VERILOG

Hardware Description Language

Version: 9807

Chip Implementation Center

蔡慶宏
(03) 577-3693 # 144
chtsai@mbox.cic.edu.tw

李杰原
(03) 577-3693 # 154
leejy@mbox.cic.edu.tw

Table of Contents

Table of Contents (cont.)

Schedule

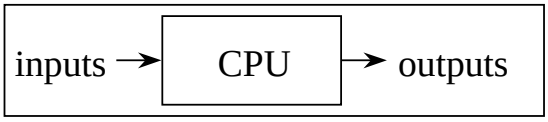
	Day 1	**Day 2**
9:00 --- 12:00	Introduction Sample Design Lexical Conventions Data Types and Logic System *Lab*	Behavior Modeling Interactive Debugging in Verilog-XL Modeling Memory User Defined Task and Function *Lab*
12:00 --- 13:00	**Lunch**	**Lunch**
13:00 --- 17:00	Structural Modeling Support for Verification Assignments *Lab*	Specify Blocks Modeling ASIC Libraries Simulation with Compass Cell Library *Lab*

Chapter 1

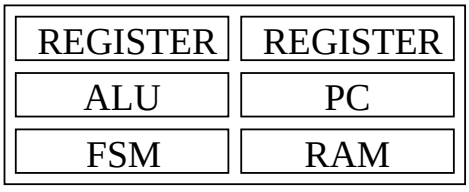
Introduction

Top Down ASIC Design Flow

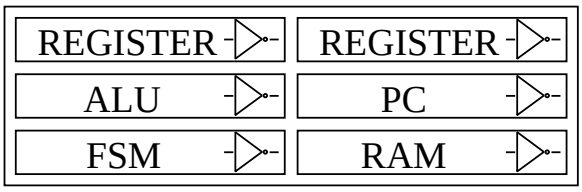
– Functionality
– performance
– data flow.



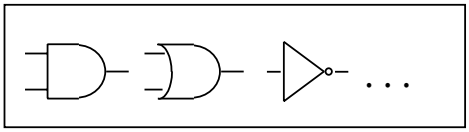
– Partitioning
– model the submodule at behavior level
– re-simulation



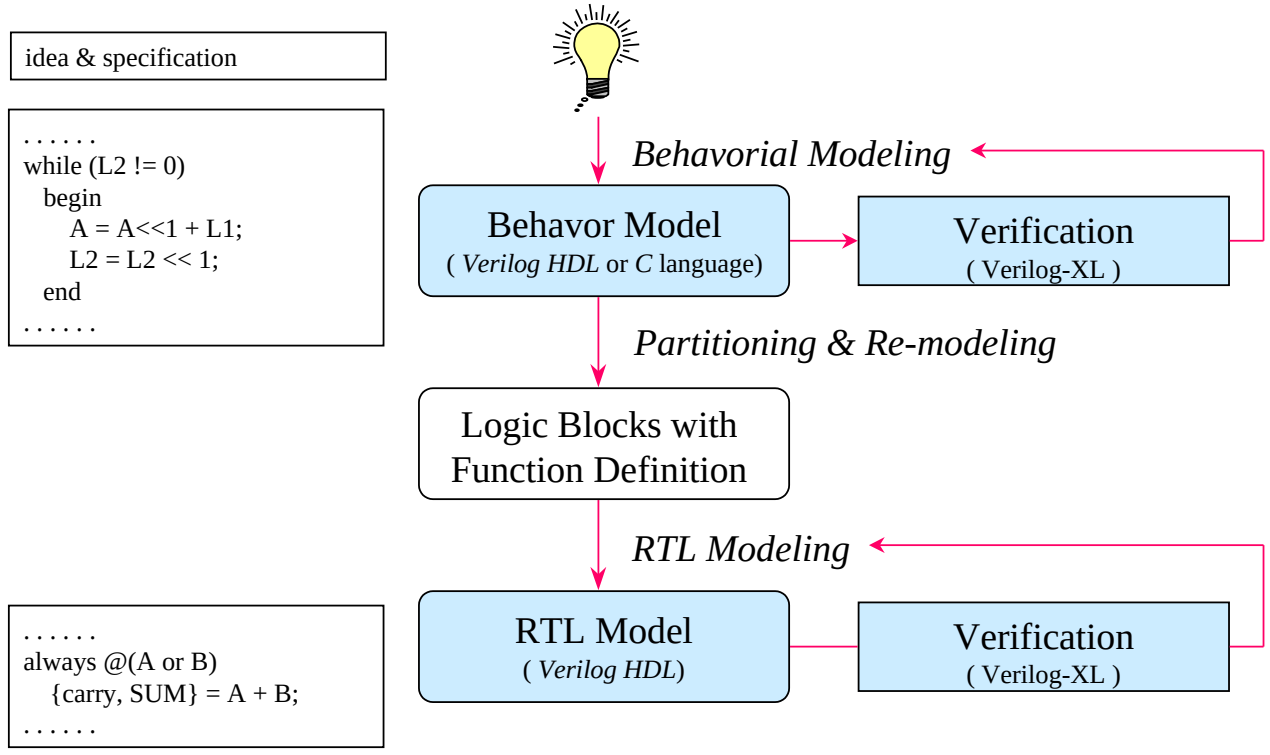
– re-model the submodule using library component
– re-simulation



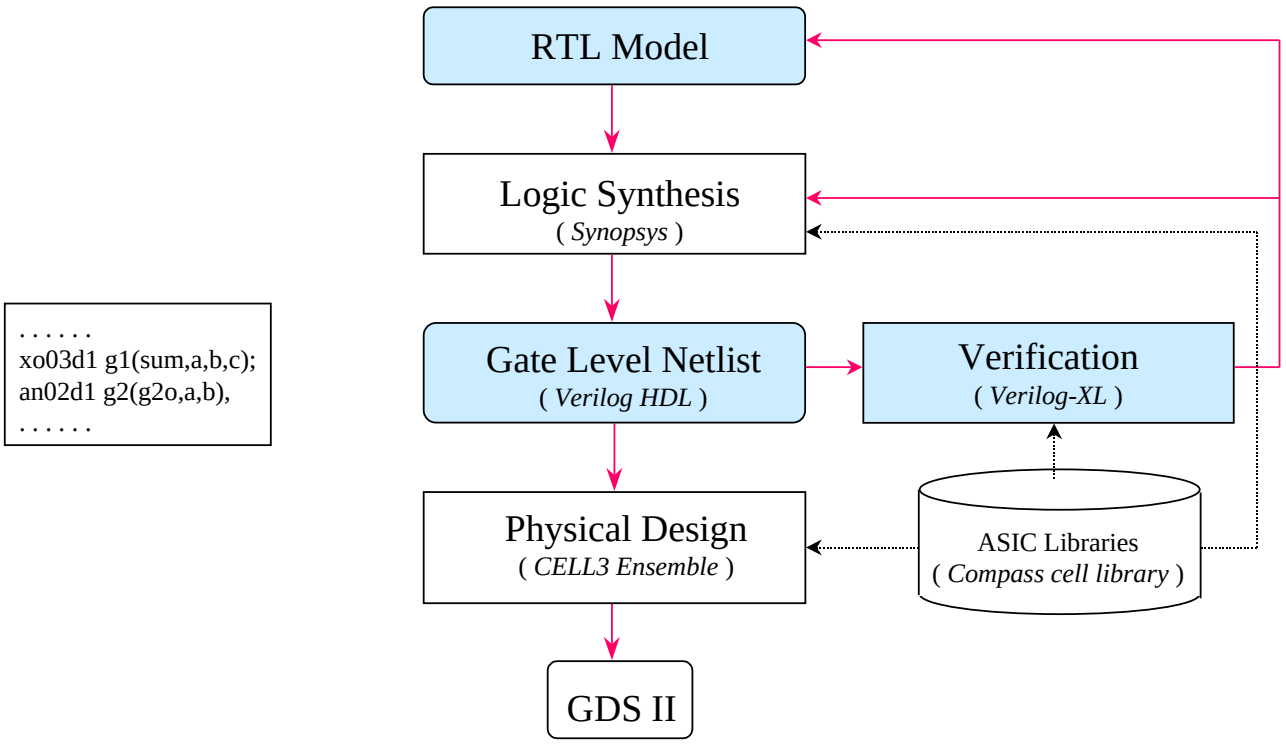
– Pre-designed and tested library component



Top Down ASIC Design Flow (cont.)



Top Down ASIC Design Flow (cont.)



Introduction to HDL

- **HDL -- Hardware Description Language**
- **Why use an HDL?**
  - It is becoming very difficult to design directly on hardware.
  - It is easier and cheaper to explore different design options.
  - Reduce time and cost.

Key Features of HDL

- **HDL has *high-level programming language* constructs and constructs to describe the *connectivity* of your circuit.**
- **Various levels of abstraction.**
- **Functionality as well as timing.**
- **Concurrency.**
- **Time.**

Different Levels of Abstraction

- **Architectural / Algorithmic**

  A model that implements a design algorithm in high-level language constructs.

- **Register Transfer Logic (RTL)**

  A model that describes the flow of data between registers and how a design process these data.

- **Gate**

  A model that describes the logic gates and the interconnections between them.

- **Switch**

  A model that describes the transistors and the interconnections between them.

Verilog HDL

- **Brief history of Verilog HDL**
  - 1985 : Verilog language and related simulator *Verilog-XL* were developed by *Gateway Automation*.
  - 1989 : *Cadence Design System* purchased *Gateway Automation.*
  - 1990 : *Cadence* released Verilog HDL to public domain.
  - 1990 : *Open Verilog Internaional* (OVI) formed.
  - 1995 : IEEE standard 1364 adopted.
- **Features**
  - Ability to mix different levels of abstraction freely.
  - One language for all aspects of design, test, and verification.

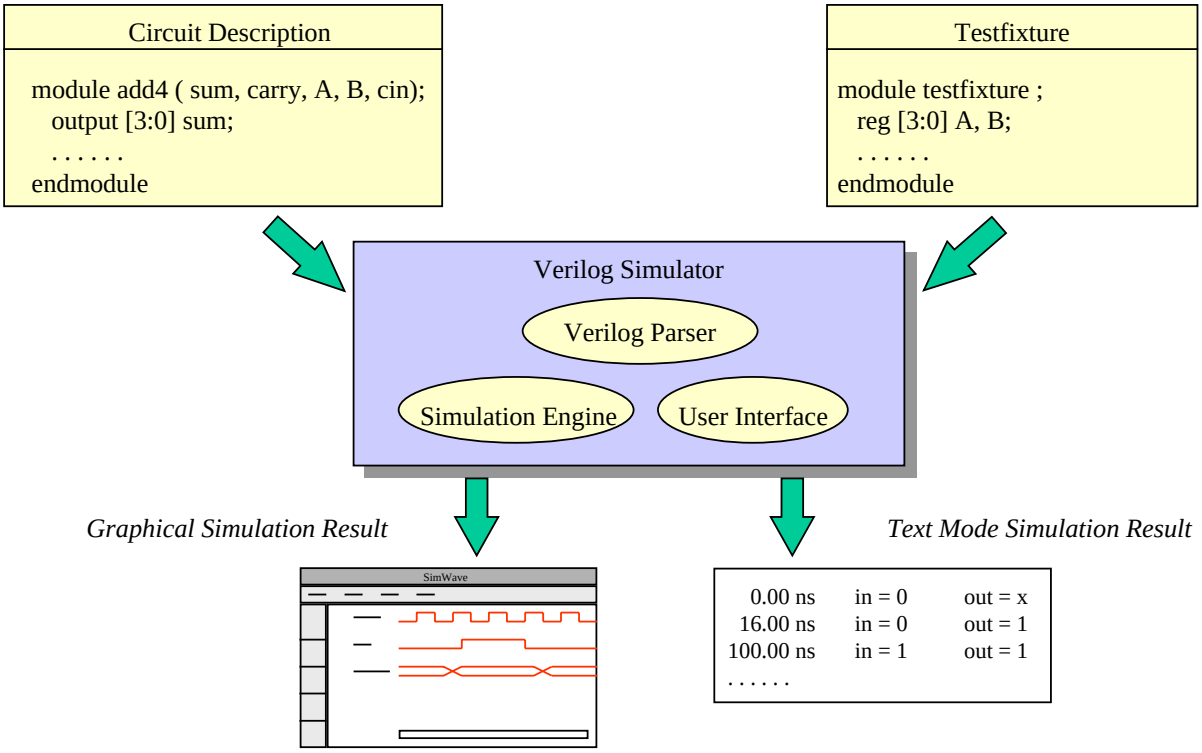
Verilog HDL Behavior Language

- **Structural and procedural like the *C* programming language.**
- **Used to describe algorithmic level and RTL level Verilog models.**
- **Key features**
  - procedural constructs for conditional, if-else, case, and looping operations.
  - arithmetic, logical, bit-wise, and reduction operations for expressions.
  - timing control.

Verilog HDL Structural Language

- **Used to describe gate-level and switch-level circuits.**
- **Key features**
  - a complete set of combinational primitives.
  - support primitive gate delay specification.
  - support primitive gate output strength specification.

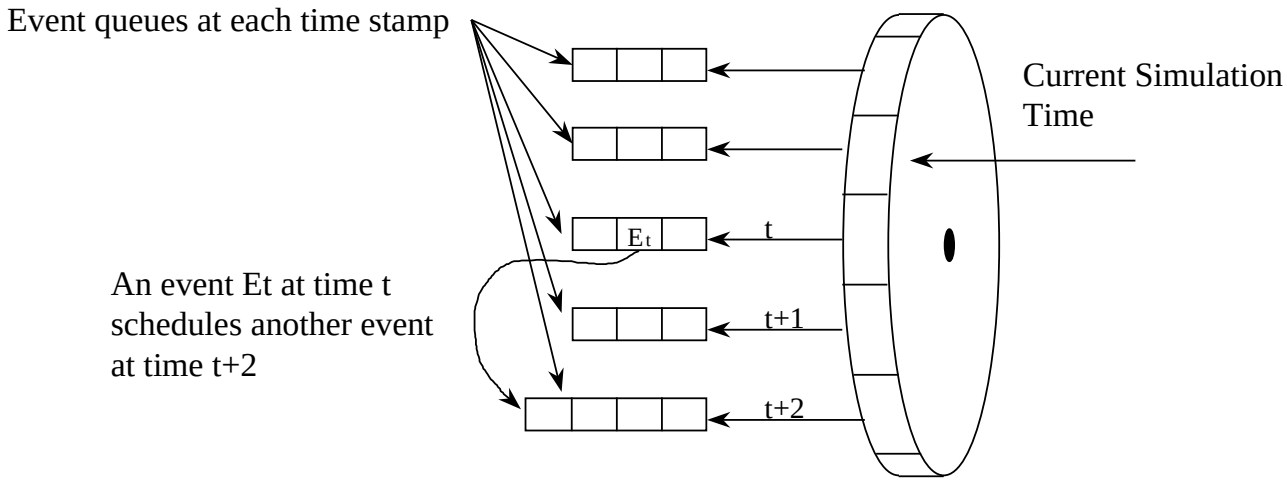
Verilog Simulator



Verilog-XL Logic Simulator

- **It reads Verilog HDL and simulates it.**
- **An event-driven logic simulator.**
  - only those elements that might cause a changes in circuit state, during a given simulation time, are simulated.
  - efficient, because it “evaluates when necessare”.

