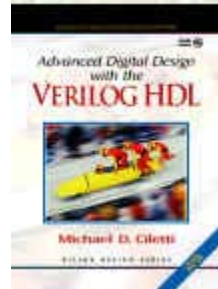


Advanced Digital Design with the Verilog HDL



M. D. Ciletti

Department
of

Electrical and Computer Engineering
University of Colorado
Colorado Springs, Colorado

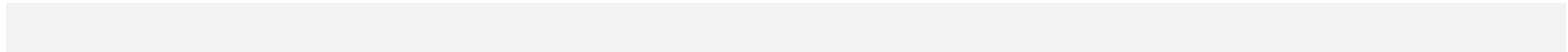
ciletti@vlsic.uccs.edu

Draft: Chap 4: Intro Logic Design with Verilog (rev 9/17/2003)

Copyright 2000, 2002, 2003. These notes are solely for classroom use by the instructor. No part of these notes may be copied, reproduced, or distributed to a third party, including students, in any form without the written permission of the author.

Note to the instructor: These slides are provided solely for classroom use in academic institutions by the instructor using the text, *Advance Digital Design with the Verilog HDL* by Michael Ciletti, published by Prentice Hall. This material may not be used in off-campus instruction, resold, reproduced or generally distributed in the original or modified format for any purpose without the permission of the Author. This material may not be placed on any server or network, and is protected under all copyright laws, as they currently exist. I am providing these slides to you subject to your agreeing that you will not provide them to your students in hardcopy or electronic format or use them for off-campus instruction of any kind. Please email to me your agreement to these conditions.

I will greatly appreciate your assisting me by calling to my attention any errors or any other revisions that would enhance the utility of these slides for classroom use.

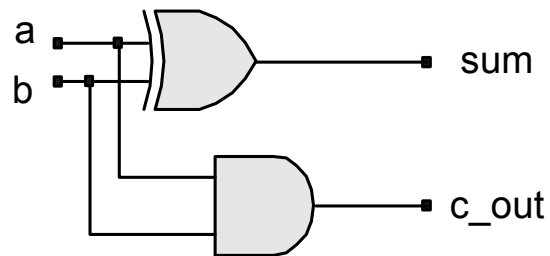


COURSE OVERVIEW

- Review of combinational and sequential logic design
- Modeling and verification with hardware description languages
- Introduction to synthesis with HDLs
- Programmable logic devices
- State machines, datapath controllers, RISC CPU
- Architectures and algorithms for computation and signal processing
- Synchronization across clock domains
- Timing analysis
- Fault simulation and testing, JTAG, BIST

Introductory Example: Half Adder

- Verilog primitives encapsulate pre-defined functionality of common logic gates
- The counterpart of a schematic is a structural model composed of Verilog primitives



```
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
```

Primitives

Verilog has 26 built-in primitives (combinational)

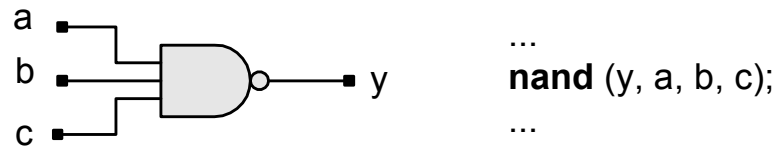
n-Input	n-Output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif0

MODELING TIP

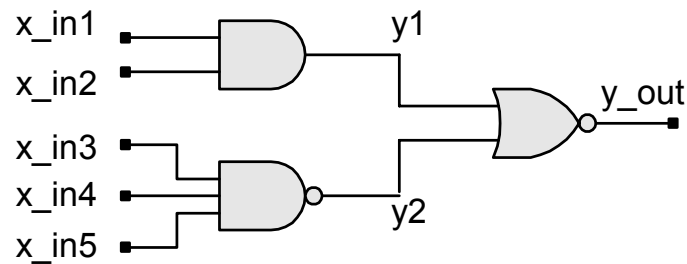
The output port of a primitive must be first in the list of ports.
The instance name of a primitive is optional.

3-Input Nand

- Model structural detail by instantiating and connecting primitives



Structural Details:



```

wire    y1, y2;

nor     (y_out, y1, y2);
and     (y1, x_in1, x_in2);
nand    (y2, x_in3, x_in4, x_in5);

```

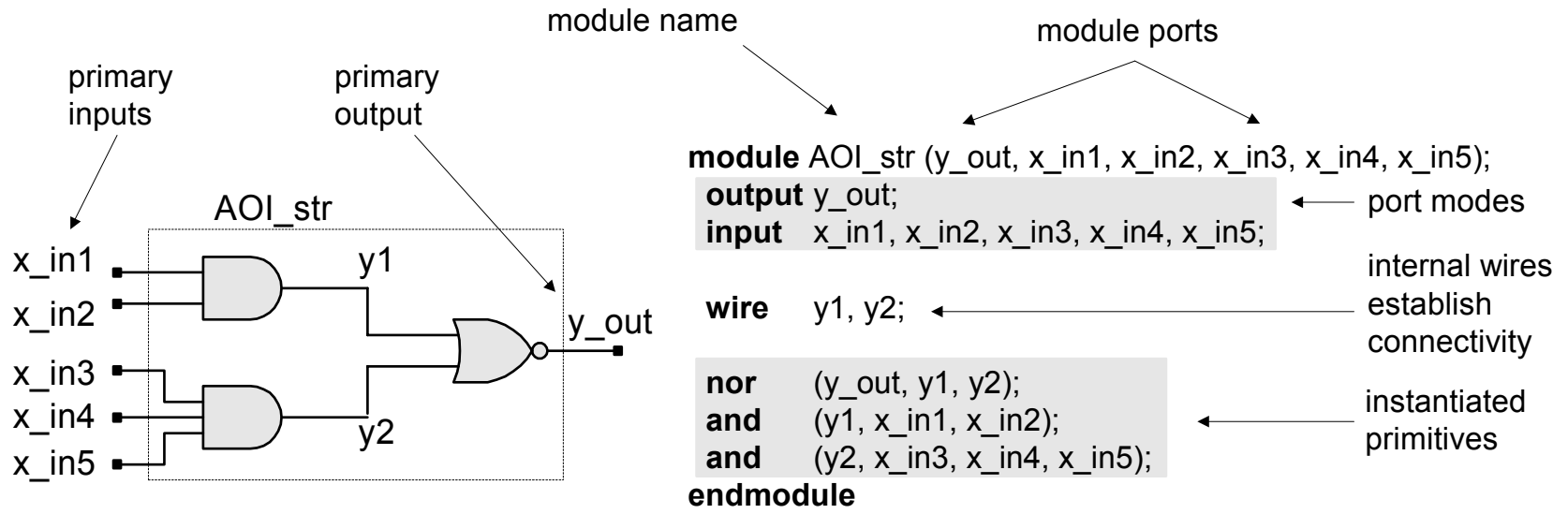
Design Encapsulation

- Encapsulate structural and functional details in a module

```
module my_design (module_ports);  
  
... // Declarations of ports go here  
... // Structural and functional details go here  
  
endmodule
```

