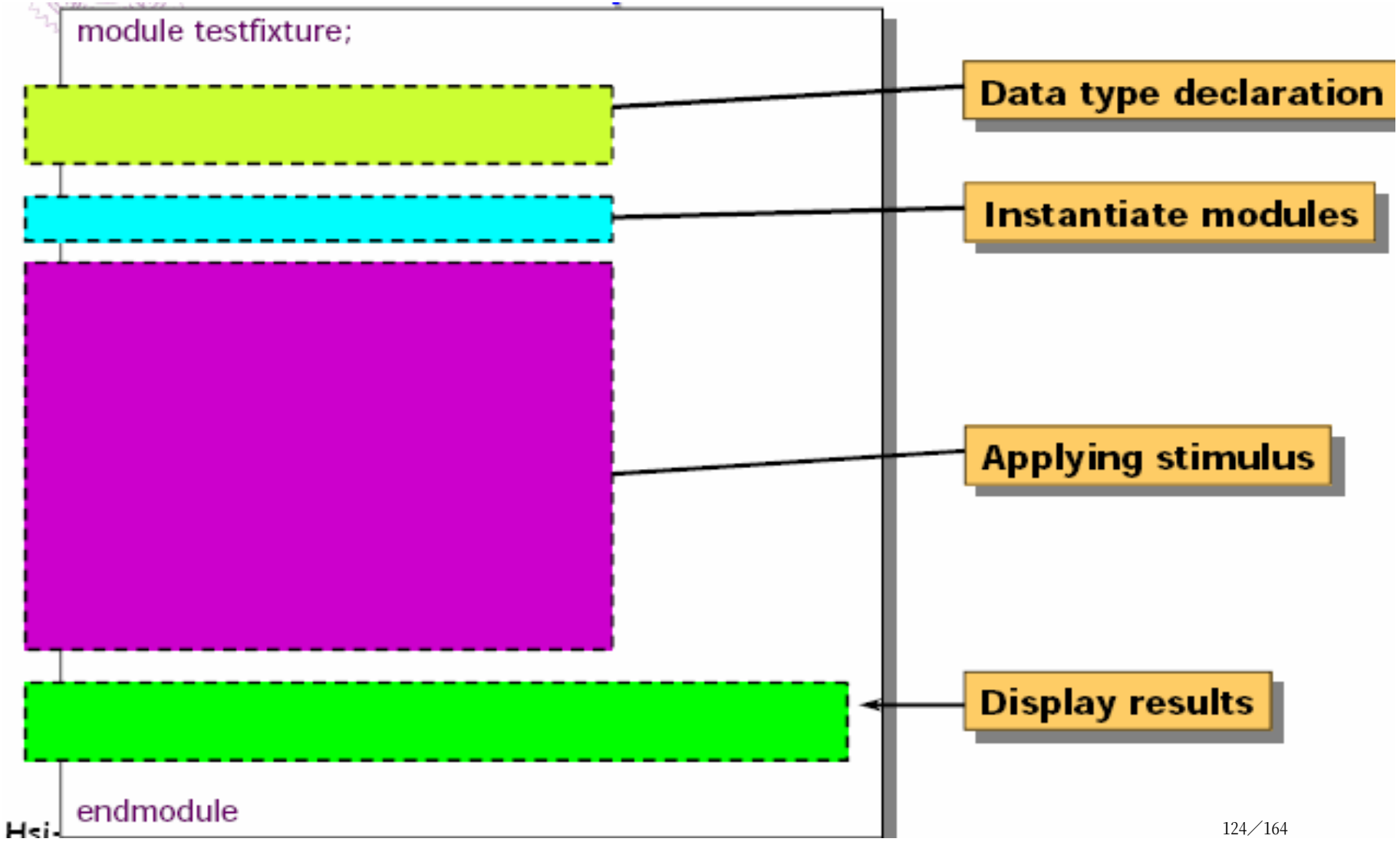
- **Testbenches:** are an important tool in the design flow of an ASIC
- Much effort is expended to develop a thorough testbench,because it is an insurance policy against failure
- The plan should specify :

  what features will be tested and how they will be tested in the testbench
- A test plan should be developed before the testbench is written

A testbench's basic organization



A complete testbench



The general structure: how to develop testbenches

```
module t_DUTB_name();   // substitute the name of the UUT
  reg …;                    // declaration of register variables for primary
  wire …;                   // inputs of the UUT
  parameter                  // declaration of primary outputs of the UUT
                               timeout=// provide a value
  UUT_name M1_instance_name (UUT ports go here);
  initial $monitor ();        // specification of signals to be monitored and
                              // displayed as text
  initial #time_out $stop;  // stopwatch to assure termination of simulation
  initial                    // develop one or more behaviors for pattern
                             // generation and/or
                             // error detection
  begin                      //behavioral statements generating waveforms to the
                             // input ports, and comments documenting the test.
                             // Use the full repertoire of behavioral constructs for
                             // loops and conditionals
  end
endmodule
```

4.2.6 Sized Numbers

- **Sized numbers** specify the number of bits that are to be stored for a value
- **Unsized numbers** are stored as integers having length determined by the host simulator (usually 32 bits)
- **Four formats**:

  binary (b), decimal (d), octal (o), and hexadecimal (h)
- Not case-sensitive
- By default a number is interpreted to be a decimal value
- For example
- 8'ha denotes an 8-bit stored value corresponding to the hexadecimal number a, the binary value in memory will be 0000_1010

Number Representation Examples

Number representation	Binary equivalent	Explanation
4'd5	0101	Decimal 5 is interpreted as a 4-bit number.
8'b101	00000101	Binary 101 is turned into an 8-bit number.
12'h5B_3	010110110011	Binary equivalent of hex; underscore is ignored.
-8'b101	11111011	This is the 2's complement of the number in the above example.
10'o752	0111101010	This is the octal 752 with a 0 padded to its left to make it a 10-bit number.
8'hF	00001111	Hexadecimal F is expanded to 8 bits by padding zeros to its left.
12'hXA	xxxxxxxx1010	Hexadecimal XA is expanded to 12 bits by extending the left X.
12'shA6	Signed 111110100110	This is an 8-bit number treated as a 2's complement signed number.
-4'shA	Signed 0110	The 2's complement (because of the minus sign) 4-bit A is regarded as a signed number.
596	1001010100	This is a positive constant.

4.3 Propagation Delay

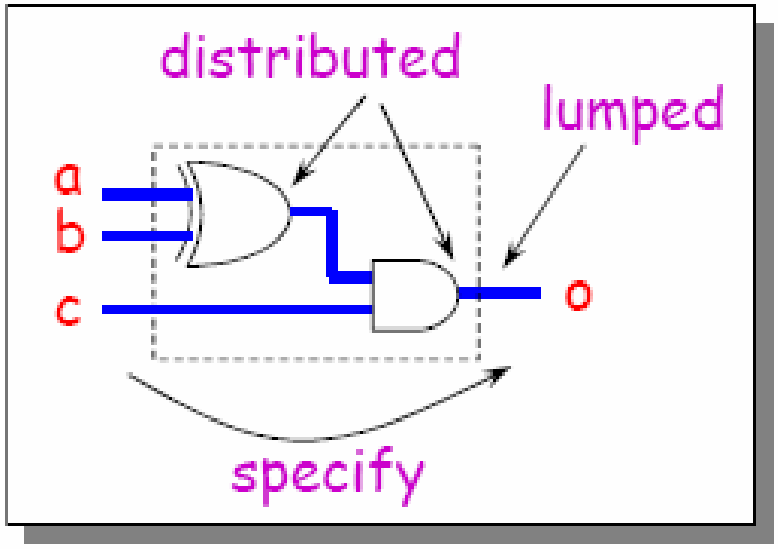
- Physical logic gates have a propagation delay between the time that an input changes and the time that the output responds to the change
- All primitives and nets have a default propagation delay of 0
- 0 delay means that the output responds to the input immediately