Time Wheel in Event-Driven Simulation



- **Time advances only when every event scheduled at that time is executed.**

Other Simulators

- **VCS (ViewLogic).**
  - Complete integration with *ViewSim* and *SpeedWave*.
  - Platform
    - – Windows 95 and Windows NT
    - – SunOS
  - Native code generation for rapid compilation.
- **NC-Verilog (Cadence Design System).**
  - Native compiled code software execution technique.
  - Reduced compilation time and execution time.
  - Platform
    - – Solaris

Summary

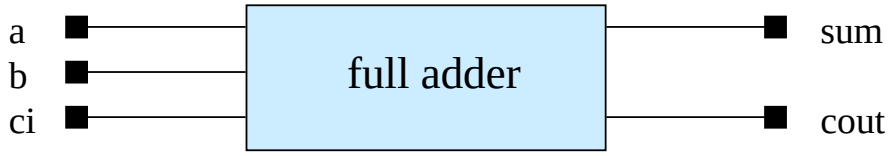
- **Top-down design flow.**
- **Verilog history.**
- **Verilog simulator.**

Chapter 2

Sample Design

A Full Adder Example

- **Symbol of a full adder.**

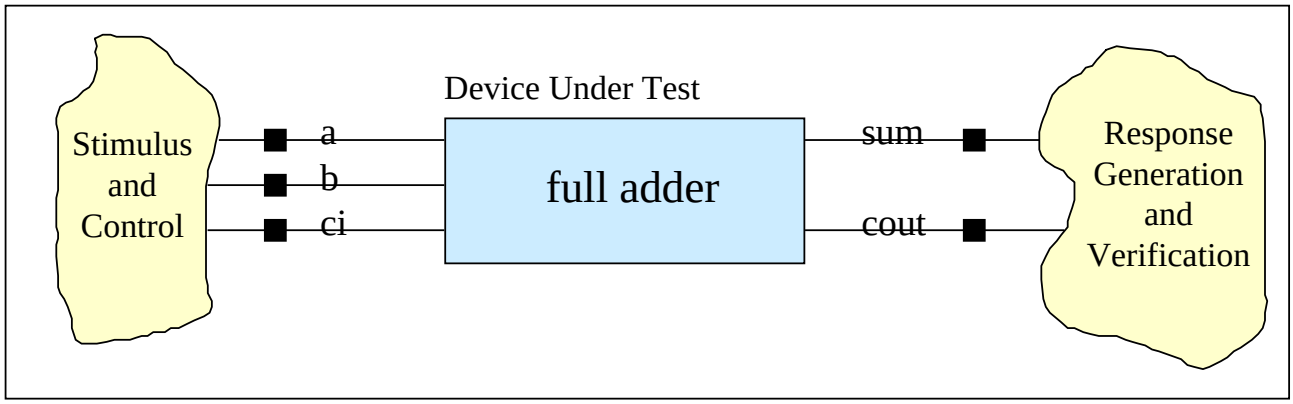


- **Expected behavior of a full adder.**

a	b	ci	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Testing and Verification of the Full Adder

Test Fixture



- **Test the full adder's Verilog model by applying test patterns and observing its output responses.**

Overview of Verilog Module

A Verilog module includes the following parts :

module *module_name* (*port_name*);

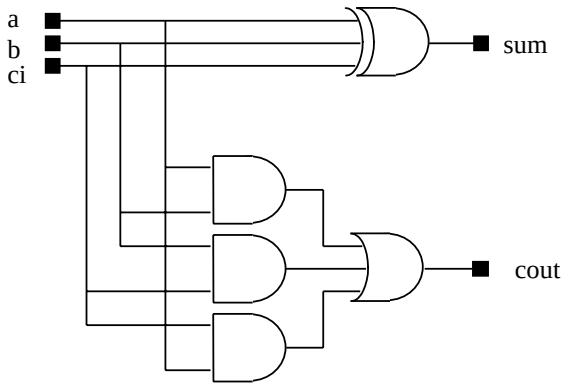
- *port declaration*
- *data type declaration*
- *task & function declaration*
- *module functionality or structure*
- *timing specification*

endmodule

Various Abstract of Full Adder

- **Gate level Verilog description of a full adder**

```
module fadder(sum,cout,a,b,ci);
  // port declaration
  output sum,cout;
  input  a,b,ci;

  // netlist description
  xor U0(sum, a, b, ci);
  and U1(net1, a, b);
  and U2(net2, b, ci);
  and U3(net3, ci, a);
  or   U4(cout, net1, net2, net3);
endmodule
```



Various Abstract of Full Adder (cont.)

- **Behavior level Verilog description of a full adder**

```
module fadder(sum,cout,a,b,ci);
  // port declaration
  output sum,cout;
  input  a,b,ci;
  reg    sum,cout;

  // behavior description
  always @(a or b or ci)
    begin
      sum = a ^ b ^ ci;
      cout = (a&b) | (b&ci) | (ci&a);
    end
endmodule
```

Whenever *a* or *b* or *c* changes its logic state, evaluate *sum* and *cout* by using the equation

$sum = a \oplus b \oplus ci$

$cout = ab + bc + ca$

Test Fixture Template

```
module testfixture;

 // Data type declaration

 // Instantitate modules

 // Applying stimulus

 // Display results

endmodule
```


Test Fixture --- Data Type Declaration

- **Test patterns must be first stored in storage elements and then applied to DUT (Device Under Test).**

```
module testfixture;

 // Data type declaration
 reg a, b, ci;
 wire sum, cout;

 // Instantitate modules

 // Applying stimulus

 // Display results
endmodule
```

Declare three storage element *a, b, c* to store the test pattern.

Test Fixture --- Making an Instance

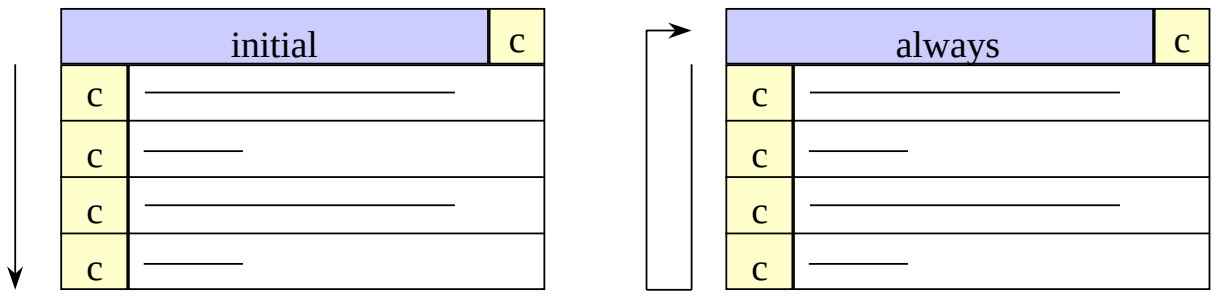
- **To make an instance is to instantiate a pre-defined module in current scope and connect its I/O port to other devices to make up a network.**

```
module testfixture;

  // Data type declaration
  reg a, b, ci;
  wire sum, cout;

  // Instantitate modules
  fadder adder_0(sum, cout, a, b, ci);

  // Applying stimulus

  // Display results
endmodule
```

- Instantiate a instance of full adder, where *adder_0* is this instance's name.
- Only module name and port declaration are needed by other modules that use this module. So you can treate it as a black box.
- You can use both structural model or behavior model to describe the submodule *fadder* .

Test Fixture --- Describing Stimulus

- **The testfixture will be described behaviorly.**
- **Procedural blocks are the bases of behavior modeling.**
- **Procedural blocks are of two types**
  - *initial* procedural blocks
  - *always* procedural blocks



Test Fixture --- Describing Stimulus (cont.)

```
module testfixture;
  // Data type declaration
    reg a, b, ci;
    wire sum, cout;
  // Instantitate modules
    fadder adder_0(sum, cout, a, b, ci);
  // Applying stimulus
    initial begin
      a = 0; b = 0; ci = 0;
      #10 ci = 1;
      #10 ci = 0; b = 1;
      #10 ci = 1;
      #10 a = 1; b = 0; ci = 0;
      #10 ci = 1;
      #10 ci = 0; b = 1;
      #10 ci = 1;
      #10 $finish;
      end
endmodule
```

Assign values to abstract storage elements.

#10 is used to specify 10 time unit delay.

$finish is a system task that ends the simulation.

Test Fixture --- Response Generation

- **Verilog provides you a set of system tasks and functions to display circuit output response.**
  - text format output.
    - $display(a,,b,,ci,,sum,,cout);
    - $monitor($time,,a,,b,,ci,,sum,,cout);
  - graphic output.
    - $gr_waves(“a”,a,”b”,b,”ci”,ci,”sum”,sum,”cout”,cout);
  - waveform display tool.
    - simWave.

A Complete Test Fixture

```
module testfixture;
  // Data type declaration
     reg a, b, ci;
     wire sum, cout;
  // Instantitate modules
     fadder adder_0(sum, cout, a, b, ci);
  // Applying stimulus
     initial begin
        a = 0; b = 0; ci = 0;
        #10 ci = 1;
        #10 ci = 0; b = 1;
        ......
        #10 $finish;
        end
  // Display result
   initial
     $monitor($time,"sum=%b cout=%b a=%b b=%b ci=%b",sum,cout,a,b,ci);
endmodule
```

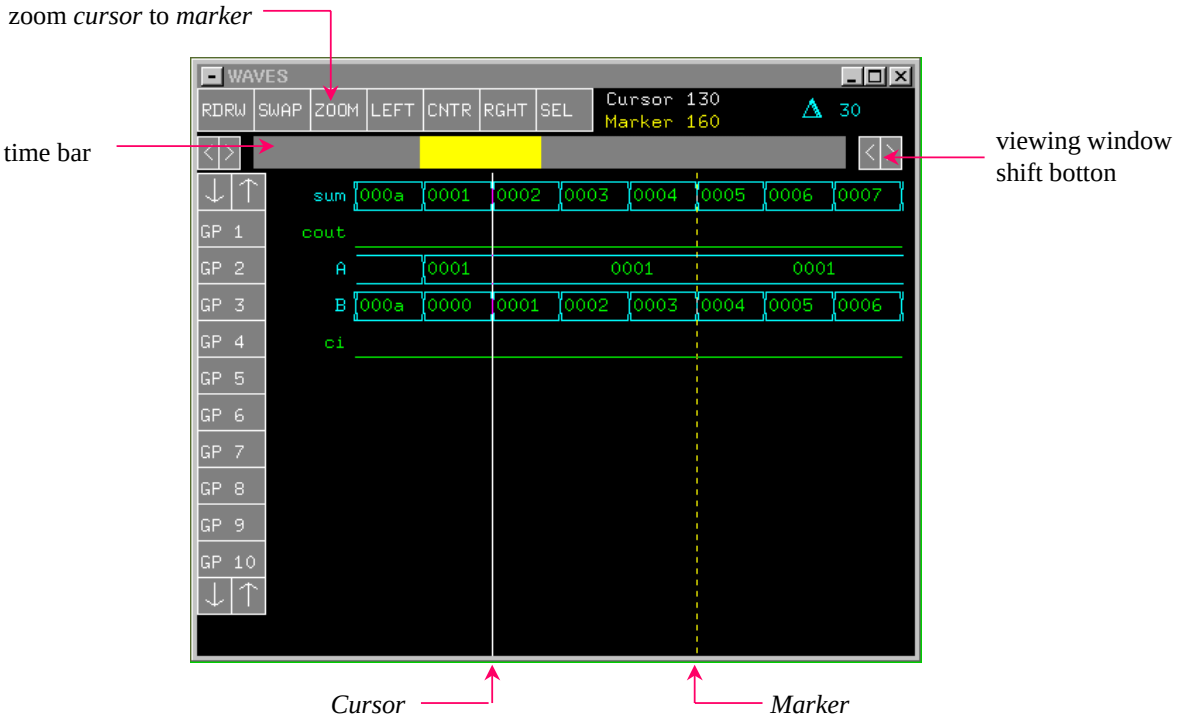
This will generate text output like

```
 0 sum=0 cout=0 a=0 b=0 ci=0
10 sum=1 cout=0 a=0 b=0 ci=1
.....
```


Graphic Output

- **Verilog-XL supports two types of graphical output :**
  - GR waves.
  - Simwave.
- **GR waves**
  - viewing simulation result while simulating.
- **Simwave**
  - viewing simulation result after simulation is completed.
  - you can re-check your last simulation result at any time.

GR Waves



GR Waves (cont.)

- **$*gr_waves* system task can generate graphic output.**

```
module testfixture;
  // Data type declaration
  ......
  // Instantitate modules
  ......
  // Applying stimulus
  ......
  // Display results
initial begin
     $gr_waves(”sum”, sum, ”cout”, cout, ”a”, a, ”b”, b, ”ci”, ci );
     #80 $stop;
     #1   $finish;
   end
endmodule
```

SimWave

- **Using system tasks to save the circuit state into waveform database.**
- **You can use SimW*ave* to view the waveforms after *Verilog-XL* simulation.**
- **Example**

```
module testfixture;
  ......
  // Display results
initial begin
     $shm_open("adder.shm");  // open waveform database " adder.shm ".
     $shm_probe("A");         // save the state of all nodes in current scope.
     #1   $stop;
   end
endmodule
```


Start the Sim*Wave*

- *SimWave* **allows you to display waveforms in a graphic window**
- **To start** *SimWave***, type** *simwave* **under unix prompt line.**

  unix % simwave &

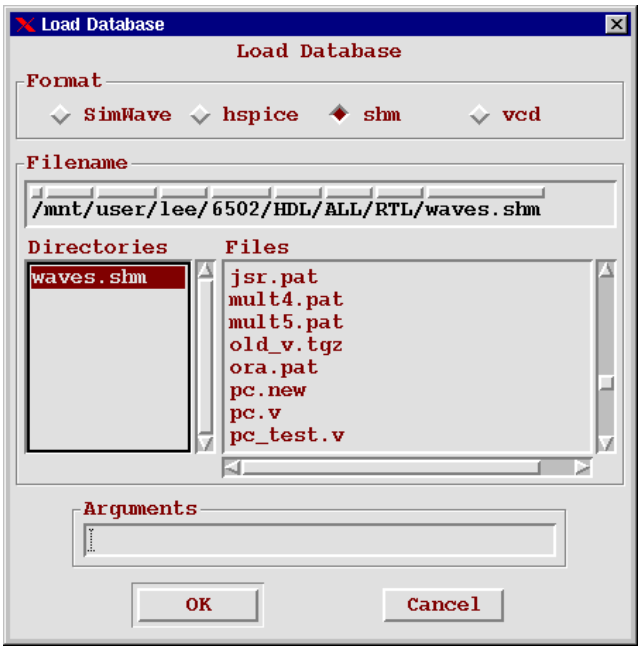
  Then the *SimWave* **graphic window will appear.**
- **Use the following menu command to load a pre-saved waveform database.**