- Encapsulation makes the model available for instantiation in other modules
- 

Structural Models

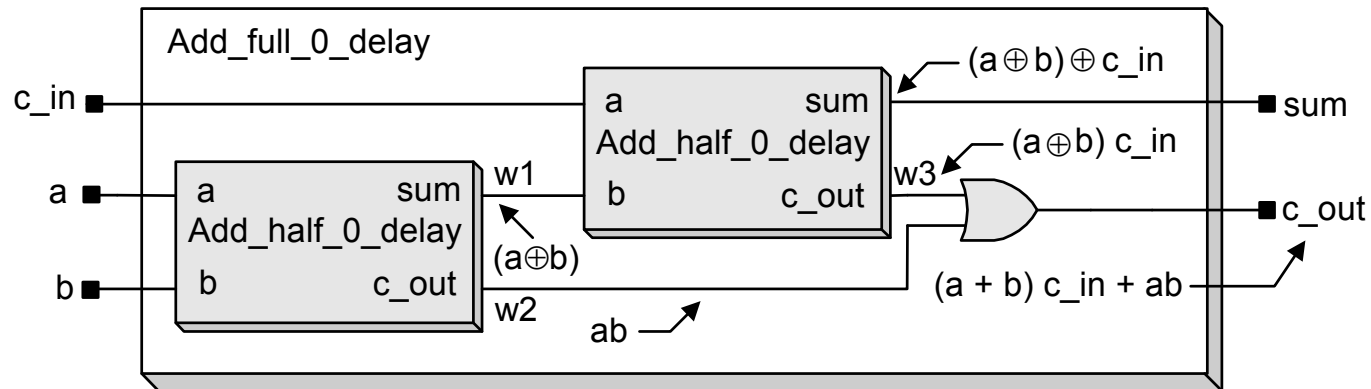


Language Rules

- Verilog is a case sensitive language (with a few exceptions)
- Identifiers (space-free sequence of symbols)
 - upper and lower case letters from the alphabet
 - digits (0, 1, ..., 9)
 - underscore (_)
 - \$ symbol (only for system tasks and functions)
 - Max length of 1024 symbols
- Terminate lines with semicolon
- Single line comments: // A single-line comment goes here
- Multi-line comments: /* Do not /* nest multi-line comments*/ like this */

Nested Modules

- Model complex structural detail by instantiating modules within modules

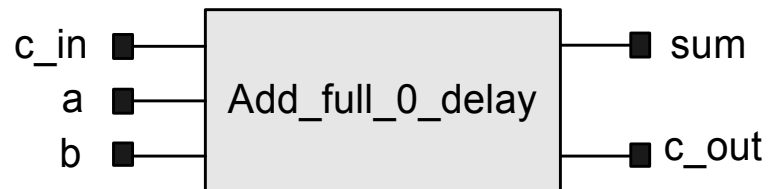


MODELING TIP

Use nested module instantiations to create a top-down design hierarchy.

Nested (Cont.)

Fig 4.6



```

module Add_full_0_delay (sum, c_out, a, b, c_in);
input  a, b, c_in;
output c_out, sum;
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

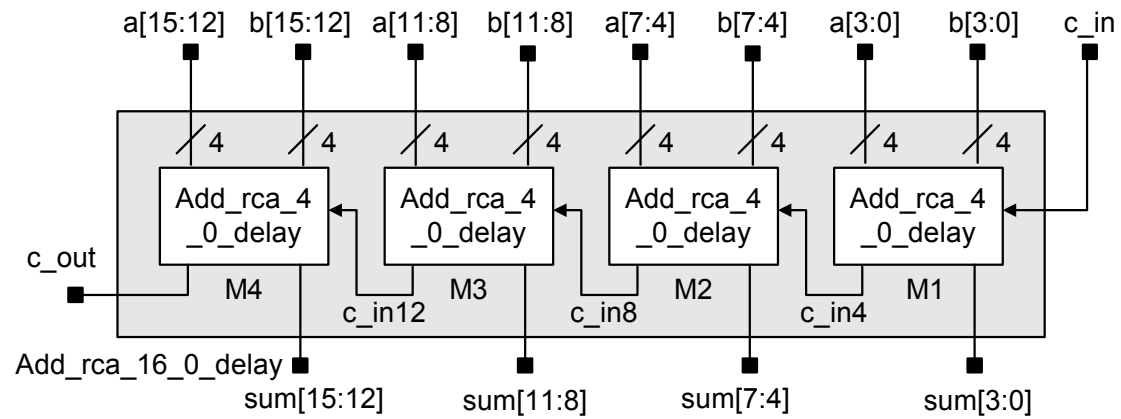
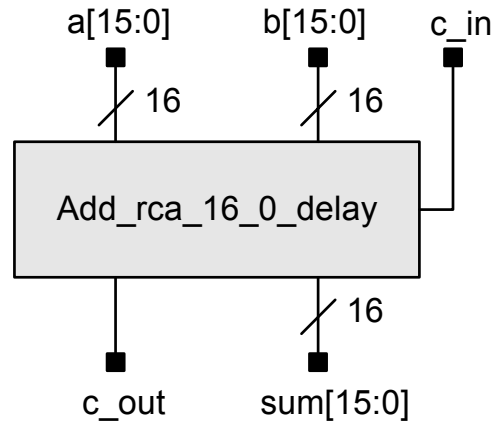
```