Delay specifications in primitives

- Verilog supports (rise,fall,turn-off) delay specification.
- All delay specifications in Verilog can be specified as (*minimum:typical:maximum*) delays.
- Example
  - Bufif (2.5:3:3.4, 2:3:3.5, 5:7:8) U2(out,in,ctl);

Modeling delays

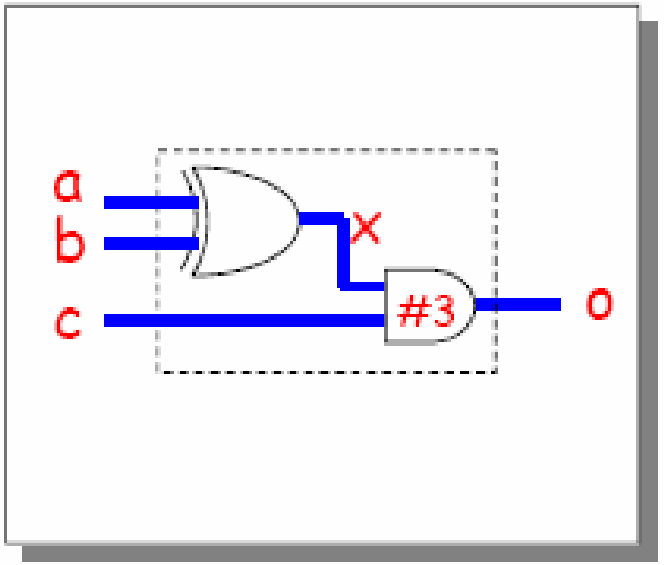
- Three ways
  - Lump the delay at the output of the last gate
  - Distribute the delay across the outputs of each gate
  - Specify pin-to-pin module path delays in a ***specify*** block
- Synthesis tools totally ignore all delay specification in the Verilog source code.



Lumped Delays

- Most easy to implement
- Least accurate
  - All pin-to-pin paths have the same delay
- Simulates fastest

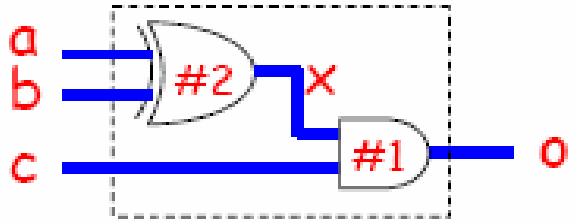
```
`timescale 1ns/1ns...
module xor_and(o,a,b,c);
output o;
input a,b,c;

xor (x,a,b);
and #3 (o,c,x);

endmodule
```



Distributed Delays

- More difficult to implement
- More accurate
  - Pin-to-pin paths can have different delays
- Simulates slower

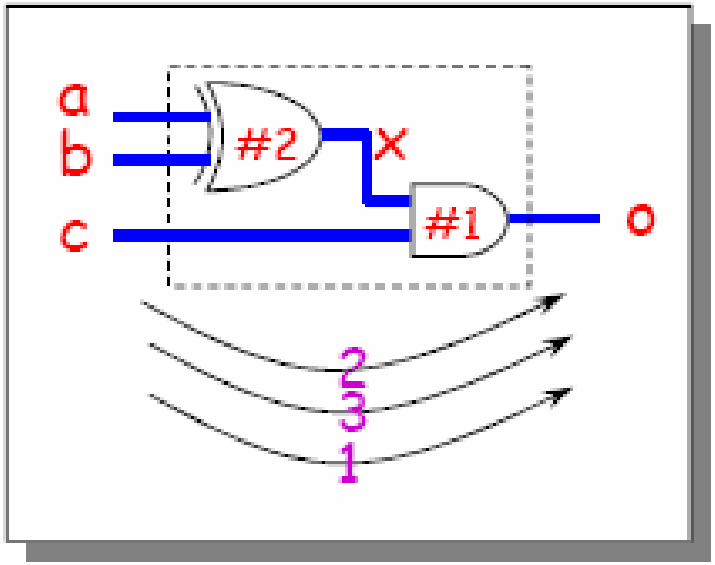
```
`timescale 1ns/1ns...
module xor_and(o,a,b,c);
output o;
input a,b,c;

xor #2 (x,a,b);
and #1 (o,c,x);

endmodule
```



Module Path Delays

Specify pin-to-pin module path delays

```
`timescale 1ns/1ns...
module xor_and(o,a,b,c);
output o;
input a,b,c;

xor (x,a,b);
and (o,c,x);

specify
  (a*>o)=2;
  (b*>o)=3;
  (c*>o)=1;
endspecify

endmodule
```



Specify

- A specify block is used **to describe path delays from a module's inputs to its outputs, and timing constraints** such as setup and hold times.
- A specify block allows the timing of a design to be described separately from the behaviour or structure of a design.
- **Syntax**

```
specify
  SpecifyItems...
endspecify
```

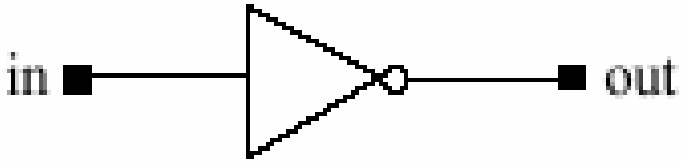
Delays on Primitive Instances

- **Instances of primitives may include delays**

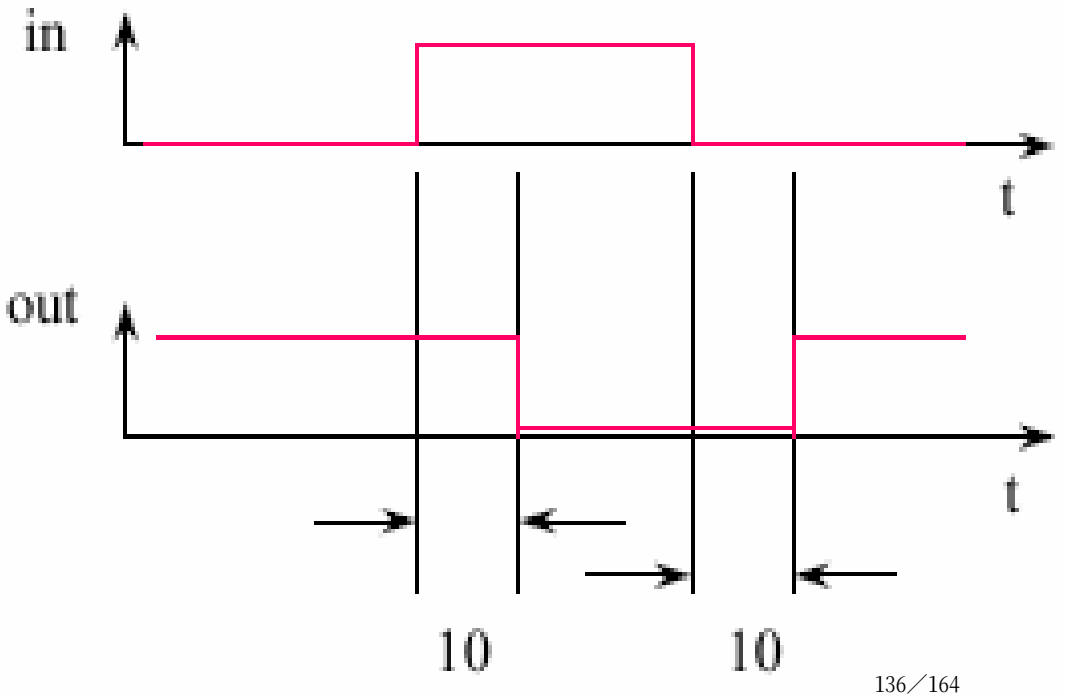
```
bufb1(a, b);                // Zero delay
buf  #3  b2(c, d);          // Delay of 3
buf  #(4,5)  b3(e, f);      // Rise=4, fall=5
buf  #(3:4:5)  b4(g, h);    // Min-typ-max
```

Delay Specification in Primitives

Delay specification defines the propagation delay of that primitive gate.