  File -> Database -> Load

Load the SHM waveform

graphic database format

name of graphic database



Select Signals to watch

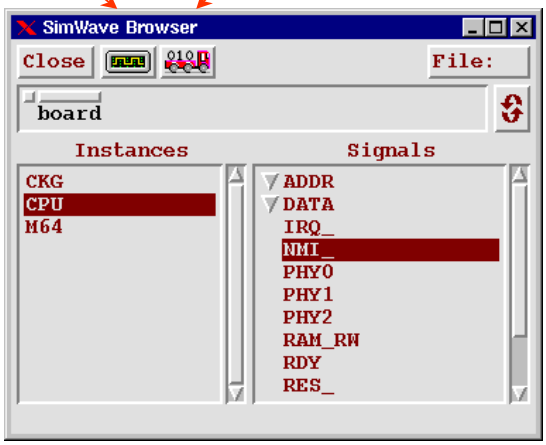
- **Use *Edit -> Add Signals* menu command to select the nodes that you want to observe.**

Display Signals

Create Bus

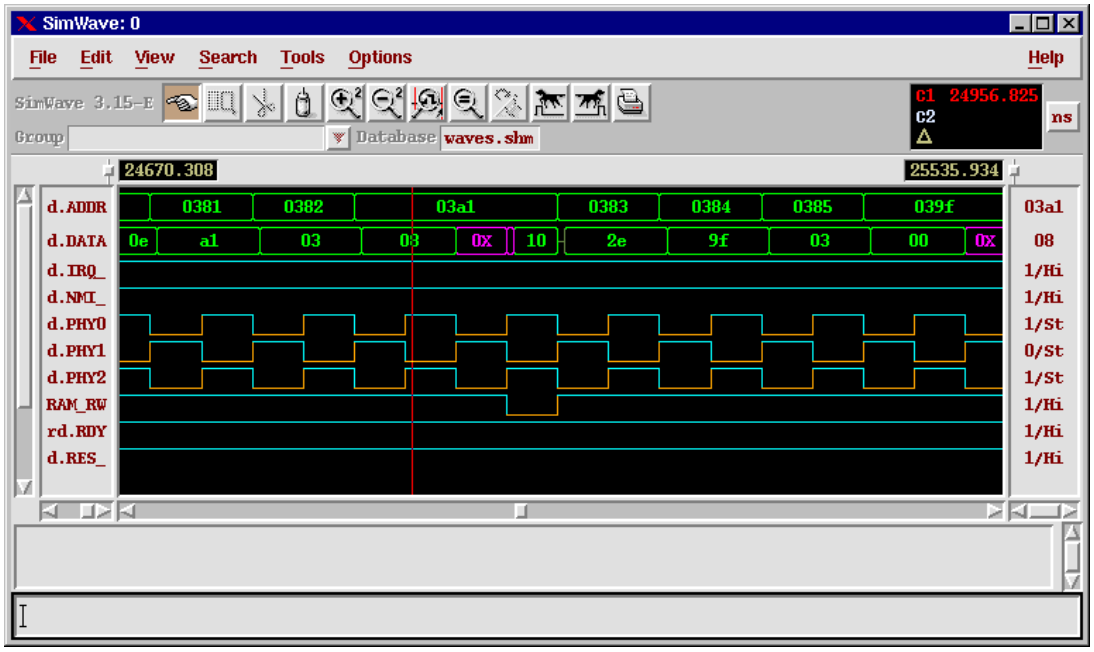
current scope

Modules in current scope



recorded signals in current scope

The SimWave Graphic Window



Strarting the Verilog-XL Simulation

- **Use the following UNIX command to start Verilog-XL simulation.**

  unix % verilog +lic_ncv adder.v

- **SYNTAX : verilog <command_line_options> <design_files>**
  - <command_line_options>
  - <design_files> are files that contain Verilog descriptions.

Strarting the Verilog-XL Simulation (cont.)

- **You can type *verilog* under UNIX to see various command line options and their corresponding actions.**

```
unix %  verilog
VERILOG-XL 2.2.1   Jul 27, 1996  12:36:50
Valid host command options:
  -f <filename>  read host command arguments from file
  -v <filename>  specify library file
.......
```

- **For example,**

unix % verilog +lic_ncv file1.v file2.v file3.v

is equivalent to

unix % verilog -f run.vc

run.vc

```
+lic_ncv
file1.v
file2.v
file3.v
```


Summary

- **Overview of Verilog simulation.**
- **Applying test patterns to your circuit.**
- **Viewing your simulation result.**
  - text mode output.
  - graphical output.

Chapter 3

Lexical Conventions in Verilog

Lexical Conventions in Verilog

Objectives

- Understand the lexical conventions used in the Verilog language.
- Learn to recognize special language tokens.

Lexical Conventions

- **Verilog is a free-format language.**
- **Verilog language source files are a stream of lexical tokens.**
- **Types of lexical tokens:**
  - operators
  - white space
  - comment
  - number
  - string
  - identifier
  - keyword

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Equality Operators	==, !=, ===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^, ~^
Unary Reduction	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	?:
Concatenations	{}

White Spaces and Comments

```
/*
 2-to-1 multiplexer,
 out = a when sel = 0; out = b when sel = 1;
*/
module MUX_2_to_1(out, a, b, sel);
  output out;
  input   a, b, sel;

  // netlist
  not (sel_, sel);
  and (a1, a, sel_), (b1, b, sel);
  or   ( out, a1, a2);
endmodule
```

White space character **:**

- White space is used to make your Verilog source code more readable.
- White space character includes *blank space (space bars)*, *tabs*, and *carriage returns*.

Integer and Real Numbers

- **Numbers can be integers or real numbers.**
- **Integers can be sized or unsized. Sized integers can be represented as**

  <size>‘<base><value>

  where

  <size> is the size in bits.

  <base> can be b (binary), o (octal), d (decimal), or h (hexadecimal)

  <value> is any legal number in the selected base and x, z, ?
- **Real numbers can be represented in decimal or scientific format.**

Integer and Real Numbers (cont.)

- **Examples**

  16 --- 32 bits decimal

  8’d16

  8’h10

  8’b0001_0000

  8’o20

  32’bx --- 32 bits x

  2’b1? --- “?” represents a high impedance bit

  6.3

  5.3e-4

  6.2E3

Strings

- **Strings are enclosed in double quotes and must be specified on one line.**

  “this is a string”

- **String variable declaration.**

  reg [8*12] stringvar ;

  initial begin

  stringvar = “Hello world!”;

  end

- **Verilog supports normal escaped characters in *C* language.**

  \n, \t, \\, \”, %%

Identifiers

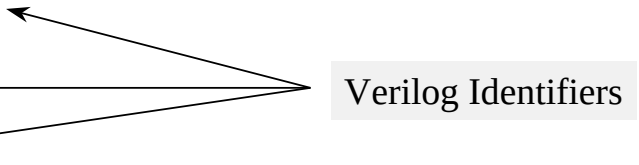
- **Identifiers are user-provided names for Verilog objects within a description.**
- **Legal characters in identifiers:**

  a-z, A-Z, 0-9, _, $
- **The first character of an identifier must be an alphabetical character (a-z, A-Z) or an underscore (_).**
- **Identifiers can be up to 1024 characters long.**

Identifiers (cont.)

- **Names of modules, ports, and instances are identifiers.**

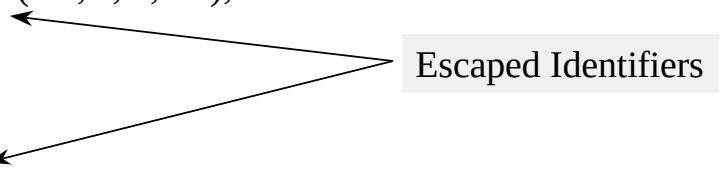
```
module MUX_2_to_1(out, a, b, sel);
  output out;
  input  a, b, sel;

  not  U0(sel_, sel);
  and  U1(a1, a, sel_),
       U2(b1, b, sel);
  or   U3( out, a1, a2);
endmodule
```



Escaped Identifiers

- **Escaped identifiers start with a backslash (\) and end with a white space.**
- **They can contain any printable ASCII characters.**
- **Backslash and white space are not part of the identifier.**

```
module \2:1MUX  (out, a, b, sel);
  output out;
  input   a, b, sel;

  not    U0(\~sel , sel);
  and   U1(a1, a, \~sel ),
        U2(b1, b, sel);
  or     U3( out, a1, a2);
endmodule
```



Keywords

- **Keywords are pre-defined non-escaped identifiers that are used to define the language construct.**
- **All keywords are defined in lower case.**
- **Examples:**

  module, endmodule
  input, output, inout
  reg, integer, real, time
  not, and, nand, or, nor, xor, ...
  parameter
  begin, end
  fork, join
  specify, endspecify
  .....

Case Sensitivity

- **Verilog is a case-sensitive language.**
- **You can run Verilog in case-insensitive mode by specifying *-u* command line option.**

case sensitive mode

```
module fadder(sum,cout,a,b,ci);
  output sum,cout;
  input  a,b,ci;
  reg    sum,cout;

  always @(a or b or ci)
    begin
      sum = a ^ b ^ ci;
      cout = (a&b) | (b&ci) | (ci&a);
    end
endmodule
```

case insensitive mode

```
module FADDER(SUM, COUT, A, B, CI);
      output SUM, COUT;
      input A, B, CI;
      reg SUM, COUT;

      always @(A or B or CI)
            begin
               SUM = A ^ B ^ CI;
               COUT = (A & B) | (B & CI) | (CI & A);
            end
  endmodule
```


Special Language Tokens

System Tasks and Functions

$<identifier>

- '$' sign denotes Verilog system tasks and functions.
- A number of system tasks and functions are available to perform different operations such as
  - Finding the current simulation time ($time).
  - Displaying/monitoring the values of signals ($display, $monitor).
  - Stopping the simulation ($stop).
  - Finishing the simulation ($finish).

Note : For more information on system tasks and functions, see the module "Support for Verification".

Special Language Tokens (cont.)

Delay Specification

#<delay specification>

- The pound sign (#) character denotes the delay specification for both gate instances and procedural statements.

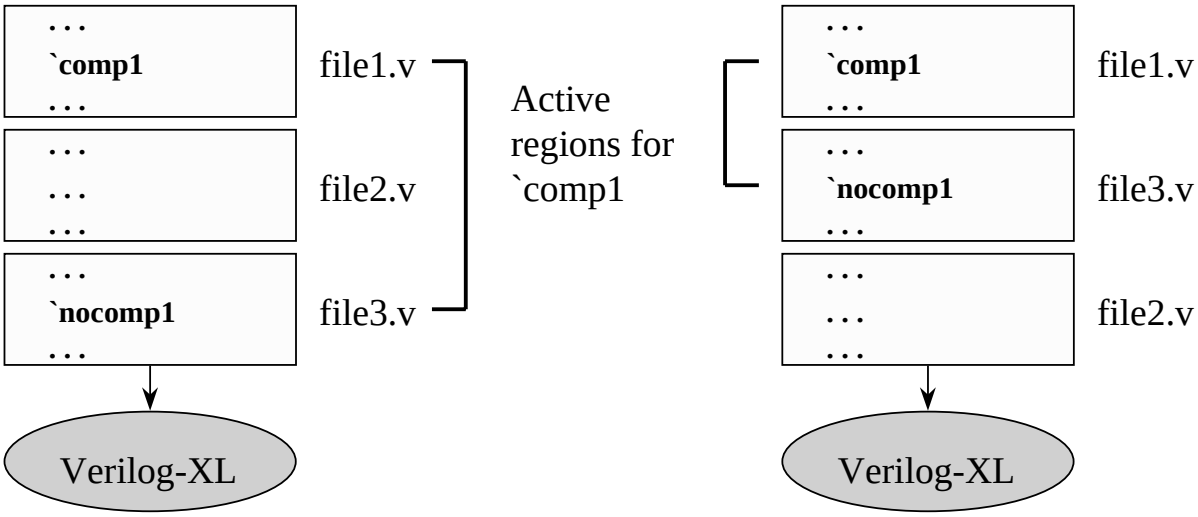
```
module MUX_2_to_1(out, a, b, sel);
   output out;
   input   a, b, sel;
   not   #1  not1(sel_, sel);
   and  #2  and1(a1, a, sel_), (b1, b, sel);
   or     #1  or1( out, a1, a2);
endmodule
```

```
module testfixture;
   reg a, b, ci;
initial begin
       #5    a = 0; b = 0; ci = 0;
       #10  ci = 1;
       . . . . . .
endmodule
```

Note : Delay specifications will be discussed in detail later**.**

Compilier Directives

- **All Verilog-XL compiler directives are preceded by the accent grave (`).**
- **Compiler directives remain active across source files.**



Compiler Directives (cont.)

- **Compiler directives remain active until they are overridden or deactivated.**
- **The `*resetall* compiler directive resets all the compiler directives to their default values (only if there is a default value).**

Text Substitution

- **The `*define* compiler directive provides a simple text-substitution facility.**

  `define <*macro_name*> <*text_string*>

  `<*text_string*> will substitute < *macro_name* > at compile time.
- **Typically use `*define* to make the description more readable.**

```
`define  HLT  3'h0
`define  SKZ  3'h1
.....
case (opcode)
   `HLT : .....
   `SKZ : .....
endcase
```

`*HLT* will be replaced as *3'h0* at compile time.

Text Inclusion

- **Use `*include* compiler directive to insert the contents of an entire file.**

  `include “global.v”

  `include “parts/count.v”

- **You can use `*include* to**
  - include global or commonly used definitions.
  - include tasks without encapsulating repeated code within module boundaries.

Timescale in Verilog

- **The `*timescale* compiler directive declares the time unit and its precision.**

  `timescale <time_unit> / <time_precision>

- **The *time_unit* argument specifies the unit of measurement for times and delays.**

```
`timescale 10ns / 1ps
module MUX_2_to_1(out, a, b, sel);
  output out;
  input  a, b, sel;
  not    #1  not1(sel_, sel);
  and    #2  and1(a1, a, sel_), (b1, b, sel);
  or     #1  or1( out, a1, a2);
endmodule
```

The intrinsic delay of this inverter is 10ns.

The intrinsic delay of this AND gate is 20ns.

Timescale in Verilog (cont.)

- **The *time_precision* argument determines how Verilog-XL rounds delay values before using them in simulation.**

```
`timescale 10ns / 1ns
module MUX_2_to_1(out, a, b, sel);
  output out;
  input  a, b, sel;
  not   #1.54  not1(sel_, sel);
  and   #2.55  and1(a1, a, sel_), (b1, b, sel);
  or    #1     or1( out, a1, a2);
endmodule
```

The intrinsic delay of this inverter is 15ns.

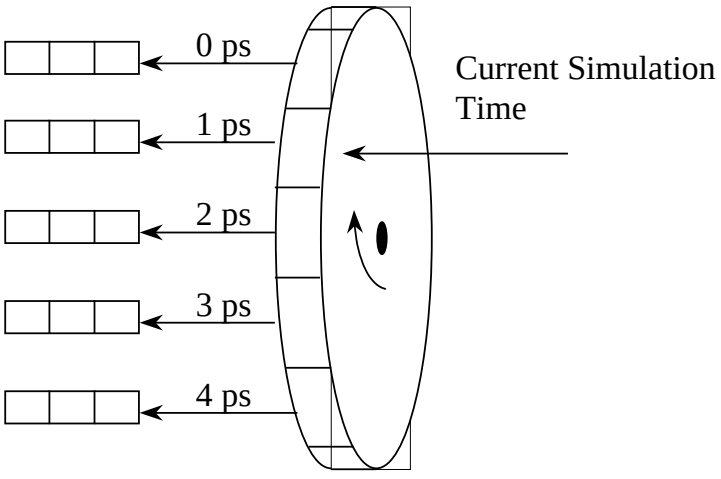
The intrinsic delay of this AND gate is 26ns.