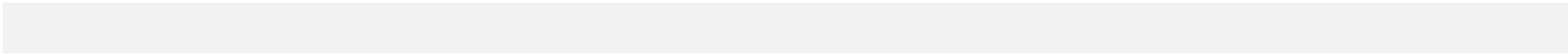
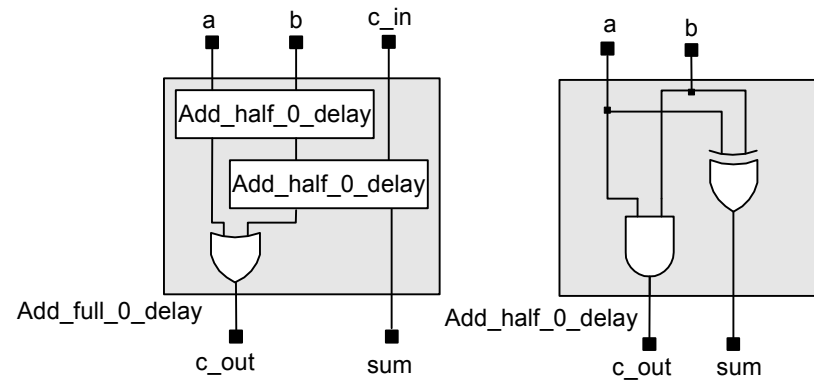
module instance name

MODELING TIP

The ports of a module may be listed in any order.
The instance name of a module is required.

Example: 16-bit Adder






Verilog Model: 16-bit Adder

```
module Add_rca_16_0_delay (sum, c_out, a, b, c_in);  
  output [15:0]    sum;  
  output          c_out;  
  input   [15:0]  a, b;  
  input          c_in;  
  wire         c_in4, c_in8, c_in12, c_out;  
  
  Add_rca_4 M1    (sum[3:0],    c_in4,    a[3:0],    b[3:0],    c_in);  
  Add_rca_4 M2    (sum[7:4],    c_in8,    a[7:4],    b[7:4],    c_in4);  
  Add_rca_4 M3    (sum[11:8],   c_in12,   a[11:8],   b[11:8],   c_in8);  
  Add_rca_4 M4    (sum[15:12],  c_out,    a[15:12],  b[15:12],  c_in12);  
endmodule
```

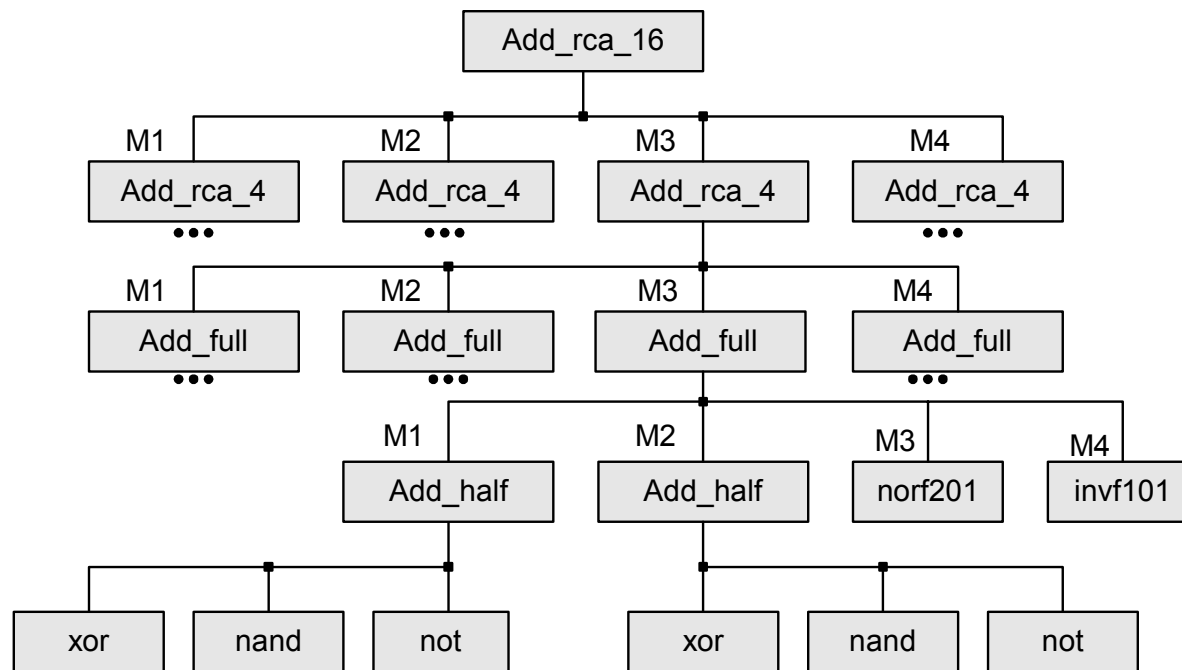
```
module Add_rca_4 (sum, c_out, a, b, c_in);  
  output [3: 0]    sum;  
  output          c_out;  
  input  [3: 0]    a, b;  
  input          c_in;  
  wire        c_in2, c_in3, c_in4;  
  
  Add_full M1 (sum[0],    c_in2,    a[0], b[0], c_in);  
  Add_full M2 (sum[1],    c_in3,    a[1], b[1], c_in2);  
  Add_full M3 (sum[2],    c_in4,    a[2], b[2], c_in3);  
  Add_full M4 (sum[3],    c_out,    a[3], b[3], c_in4);  
endmodule
```

```
module Add_full_0_delay(sum, c_out, a, b, c_in);  
  output          sum, c_out;  
  input           a, b, c_in;  
  wire            w1, w2, w3;  
  
  Add_half_0_delay M1 (w1, w2, a, b);  
  Add_half_0_delay M2 (sum, w3, c_in, w1);  
  or              M3 (c_out, w2, w3);  
endmodule
```

```
module Add_half_0_delay (sum, c_out, a, b);  
  output          sum, c_out;  
  input           a, b;  
  
  xor             M1 (sum, a, b);  
  and             M2 (c_out, a, b);  
endmodule
```



Design Hierarchy: 16-bit Adder



STRUCTURAL CONNECTIVITY

- Wires in Verilog establish connectivity between primitives and/or modules
- Data type: nets (Example: **wire**)
- The logic value of a **wire** (net) is determined dynamically during simulation by what is connected to the wire.

MODELING TIP

Use nets to establish structural connectivity.

MODELING TIP

An undeclared identifier is treated by default as a **wire**.