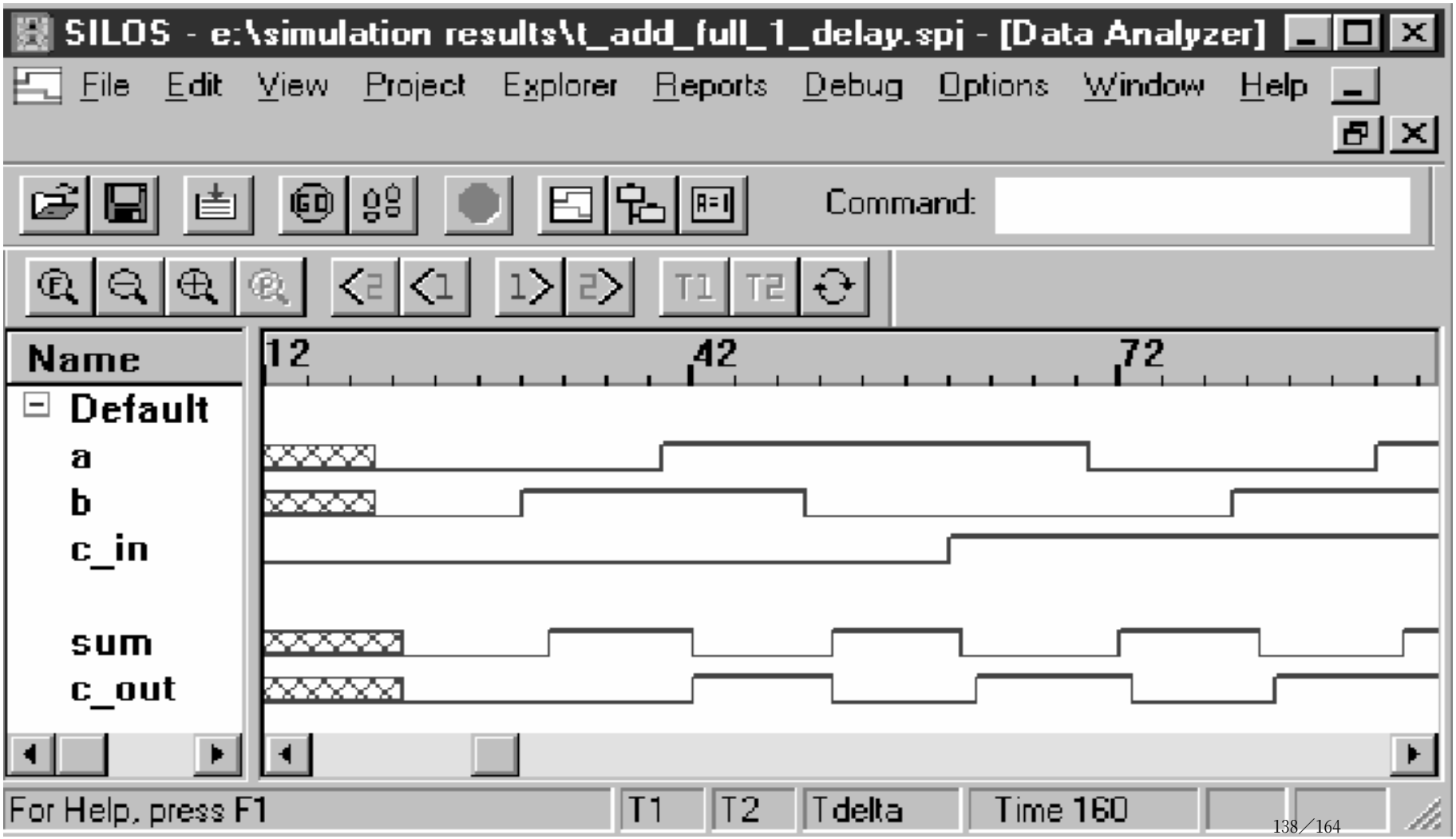
not #10 (out, in);



Example 4.8

```
module add_full (sum,c_out,a,b,c_in);
  output      sum, c_out;
  input       a,b,c_in;
  wire        w1,w2,w3;
  add_half   M1 (w1,w2,a,b);
  add_half   M2 (sum, w3,w1,c_in);
  or     #1 M3 (c_out,w2,w3);
endmodule

module add_half (sum, c_out, a, b);
  output      sum, c_out;
  input       a,b;
  xor         #1 M1 (sum,a,b);
  and         #1 M2 (c_out,a,b);
endmodule
```

Fig 4.18 The unit-delay model of the signal activity



The timing characteristics of ASIC & FPGAs

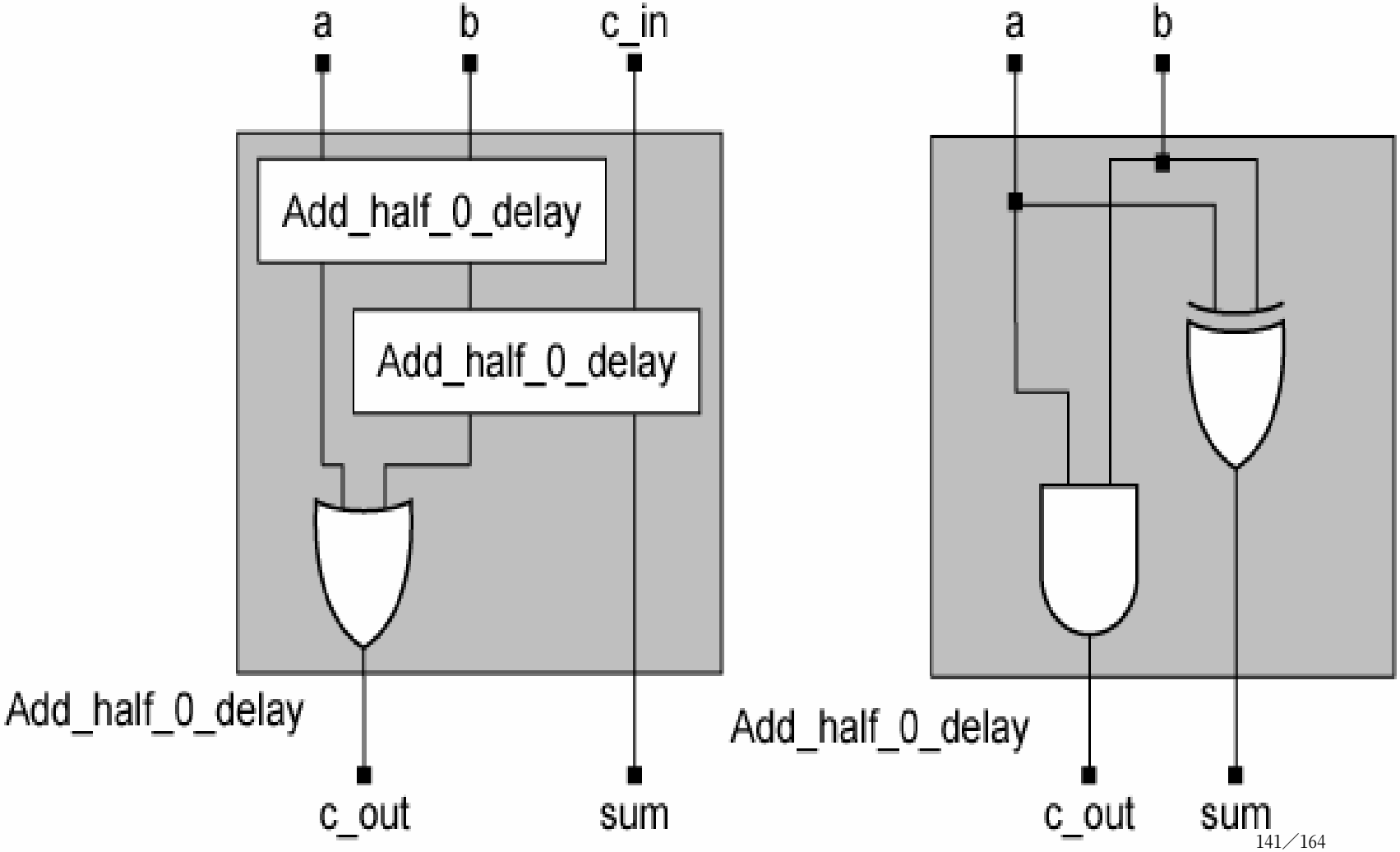
- The timing characteristics of FPGA are embedded within the synthesis tools for FPGAs
- The timing characteristics of ASIC are embedded within the model of the cell
- Circuit designers do not attempt to create accurate gate-level timing models of a circuit by manual methods.
- Designers rely on a synthesis tool to implement a design that will satisfy timing constraints