Note : The `timescale compiler directive cannot appear inside a module boundary.

Timescale in Verilog (cont.)

- **The smallest precision of all the *`timescale* determines the time unit of simulation.**

```
`timescale 1ns / 10ps
module1 (. . .);
. . .
endmodule

`timescale 100ns / 1ns
module2 (. . .);
. . .
endmodule

`timescale 1ps / 1ps
module3 (. . .);
. . .
endmodule
```



Summary

- **Lexical conventions.**
- **Numbers.**
- **Identifiers and escaped identifiers.**
- **Special language tokens.**
  - system tasks
  - delay specification
  - compiler directives

Chapter 4

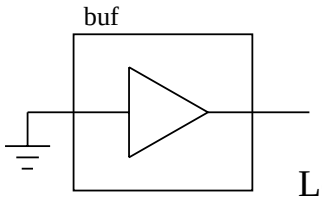
Verilog Data Types and Logic System

Verilog Data Types and Logic System

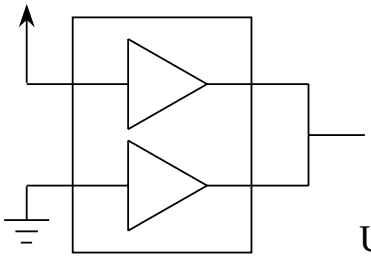
- **Data Type : Verilog use it to represent the data storage and transmission elements in digital system.**
- **The Verilog HDL logic system includes value set and strength information.**

Value Set

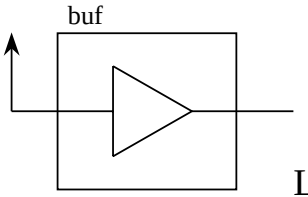
- **4-value logic system in Verilog:**



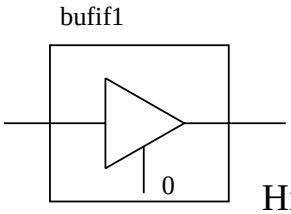
Logic Low



Unknow



Logic High



High Impedance

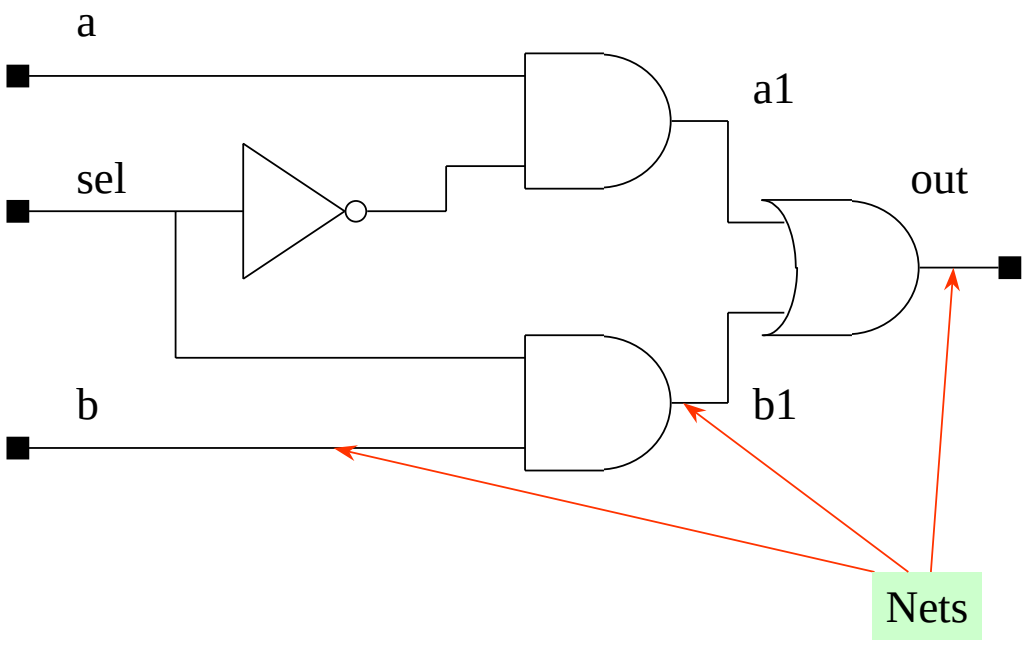
Major Data Type Class

- **Nets**
- **Registers**
- **Parameters**

Nets

- *Net* **data type represent physical connections between structural entities.**
- **A** *net* **must be driven by a driver, such as a gate or a continuous assignment.**
- **Verilog automatically propagates new values onto a net when the drivers change value.**

Nets (cont.)



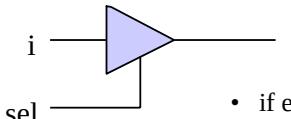
Types of Nets

- **Various net types are available for modeling design-specific and technology-specific functionality.**

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

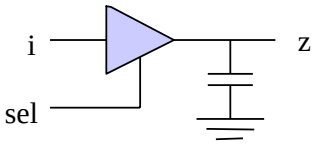
Types of Nets (cont.)

wire z ;



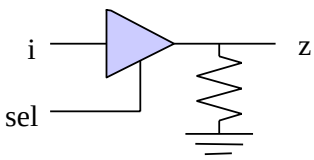
- if en = 1, z = i ;
- if en = 0, z = high impedance;

trireg z ;



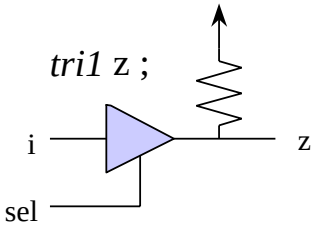
- if en = 1, z = i ;
- if en = 0, z = stores its last value;

tri0 z ;



- if en = 1, z = i ;
- if en = 0, z = 0;

tri1 z ;

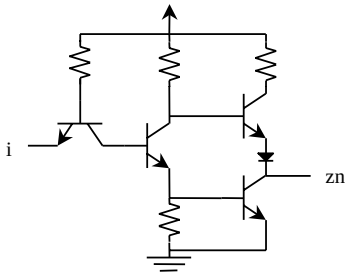


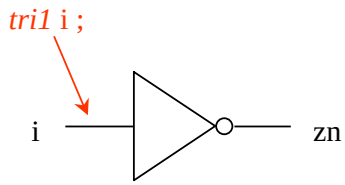
- if en = 1, z = i ;
- if en = 0, z = 1;

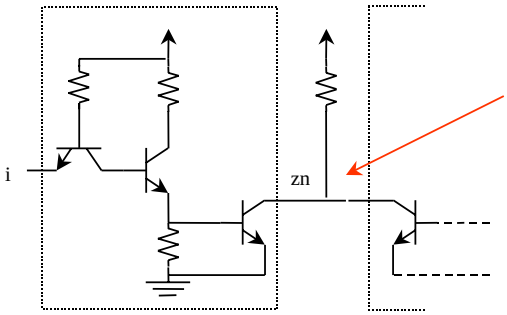
Types of Nets (cont.)

- **TTL**

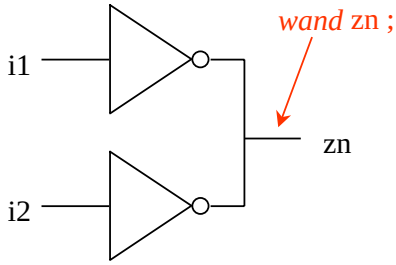
Any floating input in a TTL logic gate acts like a logical 1 applied ot that input.







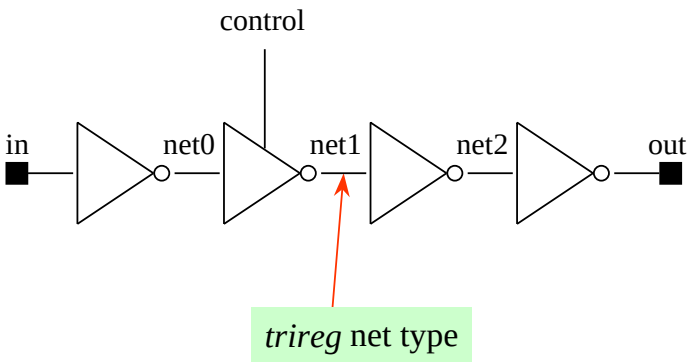
An open collector TTL output operates wired AND function when two or more gates are wired together.



Types of Nets (cont.)

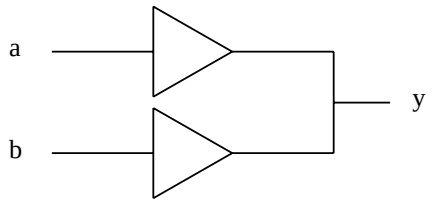
- **The default net type is *wire*.**

```
module inverter_chain(out,in);
  output  out;
  input    in;
  trireg    net1;

  not      U0(net0, in);
  notif1 U1(net1, net0);
  not      U2(net2, net1);
  not      U3(out, net2);
endmodule
```



- **use `*default_nettype <net_type>* compiler directive to change default net type.**

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

y

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

y

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

y

Declaration Syntax of Verilog Nets

- **Net declaration**

  <net_type> <range>? <delay_spec>? <net_name> <,<net_name>>*;

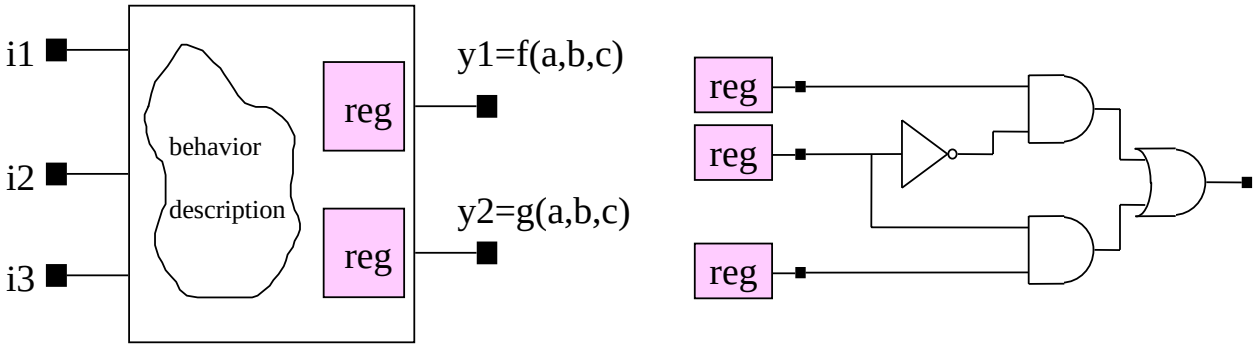
  where <range> is specified as [msb : lsb]

- **Examples**

```
module testfixture;
  wire  net1;                 //  A scalar net of type “wire”
  wand  net2;                 //  A scalar net of type “wand”
  tri  [15:0] busa;           //  A 16-bit tri-state bus
  wand  [0:31]  w1, w2;       //  Two 32-bit wires with msb = 0
  . . . . . .
endmodule
```

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.**



Registers (cont.)

```
module inverter(out, in);
  output out;
  reg    out;
  input  in;

  always @(in)
    #5 out = ~in;
endmodule

module test_fixture;
  reg stimuli;
  inverter U0(inv_out, stimuli);
  initial begin
    stimuli = 0;
    #10  stimuli = 1;
    . . . . .
   end
endmodule
```

At any time, if the input of the inverter is changed, then the output will change to the inverse state of input after 5 time unit.

At simulation time 0, you place a logic low at the inverter input.

At simulation time 10, you place a logic high at the inverter input.

Types of Registers

- **The register class consists of four data types.**

Register Types	Functionality
reg	Unsigned integer variable of varying bit width
integer	Signed integer variable, 32-bits wide. Arithmetic operations produces 2's complement result.
real	Signed floating-point variable, double precision.
time	Unsigned integer variable, 64-bit wide (Verilog-XL stores simulation time as a 64-bit positive value.)

Note : Do not confuse register data type with structural storage element (e.g., D-Flip-Flop)**.**

Declaration Syntax of Verilog Registers

- **Register declaration**

  reg <range>? <name_of_variable> <,<name_of_variable>>*;

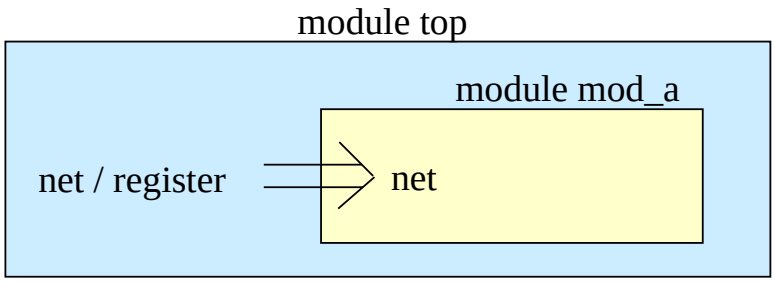
- **Examples**

```
module testfixture;
  reg a;                  //  A scalar register
  reg [3:0] opa;          //  A  4-bit vector register from msb to lsb
  reg [7:0] opa, opb;     //  Two 8-bit registers
  . . . . . .
endmodule
```

Common Mistakes in Choosing Data Types

- **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.

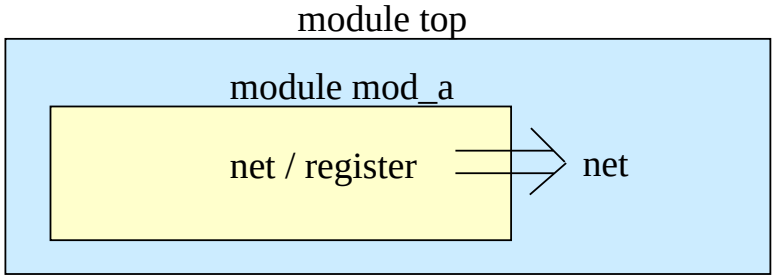
- **Error message**

" Error! Incompatible declaration, (in) defined as input "

Common Mistakes in Choosing Data Types

- **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*.

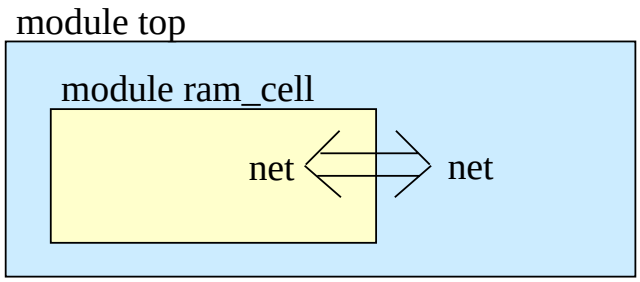
- **Error message**

" Illegal output port specification (port 0) "

Common Mistakes in Choosing Data Types

- **An inout port can be driven by a net, and it can only drive a net.**
- **Example**

```
module ram_cell(io_port, rd);
  inout io_port;
  ......
endmodule

module top;
  reg data;
  ......
  ram_cell U0(data, read);
  ......
```



You cannot use a register to drive an *inout* port.

- **Error message**

"Error! Port connections to inout port (0) have to be collapsible."