Port Connection By Name

- Connect ports by name in modules that have several ports

```
Add_half_0_delay M1 ( .b (b),  
                      .c_out (w2),  
                      .a (a),  
                      .sum (w1)  
                      );
```

actual name

formal name

Structural Model: 2-bit Comparator (p115)

Compare two 2-bit binary words:

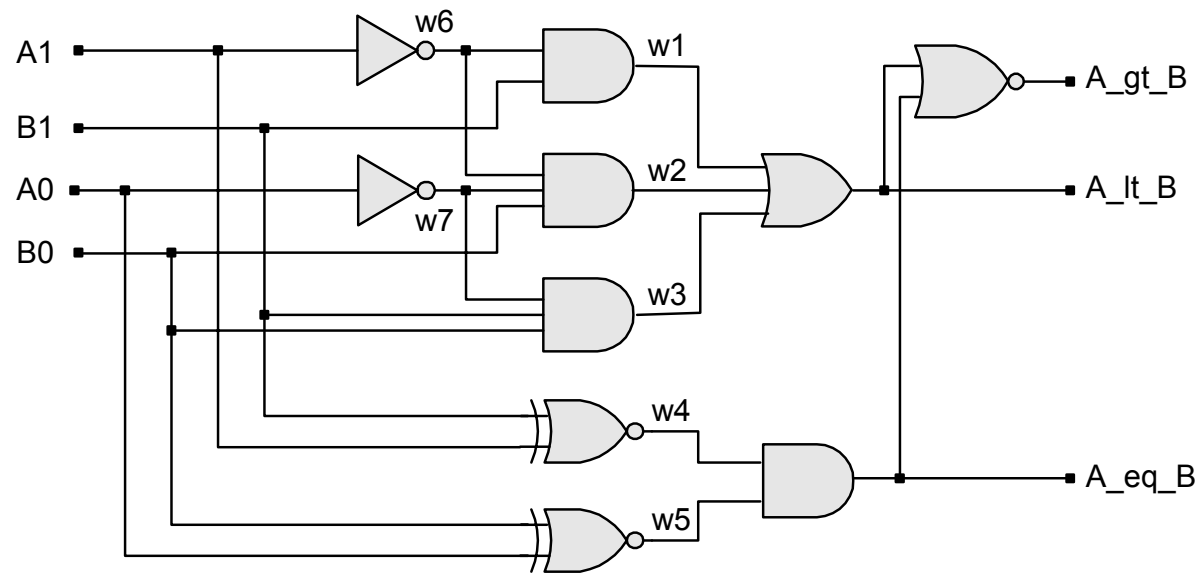
$$A_lt_B = A1' B1 + A1' A0' B0 + A0' B1 B0$$

$$A_gt_B = A1 B1' + A0 B1' B0' + A1 A0 B0'$$

$$A_eq_B = A1' A0' B1' B0' + A1' A0 B1' B0 + A1 A0 B1 B0 + A1 A0' B1 B0'$$

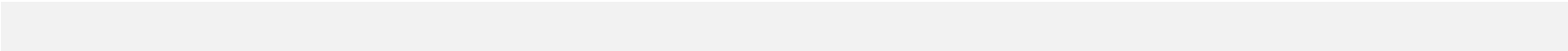
- Classical approach: use K-maps to reduce the logic and produce the schematic
- HDL approach: Connect primitives to describe the functionality implied by the schematic

- Schematic after minimization of K-maps:



Verilog (Structural) Model:

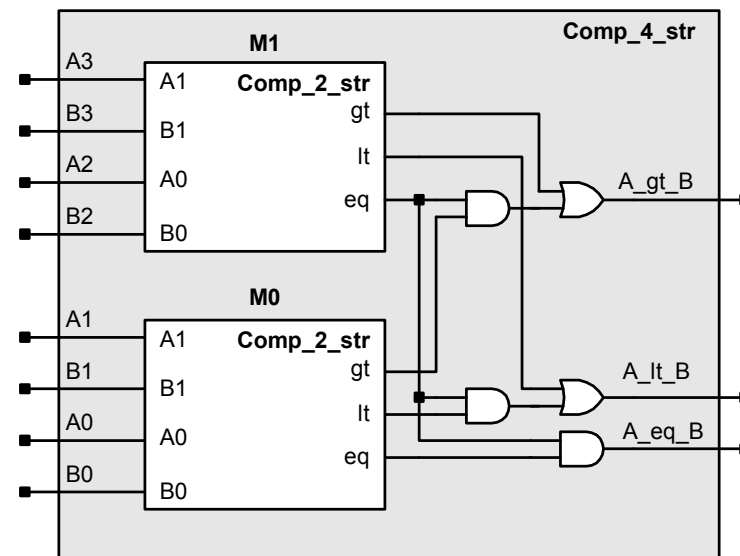
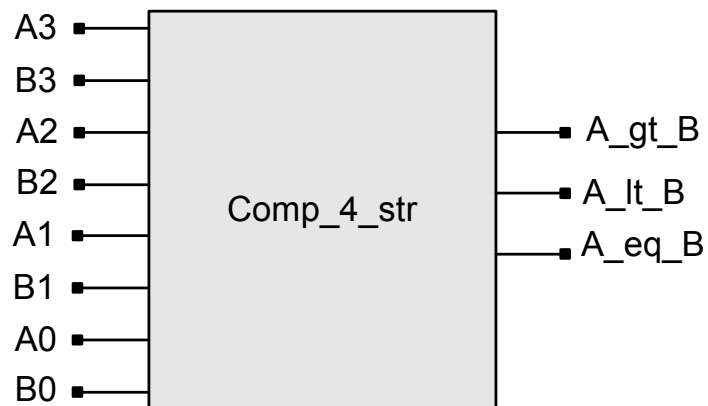
```
module compare_2_str (A_gt_B, A_lt_B, A_eq_B, A0, A1, B0, B1);  
output  A_gt_B, A_lt_B, A_eq_B;  
input   A0, A1, B0, B1;  
  // Note: w1, w2, ... are implicit wires  
  
  nor    (A_gt_B, A_lt_B, A_eq_B);  
  or     (A_lt_B, w1, w2, w3);  
  and    (A_eq_B, w4, w5);  
  and    (w1, w6, B1);  
  and    (w2, w6, w7, B0);  
  and    (w3, w7, B0, B1);  // Note: interchanging w7, B0 and B1 has no effect  
  not    (w6, A1);  
  not    (w7, A0);  
  xnor   (w4, A1, B1);  
  xnor   (w5, A0, B0);  
endmodule
```