Example 4.9 The models of Add_half_ASIC and Add_full_ASIC

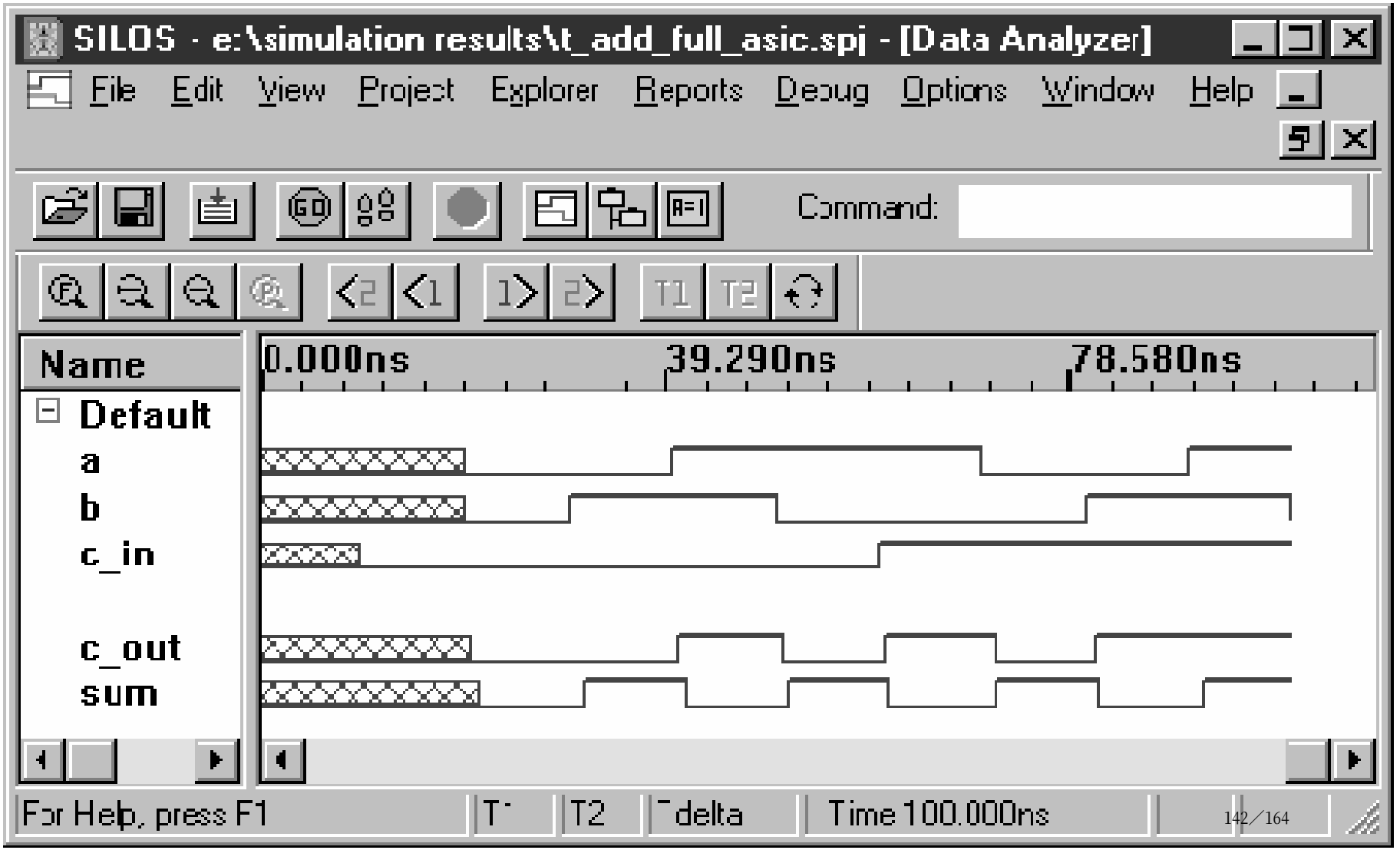
```
'timescale 1ns/1ps
module add_full_ASIC
  (sum,c_out,a,b,c_in);
  output sum,c_out;
  input a,b,c_in;
  wire w1,w2,w3;
  wire c_out_bar;
  add_half_ASIC M1(w1,w2,a,b);
  add_half_ASIC M2(sum,
    w3,w1,c_in);
  norf201 M3(c_out_bar,w2,w3);
  invf101 M4(c_out,c_out_bar);
endmodule
```

```
module add_half_ASIC(sum,c_out,a,b)
  output sum, c_out;
  input a,b;
  wire  c_out_bar;
  xorf201 M1(sum,a,b);
  nanf201 M2(c_out_bar,a,b);
  invf101 M3(c_out,c_out_bar);
endmodule
```