Parameters

- **Parameters are not variables, they are constants.**
- **Typically parameters are used to specify delays and width of variables.**
- **Examples**

```
module var_mux(out, i0, i1, sel) ;
  parameter width = 2, delay = 1 ;
  output  [width-1:0] out ;
  input   [width-1:0] i0, i1 ;
  input   sel;

  assign #delay out = sel  ?  i1  : i0 ;
endmodule
```

- if *sel* = 1, then *i1* will be assigned to *out*;
- if *sel* = 0, then *i0* will be assigned to *out*;

Overriding the Values of Parameters

- *defparam* **statement.**
  - use *defparam* and *hierarchical name* of that parameter to change the value of parameter.

```
module top;
  ......
  wire [1:0] a_out, a0, a1;
  wire [3:0] b_out, b0, b1;

  var_mux  U0(a_out, a0, a1, sel);
  var_mux  U1(b_out, b0, b1, sel);
    defparam  U1.width = 4 , U1.delay = 2 ;
 ......
endmodule
```

hierarchical name

Overriding the Values of Parameters (cont.)

- **You can use** *defparam* **to groupe all parameter value override assignment in one module.**

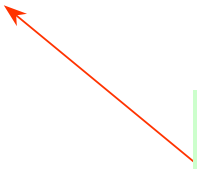
```
module top;
  ......
  wire [7:0] a_out, a0, a1;
  wire [3:0] b_out, b0, b1;

  var_mux  U0(a_out, a0, a1, sel);
  var_mux  U1(b_out, b0, b1, sel);
 ......
endmodule
```

```
module annotate;
  defparam
     top.U0.width = 8,
     top.U0.delay  = 3,
     top.U1.width = 4,
     top.U1.delay  = 2;
endmodule
```

Overriding the Values of Parameters (cont.)

- **Module instance parameter value assignment.**

```
module top;
  ......
  wire [1:0] a_out, a0, a1;
  wire [3:0] b_out, b0, b1;

  var_mux  U0(a_out, a0, a1, sel);
  var_mux  #(4, 2) U1(b_out, b0, b1, sel);
 ......
endmodule
```

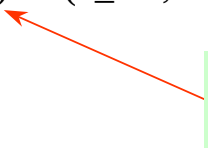


The order of assignment of parameters follows the order of declaration of parameters in the module.

Overriding the Values of Parameters (cont.)

- **In module instance parameter value assignment, you cannot skip any parameter assignment even you do not want to re-assign it.**

```
module top;
  ......
  wire [1:0] a_out, a0, a1;
  wire [3:0] b_out, b0, b1;

  var_mux  U0(a_out, a0, a1, sel);
  var_mux  #(4, ) U1(b_out, b0, b1, sel);
 ......
endmodule
```



you cannot skip the *delay* parameter assignment.

Summary

- **Verilog logic value set.**
- **Data type.**
  - net
  - register
  - parameter

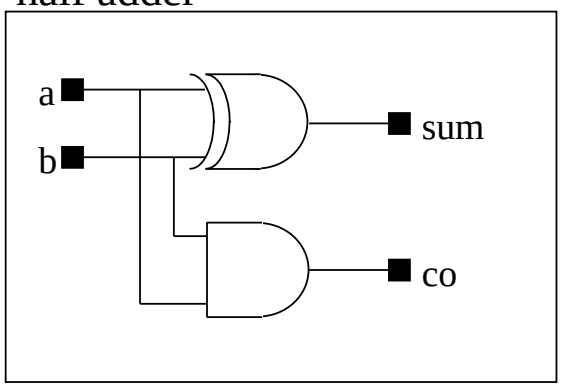
Chapter 5

Structural Modeling

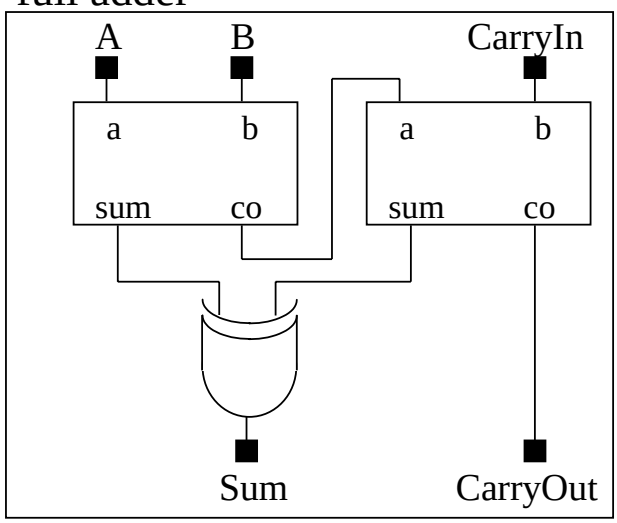
Structural Modeling

- **In structural modeling, you connect components with each other to create a more complex component.**

half adder



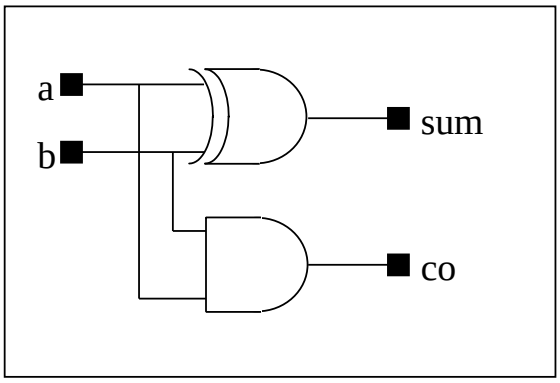
full adder



Structural Modeling (cont.)

- **Module definition**
  - A module definition is enclosed by Verilog keyword *module* and *endmodule*.
  - The module name declaration follows the *module* keyword.

half adder



module name

```
module half_adder ( port list description );
  // port declaration

  // netlist description
endmodule
```

Verilog keyword

Structural Modeling (cont.)

- **Module definition**
  - You must specify the module ports' names in parentheses, followed by the port declaration.
  - Gate-level netlist is defined in module definition.

```
module half_adder(sum,co, a, b);
  output sum, co;
  input   a, b;

  xor   U0(sum, a, b);
  and   U1(co, a, b);
endmodule
```

port names and port declaration.

Verilog Primitives

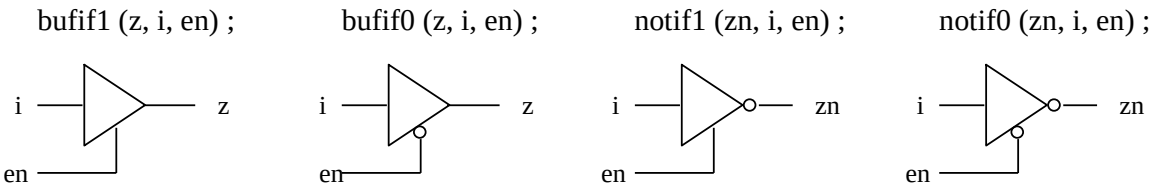
- **Verilog primitive gates provide basic logic functions.**

Primitive Name	Functionality
and	Logical AND
or	Logical OR
not	Inverter
buf	Buffer
xor	Logical Exclusive OR
nand	Logical AND Invertered
nor	Logical OR Invertered
xnor	Logical Exclusive OR Invertered

Conditional Primitives

- **Conditional primitives are enable and disable by a control pin.**

Primitive Name	Functionality
bufif1	Conditional buffer with logic 1 as enable input
bufif0	Conditional buffer with logic 0 as enable input
notif1	Conditional inverter with logic 1 as enable input
notif0	Conditional inverter with logic 0 as enable input



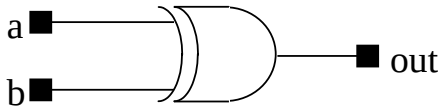
Primitive Instantiation

- **Primitive instance includes an optional instance name and a required terminal connection list.**
- **You can optionally specify the delay of a primitive instance.**
- **You must specify output terminal before input.**

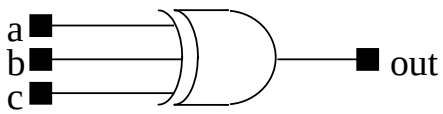
```
and  (out, in1, in2, in3, in4);        // instantiate a primitive
                                          instance without instance
                                          name
buf b1 (out1, out2, in);               // specify the instance name
notif0 #3.1 n1 (out, in, control);  //  specify the delay
```


Primitive Instantiation (cont.)

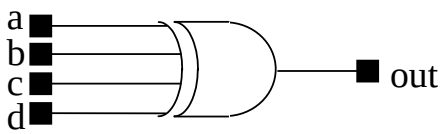
- **The number of pins for the primitive gate is defined by the number of nets connected to it.**



xor (out, a, b);



xor (out, a, b, c);

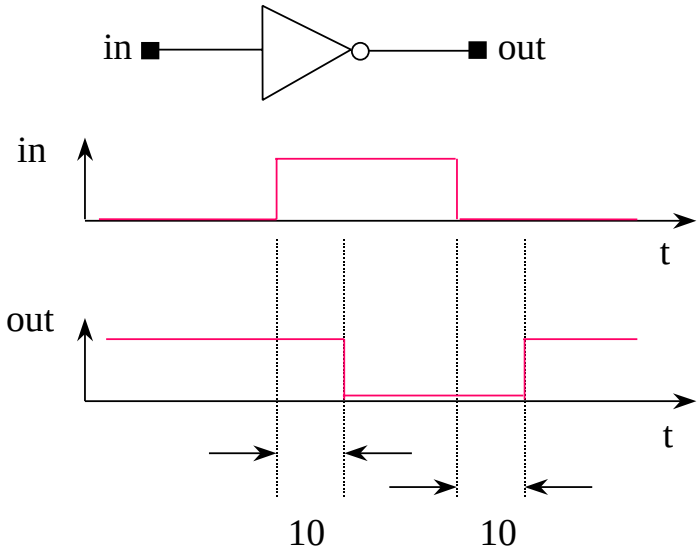


xor (out, a, b, c, d);

Delay Specification in Primitives

- **Delay specification defines the propagation delay of that primitive gate.**

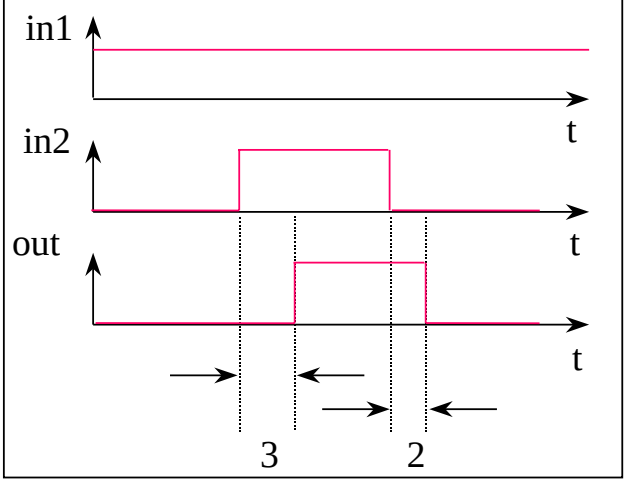
not #10 (out, in);



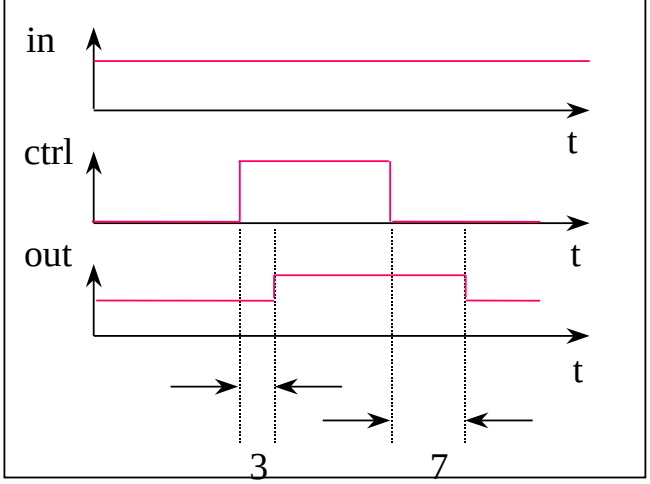
Delay Specification in Primitives (cont.)

- **Verilog supports (rise, fall, turn-off) delay specification.**

and # (3, 2) (out, in1, in2);



bufif1 # (3, 4, 7) (out, in, ctrl);



Delay Specification in Primitives (cont.)

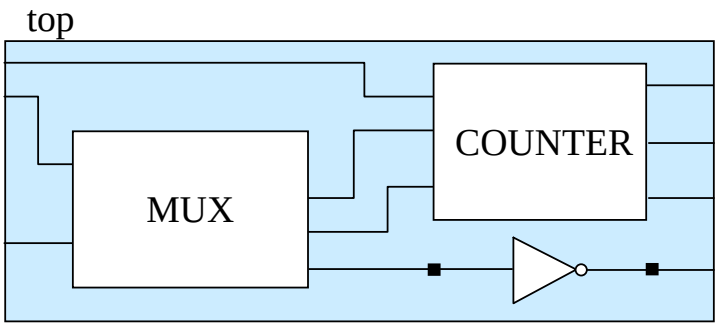
- **All delay specifications in Verilog can be specified as**
  (minimun : typical : maximum) **delays.**
- **Examples**
  - (min : typ : max) delay specification of *all* transistion.
    or # (3.2 : 4.0 : 6.3) U0(out, in1, in2);
  - (min : typ : max) delay specification of *RISE* transistion and *FALL* transistion.
    nand #(1.0 : 1.2 : 1.5, 2.3 : 3.5 : 4.7) U1(out,in1,in2);
  - (min : typ : max) delay specification of *RISE* transistion, *FALL* transistion, and *turn-off* transistion.
    bufif1 #(2.5 : 3 : 3.4, 2 : 3 : 3.5, 5 : 7 : 8) U2(out,in,ctrl);

Delay Specification in Primitives (cont.)

- **Verilog-XL uses *typical* delay values of each primitive to simulate your design.**
- **You can force Verilog-XL to use *minimum* or *maximum* delay values to simulate your design by using the following command line options**
  +mindelays , +typdelays, +maxdelays
- **Example**
  unix % verilog +lic_ncv +*mindelays* deisgn.v

Module Instantiation

- **Module instantiation allows one module to incorporate a copy of another module into itself.**



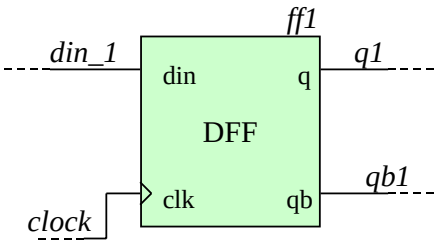

```
module top ( . . . . . . );
   port declaration

   MUX  U0( . . . . . . );
   COUNTER  U1( . . . . . . );
   not  U2( . . . );
   . . . . . .
endmodule
```

- **You need not care the inside of module instance.**
- **A module instantiation must have an instance name.**

Module Instantiation (cont.)