Example: 4-bit Comparator

- Using a structure of 2-bit comparators, form a 4-bit comparator

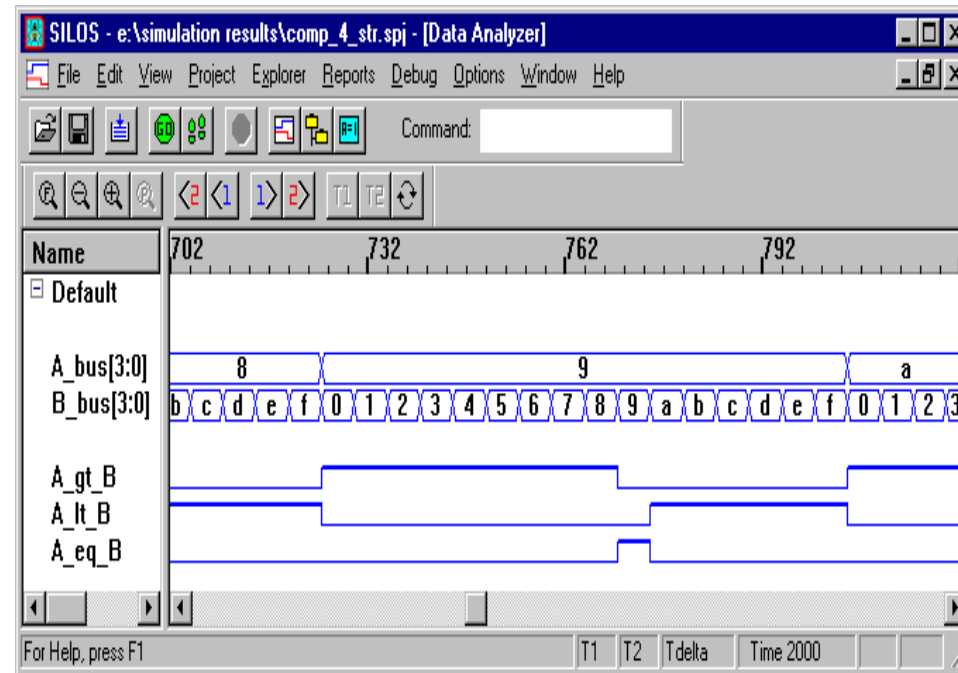
Note: A strict inequality in the higher order bit-pair determines the relative magnitudes of the 4-bit words; if the higher-order bit-pairs are equal, the lower-order bit-pairs determine the output.



Verilog Model:

```
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, B2, 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 M0 (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_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
```


Simulation Results:



Note: See the Silos –III tutorial at the web site: <http://ecweb.uccs.edu/ciletti>

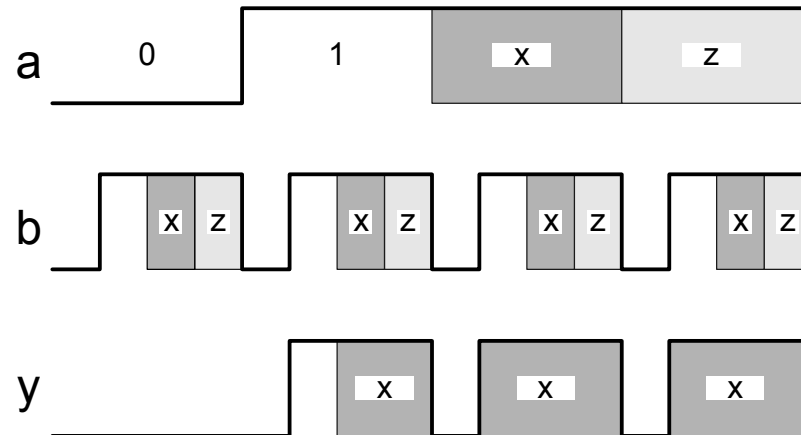
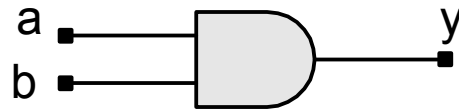
Logic System

- **Four values: 0, 1, x or X, z or Z** // Not case sensitive here
- **Primitives have built-in logic**
- **Simulators describe 4-value logic (see Appendix A in text)**

MODELING TIP

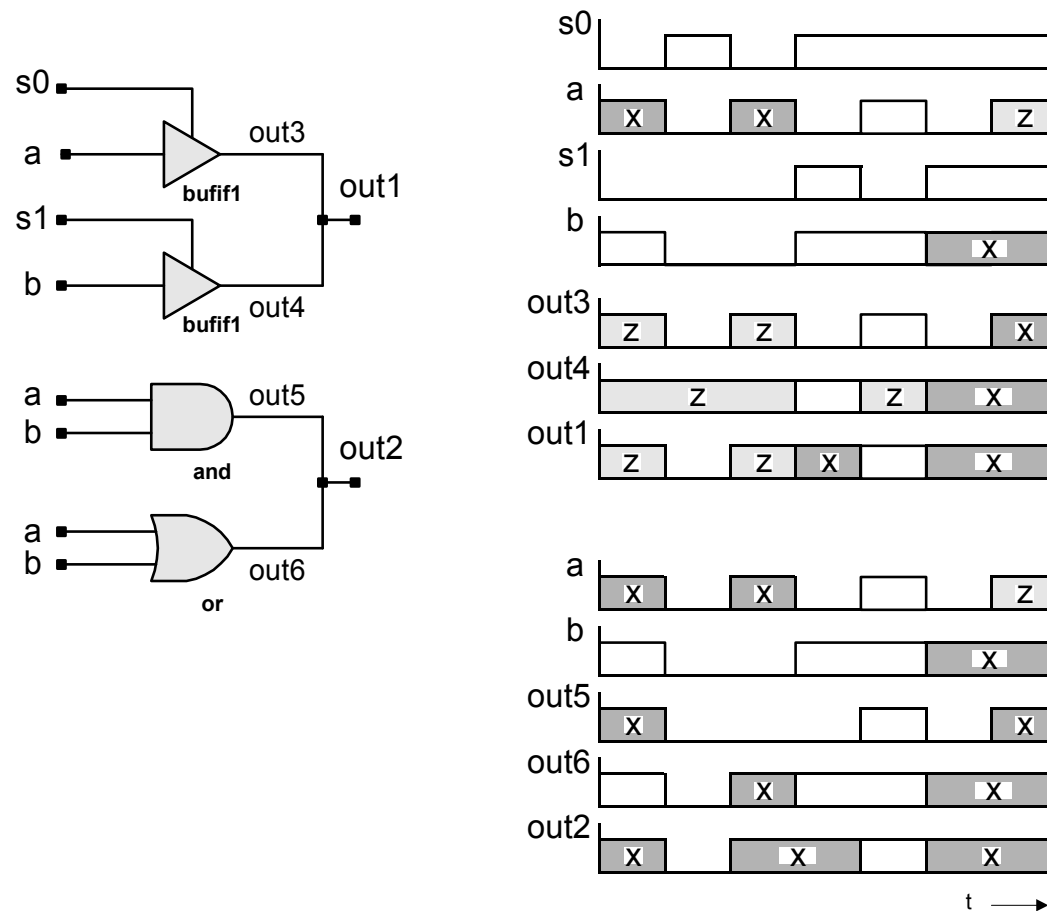
The logic value **x** denotes an unknown (ambiguous) value.
The logic value **z** denotes a high impedence.