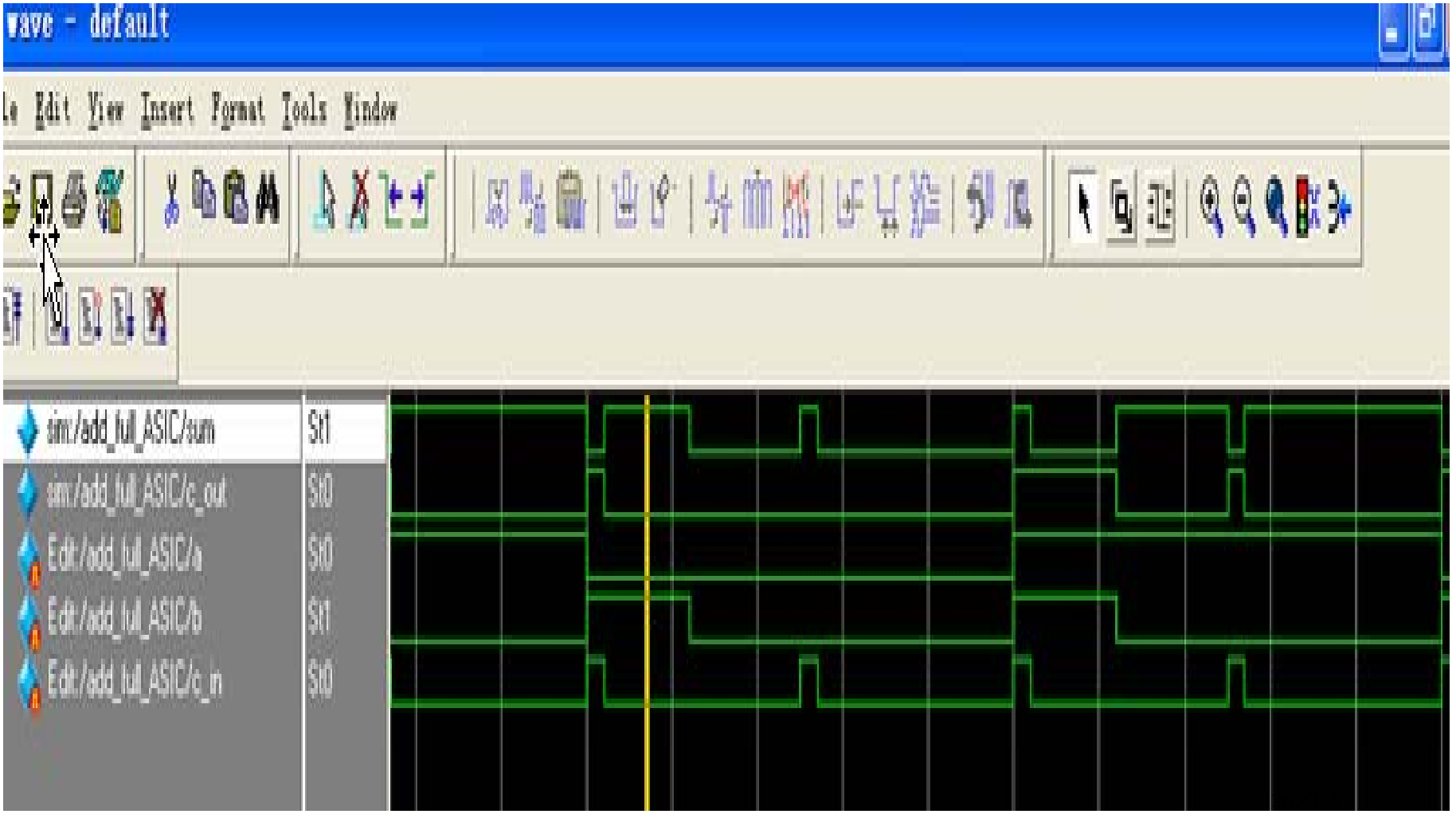
Full and half adders



The effect of propagation delays by simulating Add_full_ASIC



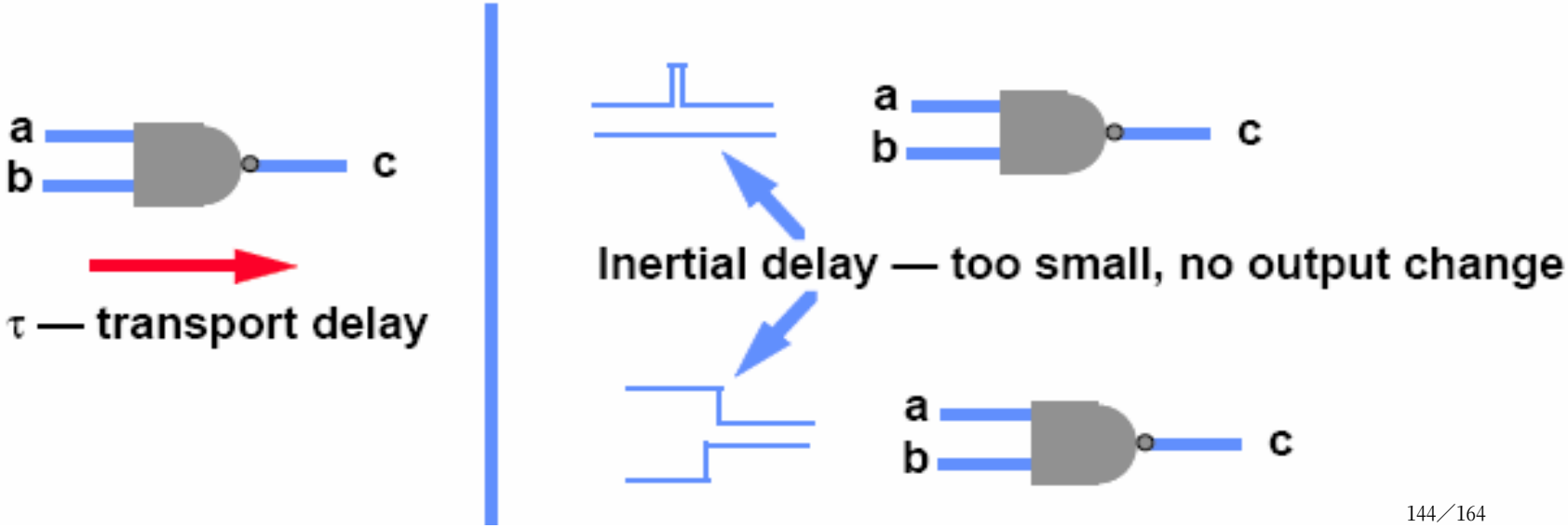
Simulation result



Kinds of delays in verilog

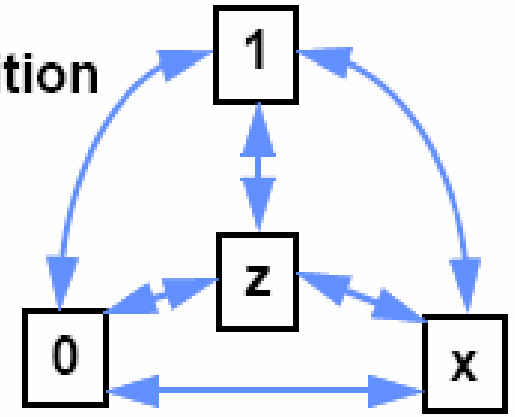
Definitions

- Zero delay models — functional testing
  - there's no delay, not cool for circuits with feedback!
- Unit delay models — all gates have delay 1. OK for feedback
- Transport delay — input to output delay
- Inertial delay — how long must an input spike be to be seen?
  - in Verilog, inertial == transport



Delay Models

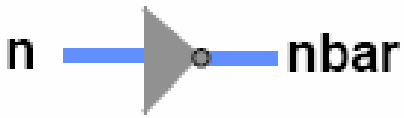
- Other factors
  - Delay can be a function of output transition
  - Need a number for each of the arrowheads



- Verilog example

```
not # (3, 5, 7) (nbar, n);
```

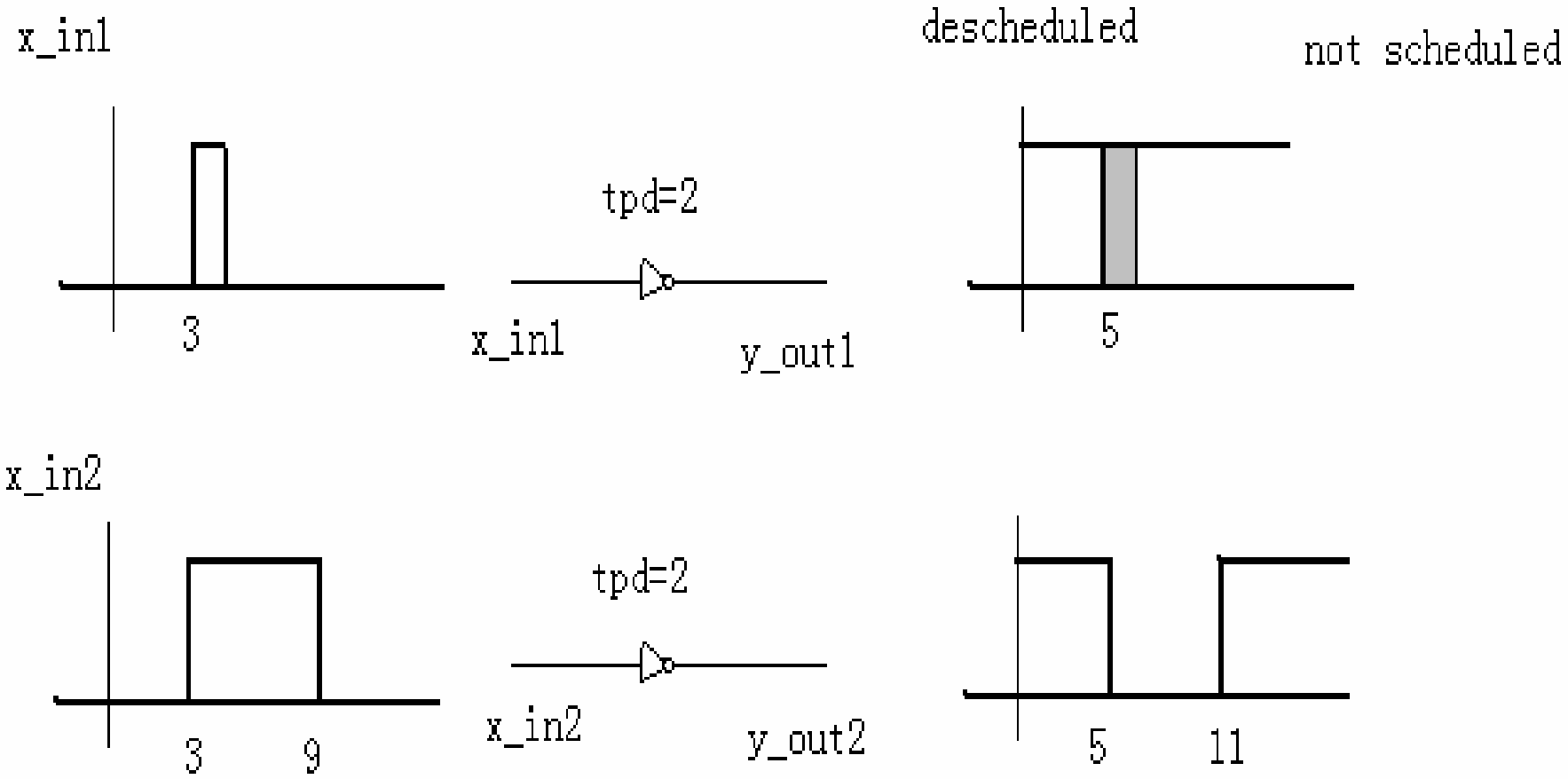
primitive gate instantiation	rising delay	falling delay	delay to z (tristate gates only)
	0->1	1->0	
	0->x	1->x	
	x->1	x->0	
	z->1	z->0	