- **You can connect module instance's ports by *port order* or *port name*.**

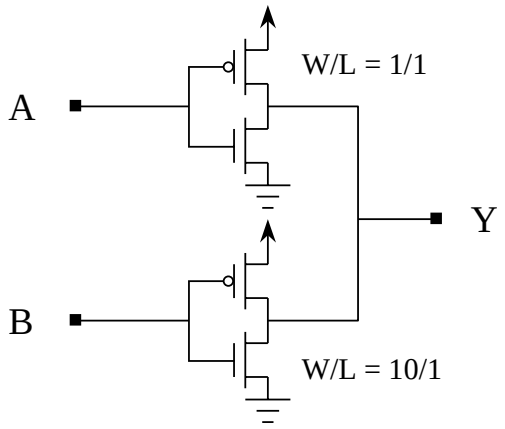
```
module DFF (q, qb, din, clk);
   output  q, qb;
   input    din, clk;
.....
endmodule
```




```
module test;
   DFF  ff1(q1, qb1, din_1, clock);    //  port order mapping
   DFF  ff2(.din(D[0]), .clk(clock), .q(O[0]), .qb(_O[0]));  // name mapping
   DFF  ff3(Q,,DIN, clock);              //  one port left unconnected
endmodule
```

Logic Strength Modeling

- **Verilog provides multiple levels of logic strengths for accurate modeling of signal contention.**
- **Logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of the hardware with maximum accuracy.**

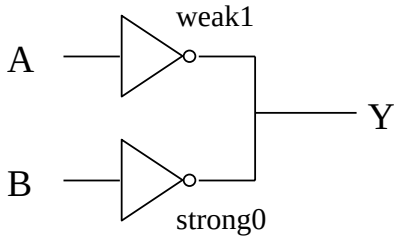


without strength modeling,
if *A*=0, *B*=1,
then *Y*=X (unknow).

however, the real situation is
if *A*=0, *B*=1,
then *Y*=0.

Logic Strength Modeling (cont.)

- **Adding logic strength properties to Verilog primitives.**



When *A*=0 and *B*=1, Verilog-XL will resolve *Y* to logic 0.

- **Strength**

0 Strength								1 Strength							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1

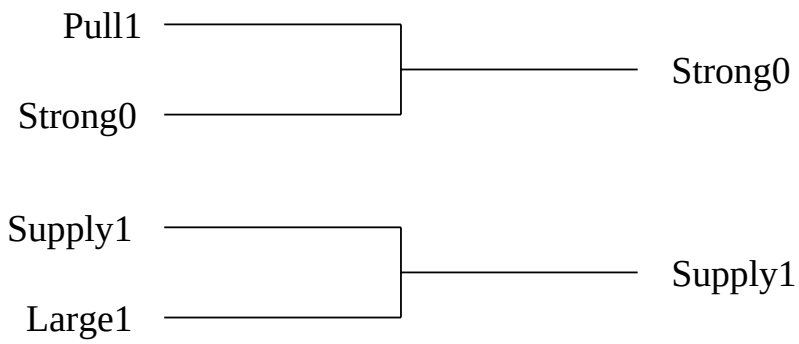
strong ⟷ weak ⟷ strong

Signal Strength Value System

Strength	Values	%v formats		Specification mnemonics	
7 Supply	Drive	Su0	Su1	supply0	supply1
6 Strong	Drive (default)	St0	St1	strong0	strong1
5 Pull	Drive	Pu0	Pu1	pull0	pull1
4 Large	Capacitive	La0	La1	large	
3 Weak	Drive	We0	We1	weak0	weak1
2 Medium	Capacitive	Me0	Me1	medium	
1 Small	Capacitive	Sm0	Sm1	small	
0 High Z	Impedance	HiZ0	HiZ1	highZ0	highZ1

Verilog Resolves Ambiguous Strength

- **If two signals of unequal strength combine in a wired net configuration, the stronger signal is the result.**



Verilog Logic Strength Modeling

- **Syntax**

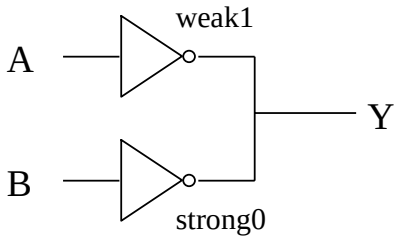
  <GATETYPE> <drive_strength>?<delay>?<gate_instance>;

  where <drive_strength> is (<strength0>,<strength1>) or (<strength1>,<strength0>).

- **Example**

  not (weak1, strong0) (Y, A);

  not (strong0, strong1) (Y, B);



- **You can use *%v* format specifier to display the strength of a net.**

  $monitor (”At time %t, the strength of Y output is %v”, $time, Y);

Summary

- **Verilog primitive cells.**
- **Structural modeling.**
  - Primitive cell instantiate.
  - Module instantiate.
  - Connections between primitive cells and modules.
  - delay specifications.
- **Logic Strength Modeling.**

Chapter 6

Support for Verification

Support for Verification

- **After you have finished your design, you can use Verilog supplied system tasks and functions to verify or debug your design.**
- **Verilog supports both text output and graphical output.**
- **Verilog also supports file output.**

Support for Verification (cont.)

- **Verilog has system functions to read current simulation time.**
  - $time
  - $stime
  - $realtime
- **Verilog has system tasks to support text output.**
  - $display
  - $strobe
  - $write
  - $monitor
- **Verilog has system tasks to support graphic output.**
  - SimWave
  - grwaves

Reading Simulation Time

- **The *$time, $stime,* and *$realtime* system functions return the current simulation time.**
- **Each of these functions returns a value that is scaled to the time unit of the module that invoked it.**
- ***$time* returns time as a 64-bit integer.**
- ***$stime* returns time as a 32-bit integer.**
- ***$realtime* returns time as a real number.**

Display Time Information

- **You can use the *$timeformat* system task and the *% t* format specifier to globally control how time values are displayed.**
- **$timeformat(<unit>,<precision>,<suffix>,<min_width>);**

```
`timescale 10ns / 10ps
module timeformat_test;
   initial begin
      $timeformat(-9,2," ns",20);
      #10 $display($time,,,,"This shows the time without %%t specifier.");
          $display("%t   This shows the time with %%t specifier.",$time);
      end
endmodule
```

simulation result

```
        10   This shows the time without %t specifier.
 100.00 ns   This shows the time with %t specifier.
```


Some Notes when Using $timeformat

- **You must use both *`timescale* compiler directive and *$timeformat* system task.**

  Warning! Design does not use timescales: $timeformat ignored.
- **The *unit* in *$timeformat* cannot be smaller than the smallest precision of the *`timescale*.**

```
`timescale 1us / 1us
.....
`timescale 10us / 10us
.....
$timeformat(-9, 2, " ns", 10);
.....
```

The smallest precision of `timescale is 1us, while the unit of $timeformat is 1ns.

Error! $timeformat: simulator is operating in 1ns.

Can't exceed this.

Printing Time Information

The following is an example of printing time information.

```
`timescale 1ns / 10ps
module time_test;
   reg in1;
   not   #9.49 n1(o1,in1);
   initial begin
     $display("              time   realtime      stime \t in1 \t o1 ");
     $monitor(" %d %t %d \t %b \t %b",$time,$realtime,$stime,in1,o1);
     $timeformat(-9, 2, " ns", 10);
     in1=0;
     #10 in1=1;
     end
endmodule
```

Result

time	realtime	stime	in1	o1
0	0.00 ns	0	0	x
9	9.49 ns	9	0	1
10	10.00 ns	10	1	1
19	19.49 ns	19	1	0

$display System Task

- ***$display* prints out the current values of signals in the argument list.**

  $display(signal1, signal2, signal3, signal4);

- **like *printf* in C programming language, *$display* support formated text output.**

  $display($time,,"%b \t %h \t %d \t %o", a, b, c, d);

- ***$display* support different default bases.**

  $displayb(signal1, signal2, signal3); // binary base

  $displayo(signal1, signal2, signal3); // octal base

  $displayh(signal1, signal2, signal3); // hexadecimal base

$display System Task (cont.)

```
module display_test;
  reg [3:0] data;

  initial begin
    data = 4'b1100;
    $display("from display,  data = ", data);
    $displayb("from displayb, data = ", data);
    $displayh("from displayh, data = ", data);
    end
endmodule
```

Result

```
from display,   data = 12
from displayb, data = 1100
from displayh, data = c
```


$display System Task (cont.)

- **Format specifier and escpaed character**

Format Specification		Escaped character	
%h or %H	display in hexadecimal format	\n	is the new line character
%d or %D	display in decimal format	\t	is the tab character
%o or %O	display in octal format	\\	is the backslash character
%b or %B	display in binary format	\”	is the “ character
%c or %C	display in ASCII format	%%	is the percent character
%v or %V	display net signal strength		
%m or %M	display hierarchical name		
%s or %S	display as a string		
%t or %T	display in current time format		

The *$write* and *$strobe* system tasks

- ***$write* is identical to *$display* except that it does not print a new line character.**

  $write (signal1, signal2, signal3, signal4);

- ***$strobe* is identical to *$display* except that the argument evaluation is delayed just prior to the advance of simulation time.**

  $strobe (signal1, signal2, signal3, signal4);

The *$write* and *$strobe* system tasks (cont.)

```
module strobe_test;
  reg [31:0] data;

  initial begin
    #10 $strobe("From strobe, data = %d", data);
        data = 10;
        $display("From display, data = %d", data);
        data = 30;
    end
endmodule
```

simulation result

```
From display, data =        10
From strobe, data =         30
```

The *$write* and *$strobe* system tasks (cont.)

- **Both *$write* and *$strobe* support formated text output.**

  $write ($time,,"%b \t %h \t %d \t %o", a, b, c, d);

  $strobe ($time,,"%b \t %h \t %d \t %o", a, b, c, d);

- **Both *$write* and *$strobe* support multiple default bases.**

  $writeb(signal1, signal2, signal3); // binary base

  $writeo(signal1, signal2, signal3); // octal base

  $writeh(signal1, signal2, signal3); // hexadecimal base

  $strobeb(signal1, signal2, signal3); // binary base

  $strobeo(signal1, signal2, signal3); // octal base

  $strobeh(signal1, signal2, signal3); // hexadecimal base

$monitor System Task

- ***$monitor* displays the values of arguments in argument list whenever any value of its argument changes.**

$monitor("%t",$time,"in=%b enable=%b out=%b ", in, enable, out);

$monitor will not print any thing when *$time* changes.

When any one of *in*, *enable*, or *out* change values, *$monitor* will print their current value.

- **Any subsequent *$monitor* overrides the previous call of *$monitor*.**

$monitor("%t",$time,"U0's driving strength = %v",out0);

. . . . . .

$monitor("%t",$time,"in=%b out=%b ", in, out);

This subsequent *$monitor* will override the previous one.

$monitor System Task (cont.)

```
module monitor_test;
  reg in; wire out;
  not #1 G0(out, in);

  initial
    $monitor($time,,"out = % in = %b",
    out, in);

  initial begin
    in=0;
    #10 in=1;
    #10 in=0;
    end
endmodule
```

Result

```
 0 out = x   in = 0
 1 out = 1   in = 0
10 out = 1   in = 1
11 out = 0   in = 1
20 out = 0   in = 0
21 out = 1   in = 0
```

```
module monitor_test;
  reg in; wire out;
  not #1 G0(out, in);

  initial
    $display($time,,"out = %bin = %b",
    out, in);

  initial begin
    in=0;
    #10 in=1;
    #10 in=0;
    end
endmodule
```

Result

```
0 out = x   in = x
```


File Output

- **Each of the four formatted display tasks --- $display, $write, $monitor, $strobe --- has a counterpart that writes to specific files as opposed to the log file and standard output.**

  $fdisplay(<multi_channel_descriptor>, P1, P2, ..., Pn);
  $fwrite(<multi_channel_descriptor>, P1, P2, ..., Pn);
  $fmonitor(<multi_channel_descriptor>, P1, P2, ..., Pn);
  $fstrobe(<multi_channel_descriptor>, P1, P2, ..., Pn);

- **The multi-channel-descriptor indicates where to direct the file output.**

  <multi_channel_descriptor> = $fopen("<file_name>");
  $fclose(<multi_channel_descriptor>);

File Output (cont.)

```
integer messages, broadcast, cpu_chann, alu_chann;
initial begin
   cpu_chann = $fopen("cpu.dat");   if(cpu_chann == 0) $finish;
   alu_chann = $fopen("alu.dat");     if(alu_chann == 0) $finish;
   messages = cpu_chann | alu_chann;
   broadcast = 1 | messages;
   end
// at every positive edge of clock print the following line to alu.dat
always @(posedge clock)
   $fdisplay(alu_chann, "acc = %h f = %h a = %h", acc, f, a);

/*at negative edge of reset print the following line to alu.dat, cpu.dat, standard output, and
    verilog.log */
always @(negedge reset)
   $fdisplay(broadcast, "system reset at time %t", $time);
```


Summary

- **Verilog system tasks and system functions.**
  - $display, $write, $strobe, $monitor
  - $time, $stime, $realtime
- **File output.**
  - $fopen, $fclose
  - $fdisplay, $fwrite, $fstrobe, $fmonitor

Chapter 7

Assignments

Assignments

- **Assignment : drive values into nets and registers.**
- **There are two basic forms of assignment**
  - *continuous* assignment, which assigns values to *nets*.
  - *procedural* assignment, which assigns values to *registers*.
- **basic form**

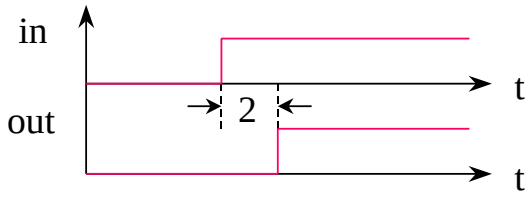
  <left_hand_side> = <right_hand_side>

Assignments	Left Hand Side
Continuous Assignment	net *wire, tri*
Procedural Assignment	register *reg, integer, real*

Continuous Assignments

- **Continuous Assignments**
  - Any changes in the RHS of the continuous assignment are evaluated and the LHS is updated.

```
assign #2 out = in;
```



Continuous Assignments (cont.)

- **Continuous assignments provide a way to model combinational logic.**

continuous assignment

```
module inv_array(out, in);
   output [31:0] out;
   input  [31:0] in;

   assign out = ~in;
endmodule
```

gate-level modeling

```
module inv_array(out, in);
   output [31:0] out;
   input  [31:0] in;

   not  G0(out[0], in[0]);
   .....
   not  G31(out[31], in[31]);
endmodule
```

Continuous Assignments (cont.)

- **Right hand side can be**
  - expression.

    assign and_out = i1 & i2;
  - value.