Example: 4-Valued Logic



Resolution of Contention Between Drivers

- The value on a wire with multiple drivers in contention may be x



Wired Logic

- The family of nets includes the types **wand** and **wor**

A **wand** net type resolves multiple driver as wired-and logic

A **wor** net type resolves multiple drivers as wor logic

The family of nets includes **supply0** and **supply1**

supply0 has a fixed logic value of 0 to model a ground connection

supply1 has a fixed logic value of 1 to model a power connection

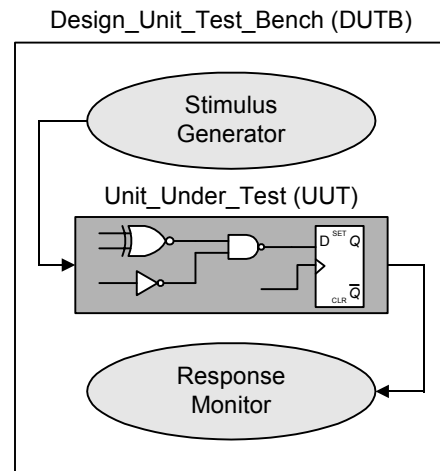
Test Methodology (p 122)

Task: systematically verify the functionality of a model.

Approaches: Simulation and/or formal verification

Simulation:

- (1) detect syntax violations in source code
- (2) simulate behavior
- (3) monitor results



Example: Testbench

```
module t_Add_half();  
  wire      sum, c_out;  
  reg       a, b;           // Storage containers for stimulus waveforms  
  
  Add_half_0_delay M1 (sum, c_out, a, b);      //UUT  
  
  initial begin              // Time Out  
    #100 $finish;           // Stopwatch  
  end  
  
  initial begin              // Stimulus patterns  
    #10 a = 0; b = 0;       // Statements execute in sequence  
    #10 b = 1;  
    #10 a = 1;  
    #10 b = 0;  
  end  
endmodule
```

Behaviors for Abstract Models

- Verilog has three types of behaviors for composing abstract models of functionality

Continuous assignment (Keyword: **assign**) - later

Single pass behavior (Keyword: **initial**) – Note: only use in testbenches

Cyclic behavior (Keyword: **always**) - later

- Single pass and cyclic behaviors execute procedural statements like a programming language
- The procedural statements execute sequentially
- A single pass behavior expires after the last statement executes
- A cyclic behavior begins executing again after the last statement executes

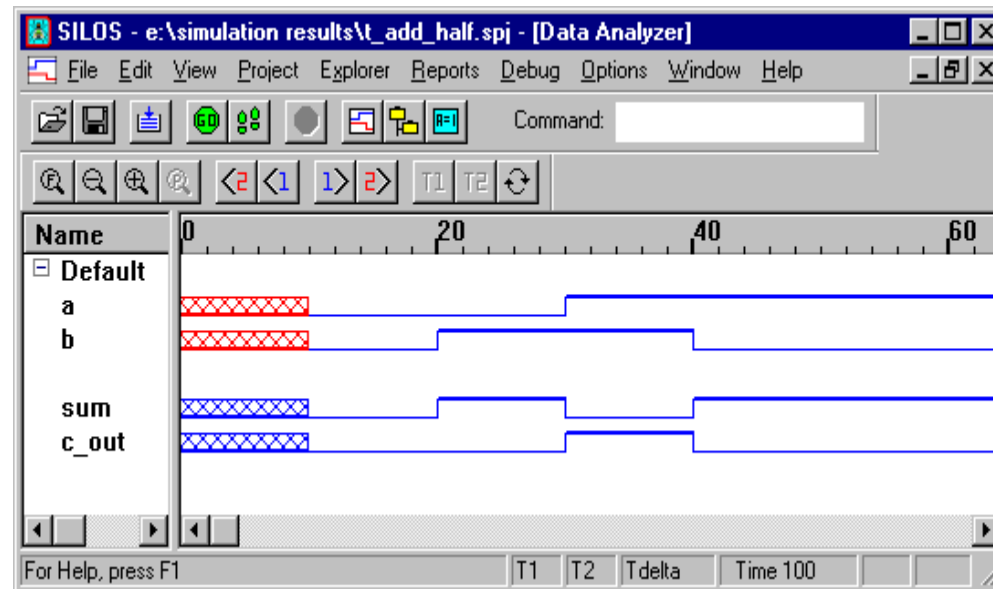
Signal Generators

- Use single-pass and cyclic behaviors to describe stimulus generators
- Statements in a behavior may be grouped in **begin ... end** blocks
- Execution begins at $t_{sim} = 0$
- # delay control operator temporarily suspends execution of a behavior
- The operator = denotes blocked procedural assignment

MODELING TIP

Use procedural assignments to describe stimulus patterns in a testbench.

Simulation Results for add_half:



MODELING TIP

A Verilog simulator assigns an *initial* value of **x** to all variables.