4.3.1 Inertial delay

- Propagation delay of a logic gate is affected by its internal structure and the circuit that it drives
- Internal delay is the intrinsic delay of the gate
- Every conducting path has some capacitance, as well as resistance, and **charge cannot accumulate or dissipate instantly**
- **Inertial delay has the effect of suppressing input pulses whose duration if shorter than the propagation delay of the gate**

Example 4.10 (Fig.4.20)

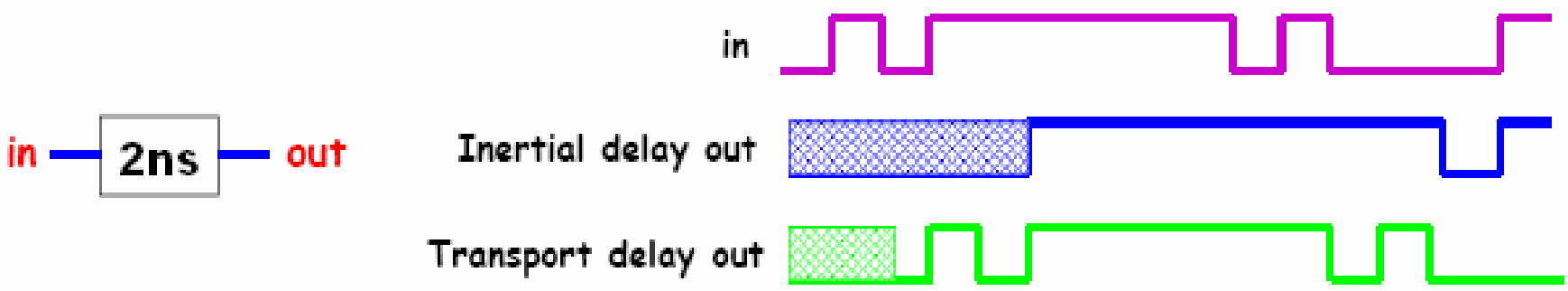


4.3.2 Transport Delay

- The time-of-flight of a signal traveling a wire of a circuit is modeled as a transport delay
- Narrow pulses are not suppressed
- Verilog can assign delay to model transport delay effects
- Wire delays are declared with the declaration of a wire
- For example:
- wire #2 A_long_wire declares that A_long_wire has a transport delay of two time steps

Inertial and Transport Delay Mode

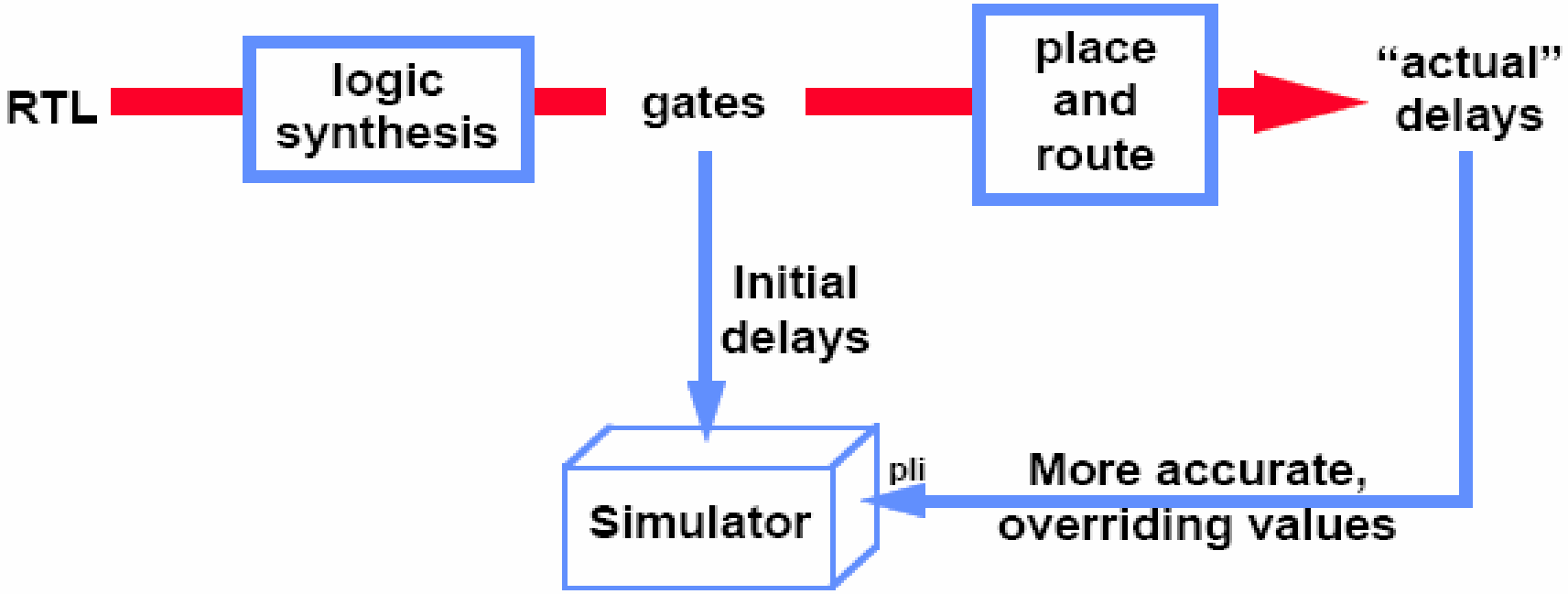
- Inertial delay mode
  - "swallows" input pulses of shorter duration than the delay
- Pure transport delay mode
  - Passes all input pulses, regardless of width
- In Verilog-XL or NC-Verilog, use +pulse_e and +pulse_r command line options.