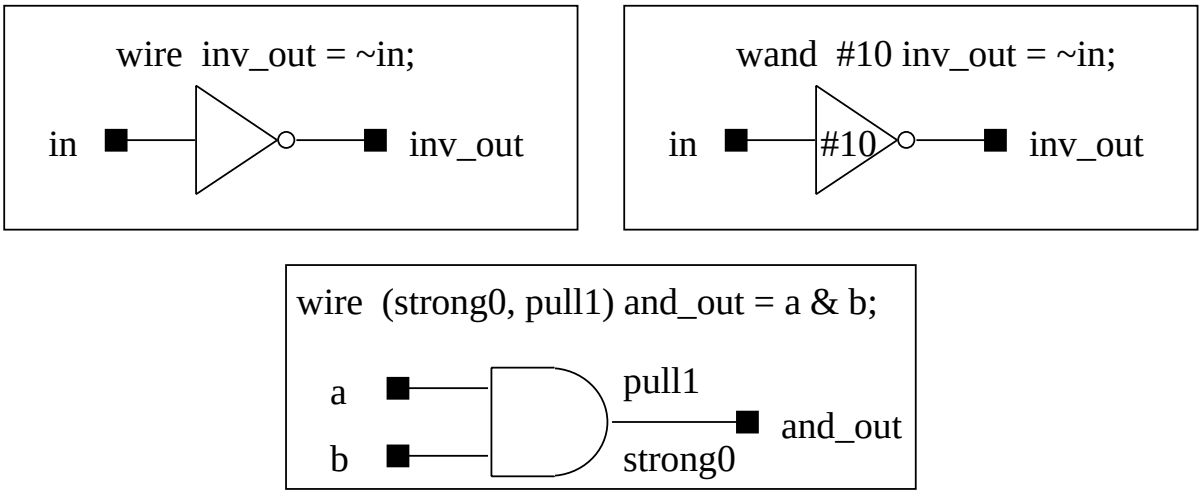
    assign net_1 = 1;
  - other net

    assign net_a = net_b;
- **Type of continuous assignment declaration**
  - the net declaration assignment
  - the continuous assignment statement

The Net Declaration Assignment

- **The net declaration assignment.**

  <NETTYPE> <drive_strength>? <range>? <delay>? <list_of_assignment>;
- **Examples**



The Continuous Assignment Statement

- **The continuous assignment statement.**

  assign <drive_strength>? <delay>? <list_of_assignment>;

- **Example**

```
module MUX2_1(out, i0, i1, sel);
    output out;
    input i0, i1, sel;

    assign  out = sel ? i1 : i0;
endmodule
```

Note : The continuous assignment statement drive a net that has been previously declared.

```
.....
reg  in;
assign  out = ~in;
......
```


Operators

- **The expression on the RHS of continuous assignment can contain the following three types of operators:**
  - arithmetic operator

operator	operation
+	arithmetic addition
-	arithmetic substraction
*	arithmetic multiplication
/	arithmetic division
%	arithmetic modulus

  - Example

    -10 % 3 = -1

    14 % -3 = 2

    The result takes the sign of the first operand.

Operators (cont.)

- unary operator (1-bit result)

operator	operation
&	unary reduction AND
~&	unary reduction NAND
|	unary reduction OR
~|	unary reduction NOR
^	unary reduction XOR
~^	unary reduction XNOR

  – unary operation will preform the operation on each bit of the operand and get a one-bit result.

    |8'b00101101 is 1'b1

Operators (cont.)

- bit-wise operators

operator	operation
~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
~^	bit-wise XNOR

  – binary bit-wise operation will perform the operation one bit of a operand and its equivalent bit on the other operand to calculate one bit for the result.

(8'b11110000 & 8'b00101101) is 8'b00100000

Operators (cont.)

- logical operators

operator	operation
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
===	logical identity
!==	the inverse of ===

– logical operator operate with logic values. (non-zero is true, and zero value is false).

if(sel == 4'h03) ……. else ……..

Operators (cont.)

- Example

opa = 0010 opb = 1100 opc = 0000

unary reduction

& opa = 0

0 & 0 & 1 & 0 = 0

logical operation

opa && opc = 0

opa = 0010 → true
opc = 0000 → false
true && false = false

bit-wise operation

opa & opb = 0000

```
  0000
& 1100
------
  0000
```

bit-wise operation

~ opa = 1101

logical operation

opa && opb = 1

opa = 0010 → true
opb = 1100 → true
true && true = true

logical operation

! opa = 0

Operators (cont.)

- example

```
assign   inv_out = ~in,
         and_out = i1 & i2;
assign   #10 inv_out2 = ~in;
```

- other operators

operator	operation
>>	logical shift right
<<	logical shift left
==, !=	equality
===, !==	identity
?:	conditional
{}	concatenate
{{}}	replicate

Shift Operator

```
module shift_register(reg_out, reg_in);
  output [5:0] reg_out;
  input  [5:0] reg_in;

  parameter shift = 3;
    assign reg_out = reg_in << shift;
endmodule
```

examples :

reg_in = 6'b011100
reg_in << 3 → 100000
reg_in >> 3 → 000011

Equality and Identity Operator

- equality operator

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
a = 2'b0x
b = 2'b0x

if (a == b)
  $display("a is equal to b");
else
  $display("a is not equal to b");
```

result : a is not equal to b

- identity operator

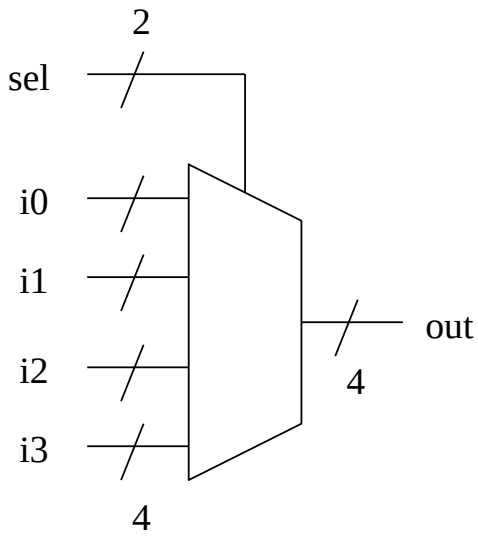
===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 2'b0x
b = 2'b0x

if (a === b)
  $display("a is identity to b");
else
  $display("a is not identity to b");
```

result : a is identity to b

Conditional Operator

```
module MUX4_1(out, i0, i1, i2, i3, sel);
   output [3:0] out;
   input  [3:0] i0, i1, i2, i3;
   input  [1:0] sel;

   assign out = (sel == 2'b00) ? i0 :
                (sel == 2'b01) ? i1 :
                (sel == 2'b01) ? i2 :
                (sel == 2'b01) ? i3 :
                4'bx;
endmodule
```



Concatenation and Replication Operator

- **Concatenation operator in LHS**

```
module add_32 (co, sum, a, b, ci);
  output co;
  output [31:0] sum;
  input  [31:0] a, b;
  input  ci;
   assign #100 {co, sum} = a + b + ci;
endmodule
```

- **Bit replication to produce *01010101***

```
assign byte = {4{2'b01}};
```

- **Sign Extension**

```
assign word = {{8{byte[7]}}, byte};
```


Procedural Assignments

- **Procedural assignments drives values onto register.**
  - reg
  - integer
  - real
  - time
- **Major difference between continuous assignment and procedural assignment**
  - the LHS are updated in different time
    - in continuous assignment, LHS are updated whenever the RHS changes valule.
    - in procedural assignment, LHS are updated when this statement is encountered.

Procedural Assignments (cont.)

```
module assignment_test;
  reg [3:0] r1, r2;
  reg [4:0] sum2;
  wire [4:0] sum1;

  assign sum1 = r1 + r2;

  initial begin
    r1=4;b0010;  r2=4'b1001;
    sum2 = r1 + r2;
    $display(" r1   r2   sum1   sum2");
    $monitorb(r1, r2, sum1, sum2);
    #10  r1 = 4'b0011;
    end
endmodule
```

continuous assignment

procedural assignment

Result

r1	r2	sum1	sum2
0010	1001	01011	01011
0011	1001	01100	01011

Procedural Assignments (cont.)

- **Major difference (cont.)**
  - The place where you put the assignment statement
    - A continuous assignment statement cannot be inside procedural blocks.
    - A procedural assignment statement must be inside procedural blocks.

```
module f_adder (sum, co, a, b, ci);
   output sum, co;
   input   a, b, ci;
   reg sum;

   sum = a ^ b ^ ci;
   always @ (a or b or ci)
      assign co = (a & b) | (b & ci) | (ci & a);
endmodule
```

Error! Illegal left-hand-side continuous assignment.

Error! Illegal left-hand-side in assign statement

Summary

- **Continuous assignment.**
- **Assignment operators.**
- **Procedural assignment.**

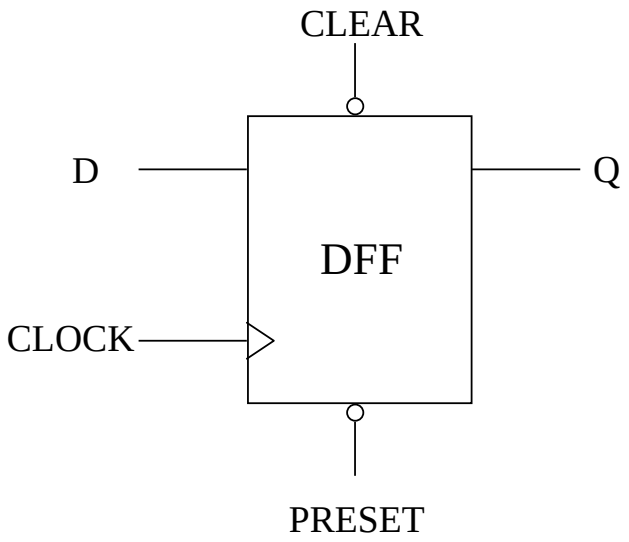
Chapter 8

Behavior Modeling

Behavior Modeling

- **At system level, system's functional view is more important than implementation.**
  - you do not have any idea about how to implement your netlist.
  - the data flow of this system is analyzed.
  - you may need to explore different design options.
- **Behavior modeling enables you to describe the system at a high-level of abstraction. All you need to do is to describe the behavior of your system.**

Behavior Modeling (cont.)



behavior of DFF

At every positive edge of CLOCK
if PRESET and CLEAR is not low, then
set Q to the value of D

Whenever PRESET goes low
Set Q to logic 1

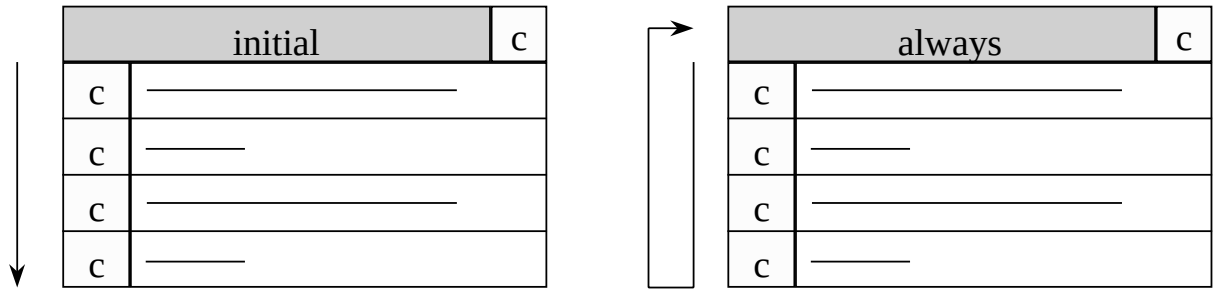
Whenever CLEAR goes low
Set Q to logic 0

Behavior Modeling (cont.)

- **In behavior modeling, you must describe your circuits'**
  - action
    - how do you model your circuit's behavior / behaviors?
  - timing control
    - at what time do what thing.
    - at what condition do what thing.
- **Verilog supports the following constructs to model circuits' behavior**
  - procedural block.
  - procedural assignment.
  - timing control.
  - control statement.

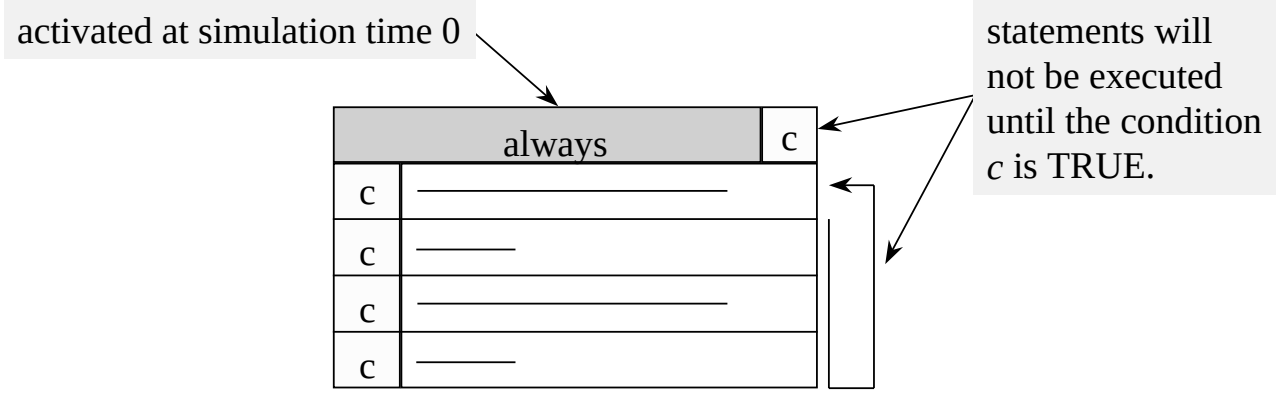
Procedural Blocks

- **In Verilog, procedural blocks are the basis of bahavior modeling.**
  - you can describe one behavior in one procedural block.
- **Procedural blocks are of two types**
  - *initial* procedural block, which execute only once.
  - *always* procedural block, which execute in a loop.



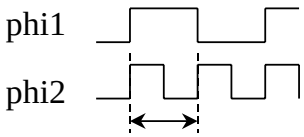
Procedural Blocks (cont.)

- **All procedural blocks are activated at simulation time 0.**
  - with enabling condition, the block will not be executed until the enabling condition evaluates to TRUE.
  - without enabling condition, the block will be executed immediately.



Procedural Blocks (cont.)

```
module clock_gen (phi1, phi2);
  output phi1, phi2;
  reg phi1, phi2;

 initial begin
    phi1 = 0; phi2 = 0;
    end

 always #100 phi1 = ~ phi1;

 always @(posedge phi1)
    begin
      phi2 = 1;
      #50 phi2 = 0;
      #50 phi2 = 1;
      #50 phi2 = 0;
    end
endmodule
```



100

These procedural blocks are activated and executed at simulation time 0.

This procedural block is activated at simulation time 0 but executed at positive edge of *phi1*.

Procedural Blocks (cont.)

- **Procedural blocks have the following components**
  - procedural assignment statements
  - timing controls
  - high-level programming language constructs
- **Use procedural assignment and high-level programming language constructs to model the actions of your circuit.**
- **Use timing controls to model when should these actions happen.**

Procedural Assignment

- **Procedural assignments drive values or expressions onto registers (*reg, integer, real, time*).**

```
module adder32 (sum, carry, a, b, ci);
  output [31:0] sum;
  output carry;
  input [31:0] a, b;
  input ci;
  reg [31:0] sum;
  reg carry;

  always @ (a or b or ci)
    { carry, sum } =  a + b + ci;
endmodule
```

Procedural Continuous Assignment

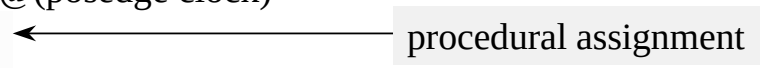
- **The *assign* and *deassign* procedural statements**
  - you can use *assign* procedural statements to **continuously** drive expressions onto registers.
  - these procedural statements must be in procedural blocks.
  - The *assign* procedural statement overrides procedural assignment to a register.
  - use *deassign* procedural statement to end a procedural continuous assignment to a register.
- **Syntax**

  assign <LHS> = <RHS>;

  deassign <LHS>;

Procedural Continuous Assignment (cont.)

```
module dff (q, d, clear, preset, clock);
  output q;
  input d, clear, preset, clock;
  reg q;

  always @(clear or preset)
    if (!clear)
      assign q = 0;
    else if (!preset)
      assign q = 1;
    else
      deassign q;

  always @(posedge clock)
    q = d;
endmodule
```



Procedural Timing Control

- **Three types of procedural block timing control**
  - simple delay control

    #100 clk = ~clk;
  - event control

    @(a or b or ci) sum = a + b + ci;

    @(posedge clock) q = d;

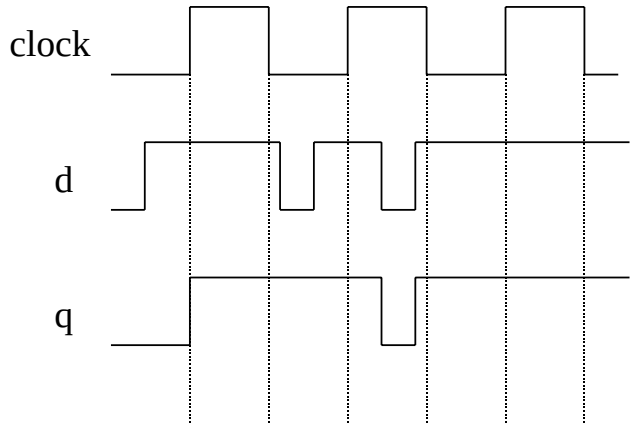
    @(negedge clear) assign q = 0;

Note : an event means any value change of a net or register.