Event-Driven Simulation (p 125)

- A change in the value of a signal (variable) during simulation is referred to as an *event*
- Spice-like analog simulation is impractical for VLSI circuits
- Event-driven simulators update logic values only when signals change

Testbench template (p 125)

Consider the following template as a guide for simple testbenches:

```

module t_DUTB_name (); // substitute the name of the UUT
  reg ...; // Declaration of register variables for primary inputs of the UUT
  wire ...; // Declaration of primary outputs of the UUT
  parameter time_out = // 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; // (Also $finish) 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

```

Representation of Numbers (p 126)

- Sized numbers specify the number of bits that are to be stored for a value
- Base specifiers:
 - b or B binary
 - d or D decimal (default)
 - o or O octal
 - h or H hexadecimal

Examples (in-class exercise):

Note Unsized numbers are stored as integers (at least 32 bits)

Propagation Delay (p 126)

- Gate propagation delay specifies the time between an input change and the resulting output change
- Transport delay describes the time-of-flight of a signal transition
- Verilog uses an inertial delay model for gates and transport delay for nets
- Inertial delay suppresses short pulses (width less than the propdelay value)

MODELING TIP

All primitives and nets have a default propagation delay of 0.

Example: Propagation Delay

Unit-delay simulation reveals the chain of events

```

module Add_full (sum, c_out, a, b, c_in);

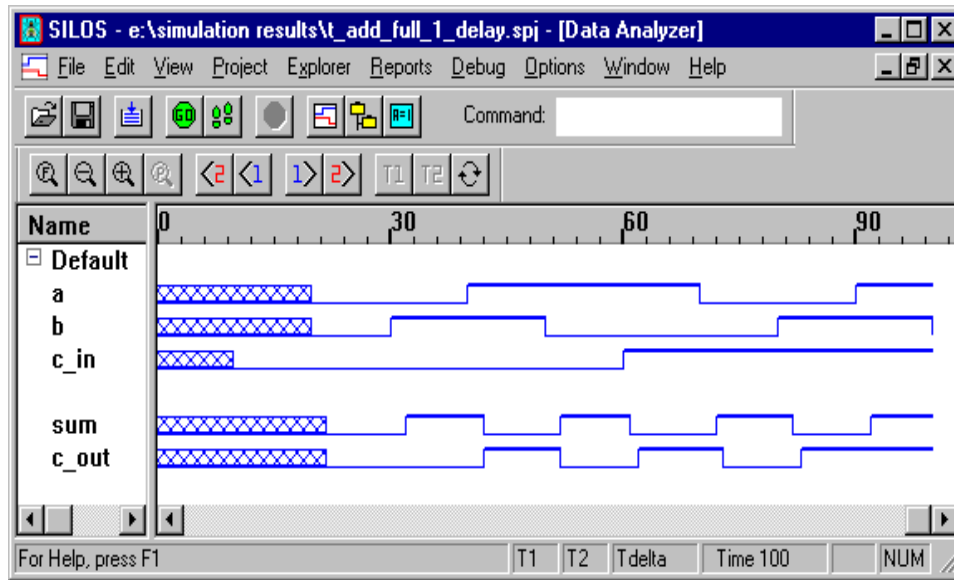
  output      sum, c_out;
  input       a, b, c_in;
  wire        w1, w2, w3;

  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);           // single delay value format
  and         #1 M2 (c_out, a, b);       // others are possible
endmodule

```



Simulation with Standard Cells

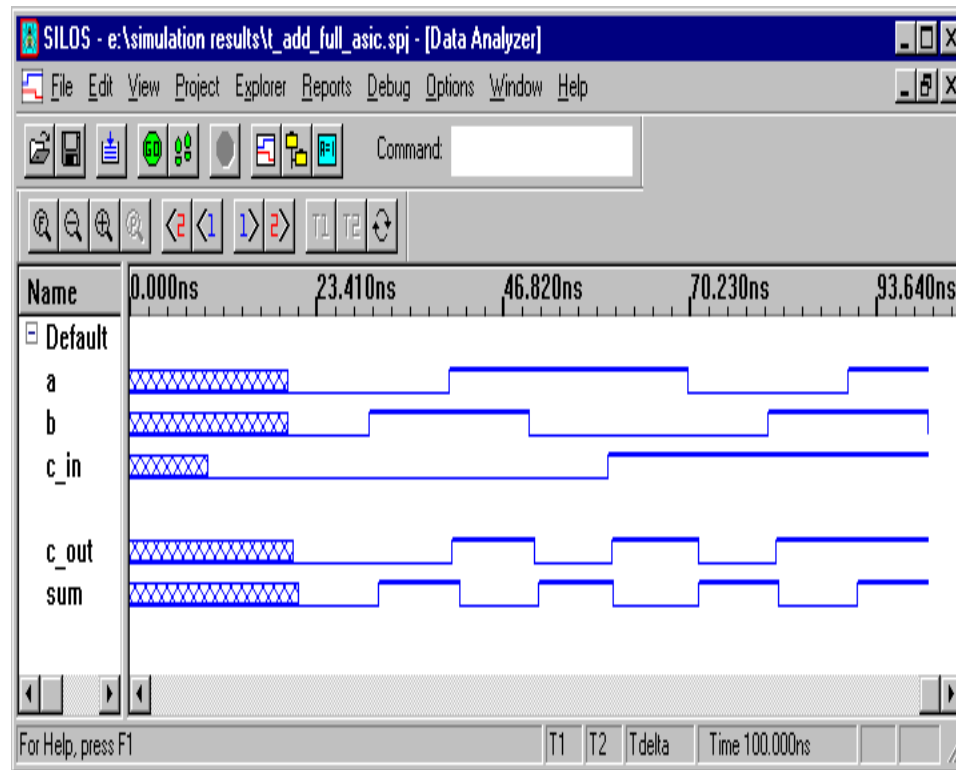
```
`timescale 1ns / 1 ps           // time scale directive for units and resolution
module Add_full_ASIC (sum, c_out, a, b, c_in);
  output      sum, c_out;
  input       a, b, c_in;
  wire        w1, w2, w3;
  wire        c_out_bar;

  Add_half_ASIC  M1 (w1, w2, a, b);
  Add_half_ASIC  M2 (sum, w3, w1, c_in);

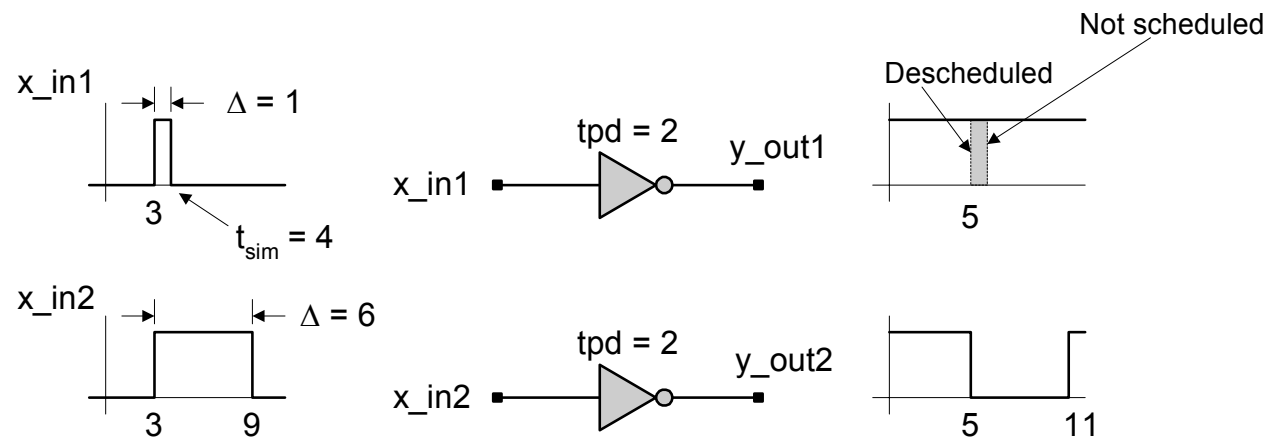
  norf201  M3 (c_out_bar, w2, w3);
  invf101  M4 (c_out, c_out_bar);
endmodule

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);           // Standard cells – down load from web page
nanf201 M2 (c_out_bar, a, b);
invf101 M3 (c_out, c_out_bar);
endmodule
```