Overridden Delays

Delays Overridden

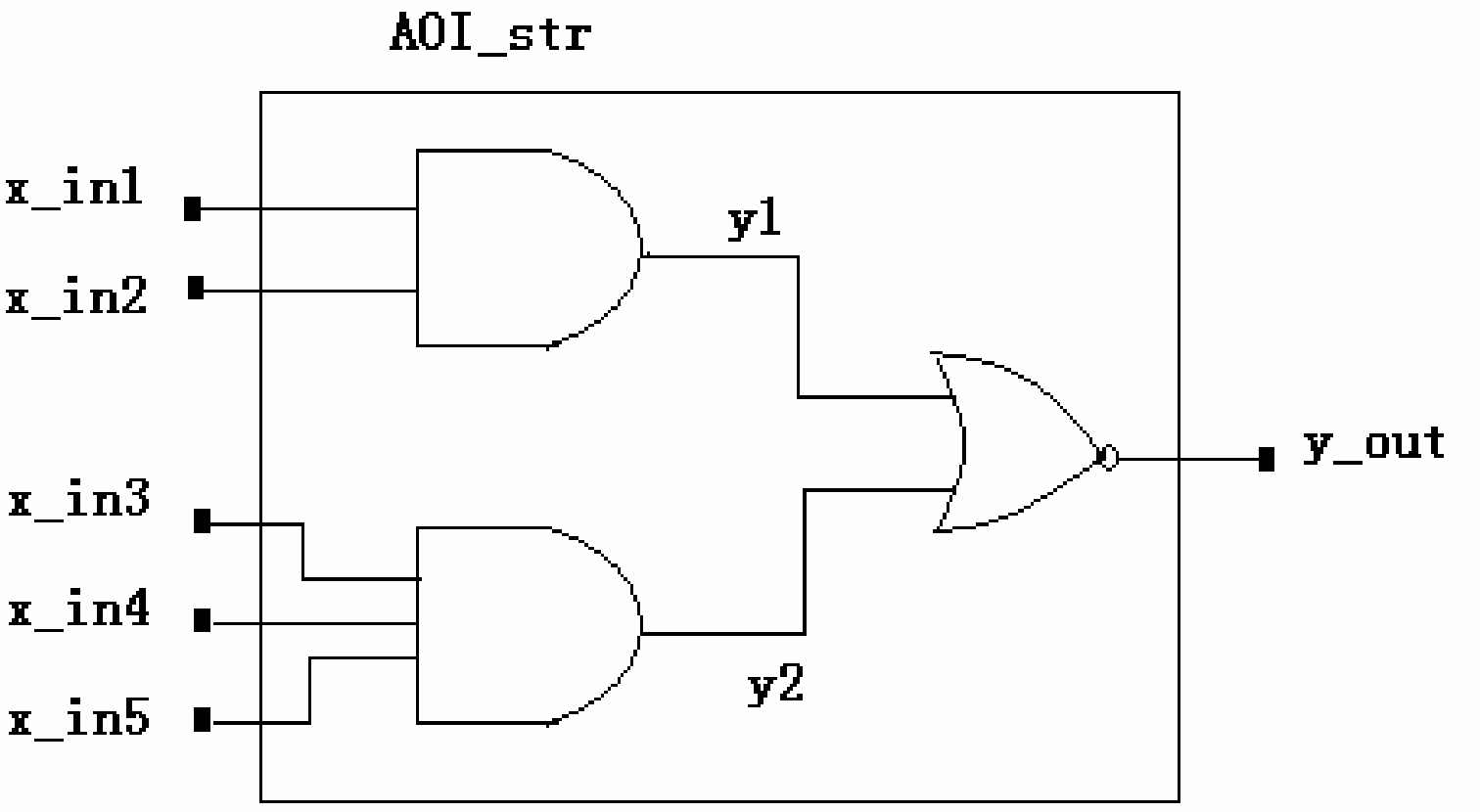
- Use "actual" delays to override specified model delays
- Most importantly, delay due to loading and path lengths is made more accurate
  - generally, this adds to the wire delay accuracy



4.4 Truth table models of combinational and sequential logic with Verilog

- Verilog has a mechanism for building user-defined primitives (UDPs)
- Use truth tables to describe sequential behavior and/or more complex combinational logic
- UDPs are widely used in ASIC cell libraries because they **simulate faster and require less storage than modules**
- A UDP has only a single, scalar (single-bit), output port; Input ports of a UDP must be scalars

AOI_UDP(and,or,inverter)

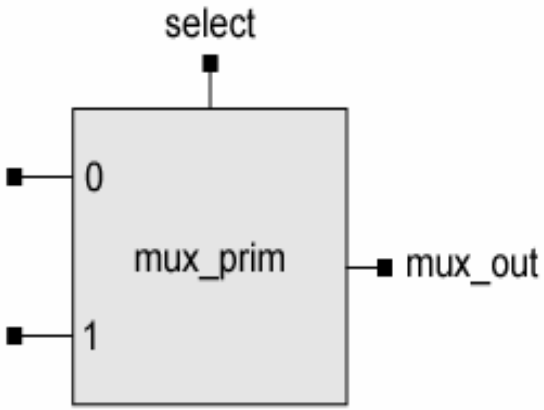


Example 4.11
AOI_UDP(and,or,inverter)

```
primitive AOI_UDP
    (y,x_in1,x_in2,x_in3,x_in4,x_in5);
  output y;
  input
    x_in1,x_in2,x_in3,x_in4,x_in5;
  table
//x1 x2 x3 x4 x5:  y
  0   0   0   0   0 : 1;
  0   0   0   0   1 : 1;
  0   0   0   1   0 : 1;
  0   0   0   1   1 : 1;
  0   0   1   0   0 : 1;
  0   0   1   0   1 : 1;
  0   0   1   1   0 : 1;
  0   0   1   1   1 : 0;

  0   1   0   0   0 : 1;

  0   1   0   0   1 : 1;
  0   1   0   1   0 : 1;
  0   1   0   1   1 : 1;
  0   1   1   0   0 : 1;
  0   1   1   0   1 : 1;
  0   1   1   1   0 : 1;
  0   1   1   1   1 : 0;

  1   0   0   0   0 : 1;
  1   0   0   0   1 : 1;
  1   0   0   1   0 : 1;
  1   0   0   1   1 : 1;
  1   0   1   0   0 : 1;

  1   0   1   0   1 : 1;
  1   0   1   1   0 : 1;
  1   0   1   1   1 : 0;

  1   1   0   0   0 : 0;
  1   1   0   0   1 : 0;
  1   1   0   1   0 : 0;
  1   1   0   1   1 : 0;
  1   1   1   0   0 : 0;
  1   1   1   0   1 : 0;
  1   1   1   1   0 : 0;
  1   1   1   1   1 : 0;
  endtable
endprimitive
```

UDP for a two-input multiplexer

```
primitive mux_prim (mux_out, select, a, b);
  output mux_out;
  input select, a, b;
  table
// select a b : mux_out
        0  0 0 :       0;   // order of table columns 5 port order of inputs
        0  0 1 :       0;   // one output, multiple inputs, no inout
        0  0 x :       0;   // only 0, 1, x on input and output
        0  1 0 :       1;   // A z input in simulation is treated as x
        0  1 1 :       1;   // by the simulator
        0  1 x :       1;   // last column is the output
        1  0 0 :       0;
        1  1 0 :       0;
        1  x 0 :       0;
        1  0 1 :       1;
        1  1 1 :       1;
        1  x 1 :       1;
        x 0 0 :       0;    //    reduce pessimism
        x 1 1 :       1;
   endtable      // Note: combinations not explicitly specified will drive
endprimitive    // 'x' under simulation
```



Using a shorthand notation ? to reduce the input entires

- The entries for the inputs in a truth table can be reduced by using a shorthand notation
- The **?** Symbol allows an input to take on any of the three values, 0,1,and ***x***.

Example 4.13 The UDP of a two-input multiplexer

table

```
// shorthand notation: ? Represents iteration of the table
   entry over
// the values 0,1,x.
// i.e., don't care on the input
   // select a  b : mux_out
   // 0         0  ? : 0;   // ?=0,1,x shorthand notation.
   // 0         1  ? : 1;
   // 1         ?  0 : 0;
   // 1         ?  1 : 1;
   // ?         0  0 : 0;
   // ?         1  1 : 1;
```

endtable

Hardware can exhibit two basic kinds of sequential behavior

- Level-sensitive behavior (e.g., transparent latch)
- Edge-sensitive behavior (e.g., a D-type flip-flop)
- Level-sensitive devices respond to any change of an input while the enabling input is high
- Edge-sensitive devices ignore their inputs until a synchronizing edge occurs
- The output of a sequential user-defined primitive must be declared to have type ***reg***

UDP of Sequential hardware devices

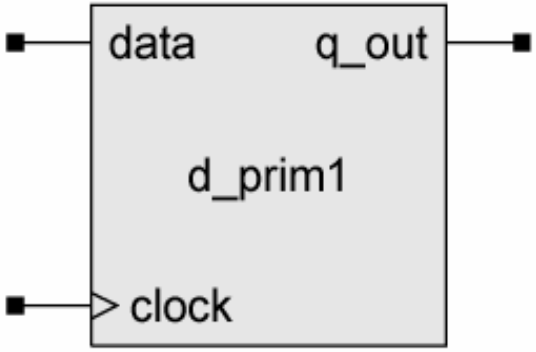
- Sequential UDP may have the behavior of one or the other, or a combination of both.
- A truth table that describes sequential behavior has input columns followed by a colon (:) and a column for the present state of the device, and another colon and a column for its next state
- The output of the UDP must be declared to have type reg because the value of the output is produced abstractly, by a table, and must be held in memory during simulation.