Inertial Delay (p 131)



Note: The falling edge of x_{in1} occurs before the response to the rising edge occurs.

Truth-Tables Models and User-Defined Primitives (p 132)

- Built-in primitives are for simple combinational logic gates and CMOS transistors
- Primitives are memory efficient and simulate fast (good for ASIC libraries)
- User-defined primitives accommodate combinational and sequential logic
- Scalar output and multiple scalar inputs
- Arrange inputs columns of truth table in same order as ports
- Put output in last column, separated by :
- Use a UDP like a built-in primitive
- Table is searched top to bottom until match is found
- z may not be used in table (z in simulation is treated as x)
- No match results in propagation of x
- See web site for more details

```
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;
```

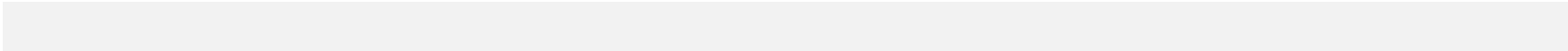
```
0 1 1 1 0 : 1;
```

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

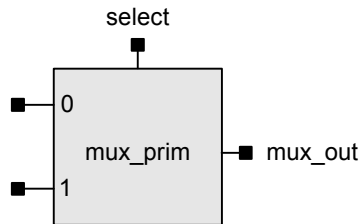
```
1 0 0 0 0 : 1;  
1 0 0 0 1 : 1;  
1 0 0 1 0 : 1;  
1 0 0 1 1 : 1;  
1 0 1 0 0 : 1;  
1 0 1 0 1 : 1;  
1 0 1 1 0 : 1;  
1 0 1 1 1 : 0;
```

```
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
```



Example: UDP



```
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 = 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
```

```
// select a b : mux_out
```



```
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 ; // Reduces pessimism  
x 1 1 : 1 ;
```

```
endtable  
endprimitive
```

```
// Note: Combinations not explicitly specified will drive 'x'  
// under simulation.
```

Alternative model using shorthand notation:

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



UDPs for Sequential Logic (p 135)

- Output is viewed as next state
- Insert a column for the present state truth
- Declare output to have type **reg**

MODELING TIP

The output of a sequential user-defined primitive must be declared to have type **reg**.

Example: Transparent Latch

```

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.

```

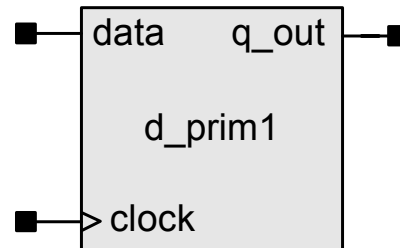
```

endtable
endprimitive

```

Example: D-Type Flip-Flop

- Notation for rising edge transition: (01), (0x), (x1)
- Notation for falling edge transition: (10), 1x), (x0)

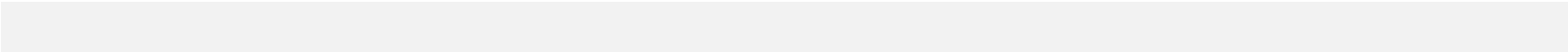


```
primitive d_prim1 (q_out, clock, data);  
  output q_out;  
  input clock, data;  
  
  reg q_out;  
  
  table  
  
  //   clk data : state : q_out/next_state
```

```
(01) 0 : ? : 0 ; // Rising clock edge
(01) 1 : ? : 1 ;
(0?) 1 : 1 : 1 ;

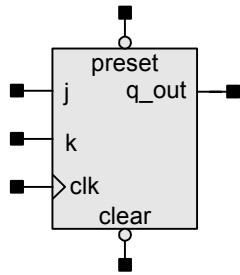
(?0) ? : ? : - ; // Falling or steady clock edge

? (??) : ? : - ; // Steady clock, ignore data
// transitions
endtable
endprimitive
```



Example: JK-Type Flip-Flop

- Level-sensitive and edge-sensitive behavior can be mixed in a UDP
- Place level-sensitive behavior at the top of the table



J-K Functionality:

- preset and clear override clock
- no change if $j = 0, k = 0$
- drive to 1 if $j = 1, k = 0$
- drive to 0 if $j = 0, k = 1$
- toggle if $j = 1, k = 1$

```

primitive jk_prim (q_out, clk, j, k, preset, clear);
  output  q_out;
  input   clk, 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	:-;
(?0)	?	?	?	?	:	?	:-;
(1x)	0	0	1	1	:	?	:-;
(1x)	0	?	1	?	:	0	:-;

(1x)	?	0	?	1	:	1	:	- ;
x	*	0	?	1	:	1	:	- ;
x	0	*	1	?	:	0	:	- ;

endtable
endprimitive

Note: * denotes any transition, and is equivalent to (??)