Example 4.14 A Transparent Latch (level-sensitive)

```
primitive latch_rp (q_out, enable, data);
  output q_out;
  input  enable, data;
  reg  q_out;
  table
//  enable data   state   q_out/next_state
     1        1   :  ?   :   1 ;
     1        0   :  ?   :   0 ;
     0        ?   :  ?   :   - ;
// above entries do not deal with enable=x.
// ignore event on enable when data=state:
     x        0   :  0   :   - ;
     x        1   :  1   :   - ;
// note: the table entry '-' denotes no change of the output
   endtable
endprimitive
```

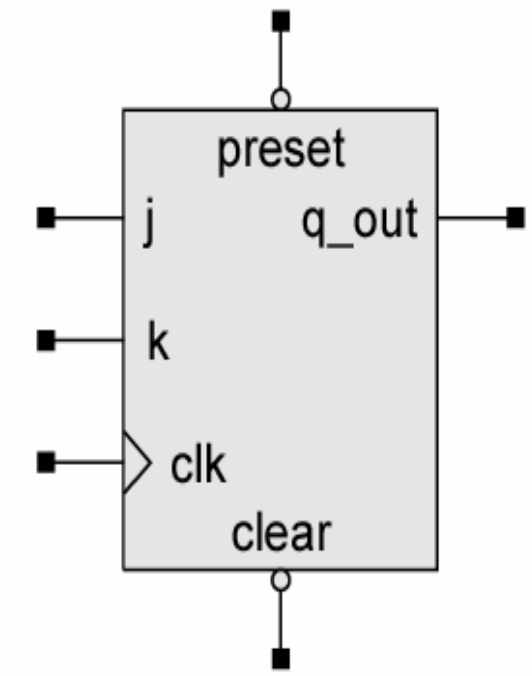
Example 4.15 D-type flip-flop (edge-sensitive)

```
primitive d_prim1 (q_out, clock, data);
  output  q_out;
  input   clock, data;
  reg     q_out;
  table
   //   clk   data   state    q_out/next_state
      (01)    0     :  ?     :  0 ;   //rising clock edge
      (01)    1     :  ?     :  1 ;
      (0?)    1     :  1     :  1 ;
      (?0)    ?     :  ?     :  - ;   // falling or steady
                                      //clock  edge
       ?     (??)   :  ?     :  - ; // stedy clock, ignore
                                       //data transitions
  endtable
endprimitive
```



Example 4.16 UDP for a J-K flip-flop (asynchronous preset and clear with edge-sensitive)

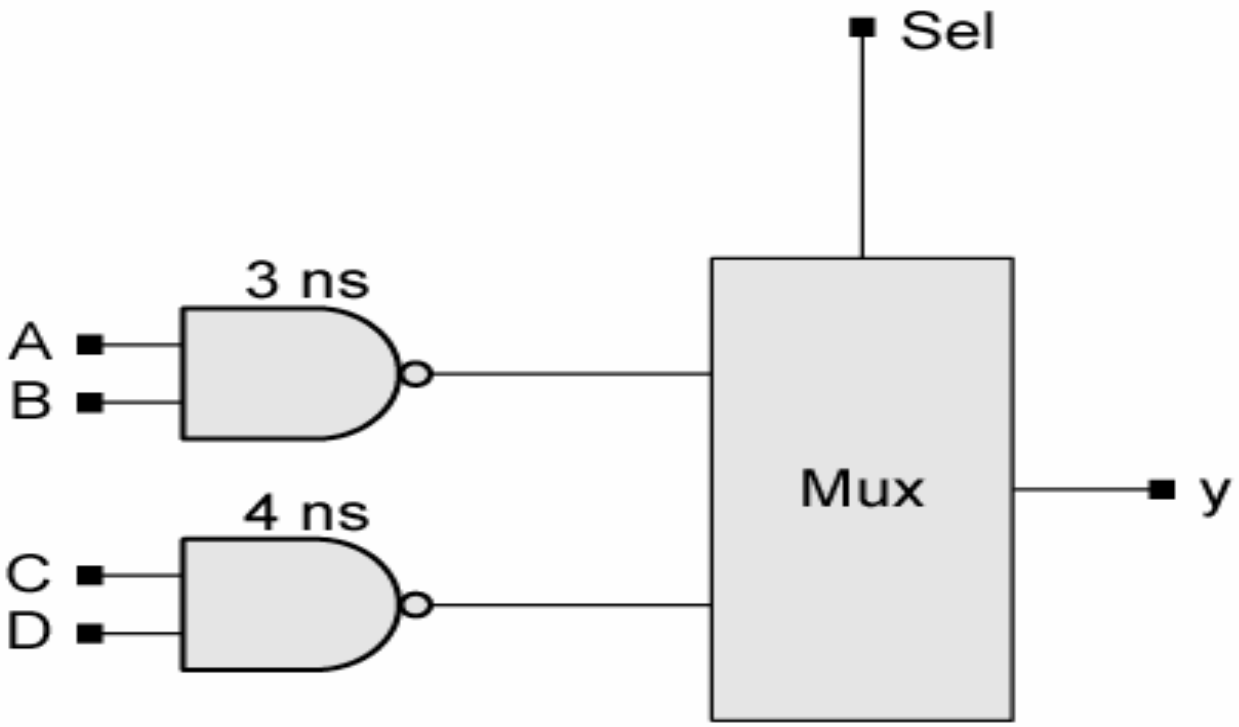
```
primitive jk_prim (q_out,clk,j,k,preset,clear);
  output q_out;
  input  cl, j, k, preset, clear;
  reg    q_out;
  table
//  clk  j   k  pre  clr  state q_out/next_state
// preset logic
      ?   ?   ?   0   1  :  ?  :  1;
      ?   ?   ?   *   1  :  1  :  1;
// clear logic
      ?   ?   ?   1   0  :  ?  :  0;
      ?   ?   ?   1   *  :  0  :  0;
// normal clocking
```



```
//   clk   j   k   pre  clr   state q_out/next_state
     r     0   0   0    0   : 0   :  1 ;
     r     0   0   1    1   : ?   :  - ;
     r     0   1   1    1   : ?   :  0 ;
     r     1   0   1    1   : ?   :  1 ;
     r     1   1   1    1   : 0   :  1 ;
     r     1   1   1    1   : 1   :  0 ;
     f     ?   ?   ?    ?   : ?   :  - ;
// j and k cases
//   clk   j   k   pre clr   state q_out/next_state
     b     *   ?   ?    ?   : ?   :  - ;
     b     ?   *   ?    ?   : ?   :  - ;
// reduced pessimism
     p     0   0   1    1   : ?   :  - ;
     p     0   ?   1    ?   : 0   :  - ;
     p     ?   0   ?    1   : 1   :  - ;
     p     ?   ?   ?    ?   : ?   :  - ;
     p     0   0   1    1   : ?   :  - ;
     p     0   ?   1    ?   : 0   :  - ;
     p     ?   0   ?    1   : 1   :  - ;
     p     *   0   ?    1   : 1   :  - ;
     p     0   *   1    ?   : 0   :  - ;
  endtable
endprimitive
```

Problem 4-15

- Write and verify a structural (gate-level) description of the circuit shown in Figure P4-15



Problem 4-16

- Write and verify a Verilog module of a bidirectional 8-bit ring counter capable of counting in either direction, beginning with first active clock edge after reset