Procedural Timing Control (cont.)

- level-sensitive event control

```
module level_latch(q, d, clock);
  output q;
  input  d, clock;
  reg q;

  always begin
    wait (clock) assign q = d;
    wait (!clock) deassign q;
    end
endmodule
```



level-sensitive timing control

Intra-Assignment Timing Control

- **Previously described timing control.**

```
#100 clk = ~clk;
@(posedge clock) q = d;
```

- **Intra-assignment timing control.**

```
clk = #100 ~clk;
q = @(posedge clock) d;
```

- **Simulators perform two steps when encounter an intra-assignment timing control statement**
  - evaluate the RHS immediately.
  - execute the assignment after a proper delay.

Intra-Assignment Timing Control (cont.)

- **Intra-assignment timing control can be accomplished by using the following constructs**

With intra-assignment construct	With intra-assignment construct

a = #5 b;	begin
	  temp = b;
	  #5 a = temp;
	end

a = @(posedge clk) b;	begin
	  temp = b;
	  @(posedge clk) a = temp;
	end

a = repeat(3)@(posedge clk) b;	begin
	  temp = b;
	  @(posedge clk);
	  @(posedge clk);
	  @(posedge clk) a = temp;
	end

Intra-Assignment Timing Control (cont.)

- **Data swap**

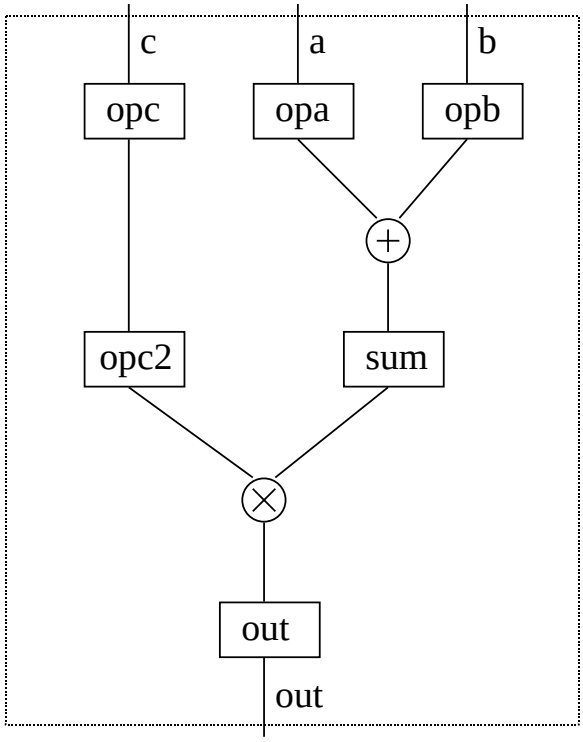
```
fork
  a = #5 b;
  b = #5 a;
join
```

- **Data shift**

```
fork
  a = @(posedge clk) b;
  b = @(posedge clk) c;
join
```

Intra-Assignment Timing Control Example

```
module data_pipe (out, a, b, c);
  output [7:0] out;
  reg [7:0] out;
  input [7:0] a, b, c;
  reg [7:0] opa, opb, opc, opc2, sum;

  always @(posedge clk)
    fork
      opa = @(negedge clk) a;
      opb = @(negedge clk) b;
      opc = @(negedge clk) c;
      sum = @(negedge clk) opa + opb;
      opc2 = @(negedge clk) opc;
      out = @(negedge clk) opc2 * sum;
    join
endmodule
```



Block Statement

- **Block statements are used to group two or more statements together.**
- **Two types of block**
  - sequential block
    - enclosed by keyword *begin* and *end.*
  - parallel block
    - enclosed by keyword *fork* and *join.*

initial	c
begin	
c	———
c	———
end	

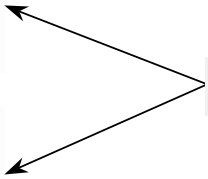
initial	c
fork	
c	———
c	———
join	

always	c
begin	
c	———
c	———
end	

always	c
fork	
c	———
c	———
join	

Block Statement (cont.)

```
begin
  #10 r = 'h35;
  #10 r = 'hE2;
  #10 r = 'h00;
  #10 -> end_wave;
end
```



equivalent statement

```
fork
  #10 r = 'h35;
  #20 r = 'hE2;
  #30 r = 'h00;
  #40 -> end_wave;
join
```


Non-Blocking Procedural Assignment

- **Blocking procedural assignment.**

  rega = #100 regb;

  rega = @(posedge clock) regb;

- **Non-blocking procedural assignment.**

  rega <= #100 regb;

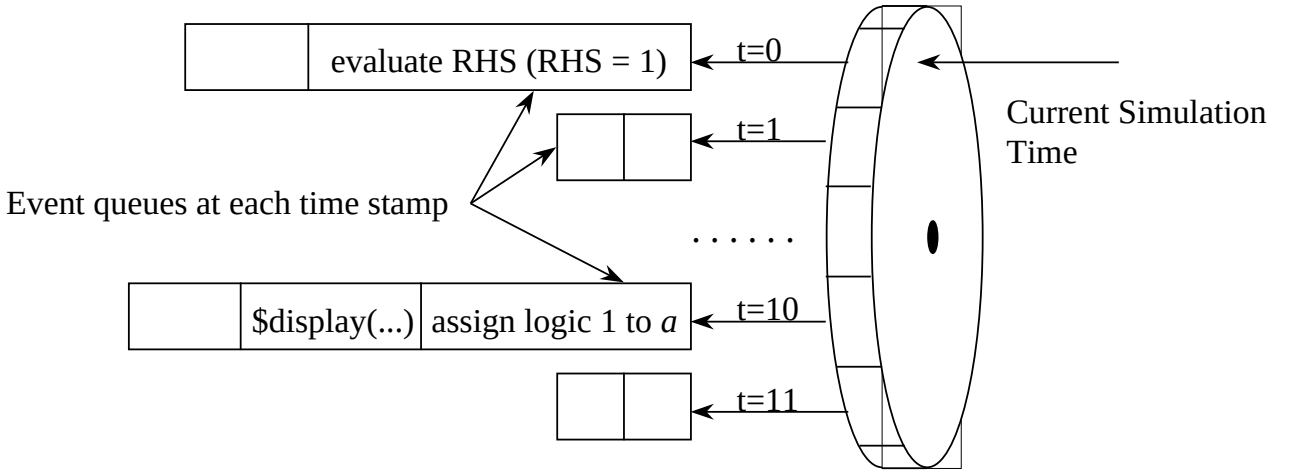
  rega <= @(posedge clock) regb;

- **Schedule the assignment without blocking the procedural flow.**
- **Simulators perform two steps when encounter an non-blocking procedural assignment statement**
  - evaluate the RHS immediately.
  - schedule the assignment at a proper time.

Non-Blocking Procedural Assignment (cont.)

```
initial begin
  a = #10 1;
  $display("current time = %t  a = %b", $time, a);
  end
```

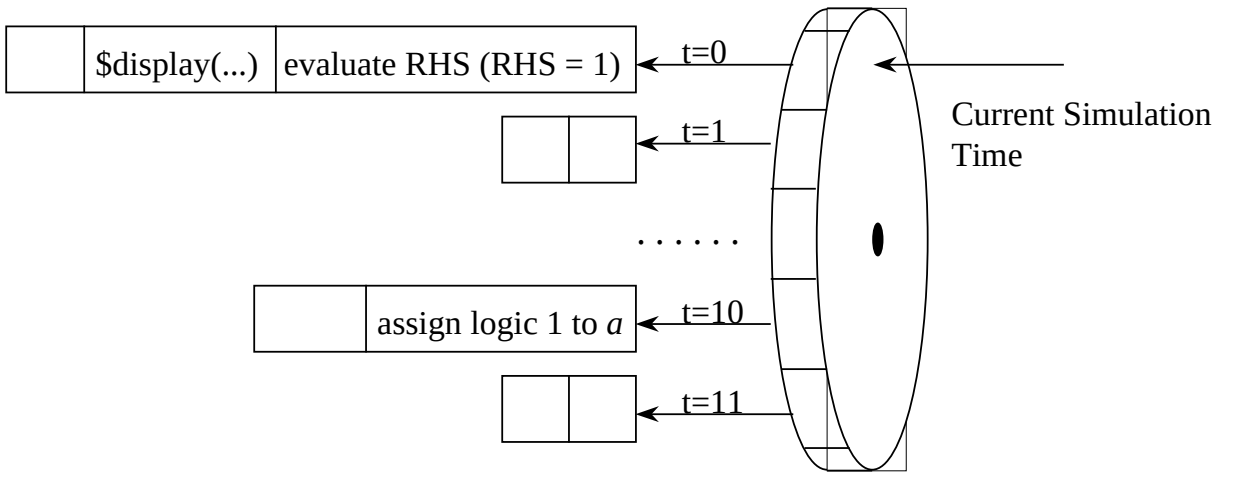
simulation result → current time = 10, a = 1



Non-Blocking Procedural Assignment (cont.)

```
initial begin
  a <= #10 1;
  $display("current time = %t  a = %b", $time, a);
  end
```

simulation result → current time = 0, a = x



Conditional Statements

If and If-Else Statements

```
if (expression)
   statement
else
   statement
```

```
if (expression)
   statement
else if (expression)
   statement
else
   statement
```


Conditional Statements (cont.)

If and If-Else Statements (cont.)

- Examples

```
if (rega >= regb)
   result = 1;
else
   result = 0;
```

```
if (index > 0)
   if (rega > regb)
      result = rega;
   else
      result = 0;
else
   $display("* Warning * index is equal or small than 0!");
```

Conditional Statements (cont.)

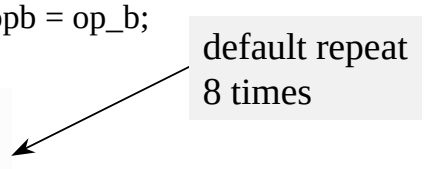
Case Statement

```
`define  pass_accum  4'b0000
`define  pass_data     4'b0001
`define  ADD           4'b0010
. . . . . .
initial
  case (opcode)
    `pass_accum : #3.5 alu_out = accum;
    `pass_data     : #3.5 alu_out = data;
    `ADD           : #3.5 alu_out = accum + data;
    `AND           : #3.5 alu_out = accum & data;
    `XOR           : #3.5 alu_out = accum ^ data;
    default        : #3.5 alu_out = 8'bx;
  endcase
```


Looping Statements

Repeat Loop

```
module multiplier(result, op_a, op_b);
  ...
  reg shift_opa, shift_opb;
  parameter size = 8;
  initial begin
    result = 0;  shift_opa = op_a;  shift_opb = op_b;
    repeat (size)
      begin
        #10  if (shift_opb[1])
                result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
      end
    end
endmodule
```



Looping Statements (cont.)

While Loops

```
initial
  begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = reg;
    while (tempreg)
      begin
        if (tempreg[0]) count = count + 1;
        tempreg = tempreg >> 1;
      end
  end  // block count1s
```

```
rega = 101;
```

tempreg	count
101	1
010	1
001	2

Looping Statements (cont.)

For Loops

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

initial
  begin : mult
    integer bindex;
    result = 0;
    for (bindex=1; bindex <= size; bindex = bindex + 1)
      if (opb[bindex])
        result = result + (opa << (bindex -1));
  end
```

Looping Statements (cont.)

For Loops

```
initial
  begin : count1s
     reg [7:0] tempreg;
     count = 0;

     for ( tempreg = rega; tempreg; tempreg = tempreg >> 1 )
       if ( tempreg[0] ) count = count + 1;
  end
```


Summary

- **Procedural blocks.**
  - initial block
  - always block
- **Timing control.**
  - simple timing control
  - level-sensitive timing control
  - edge-sensitive timing control
  - intra-assignment timing control
- **Block statement.**
- **Verilog control statement.**

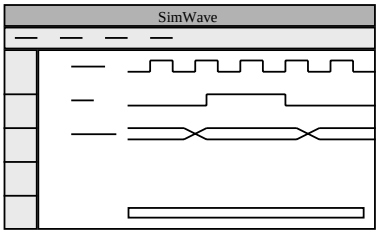
Chapter 9

Interactive Debugging in Verilog

Running in Batch Mode vs. Interactive Mode

- **In batch mode, all results are analyzed and debugged at the end of simulation run.**

```
unix %  verilog +lic_ncv design.v test.v
unix %  simwave &
```



- **In interactive mode, the simulation is analyzed at the simulation progresses.**

```
unix %  verilog +lic_ncv -s design.v test.v
C1>
```

Entering the Interactive Mode

- **Interactive mode can be entered in three ways:**

  -s command-line option.

  #10 $stop; system task placed in procedural description.

  ^c asynchronous interrupt while simulation is running.

  – unix % verilog +lic_ncv -s design.v test.v

  – .....

```
initial
  begin
    .....
    #100 $stop;
  end
.....
```


Entering the Interactive Mode (cont.)

- **In interactive mode, Verilog-XL prompts you for interactive commands by using the following command prompt:**

  C1 >

- **If Verilog-XL thinks an interactive command is not completed, then it will respond with “>“**

```
C1 > forever @(read) begin
> if (read == 1) $stop;
> end
C2 > .

C1: $stop at simulation time 0
C2 >
```

Verilog-XL responds with “>“ because it thinks the *forever* command is not completed.

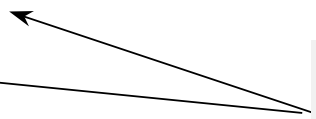
continue the simulation.

Typical Tasks in Interactive Mode

Decompile the design

- The *$list* system task decompiles the current or named scope.

```
C4 > $list;
// mux21.v
  7  `timescale  1ns /  10ps
  7  module mux21(out, i0, i1, s);
  8      output   out;   // = 1'h0, 0
  9      reg      out;   // = 1'h0, 0
 10      input    i0,    // = St0
 10               i1,    // = St1
 10               s;     // = St0
 12      always  @(s or i0 or i1)
 12          begin
 13              if(s == 0) out = i0;
                 else
 16              out = i1;
 17          end
 18  endmodule
```



Current states of declared variables and I/O port.

Typical Tasks in Interactive Mode (cont.)

Trace the simulation step by step.

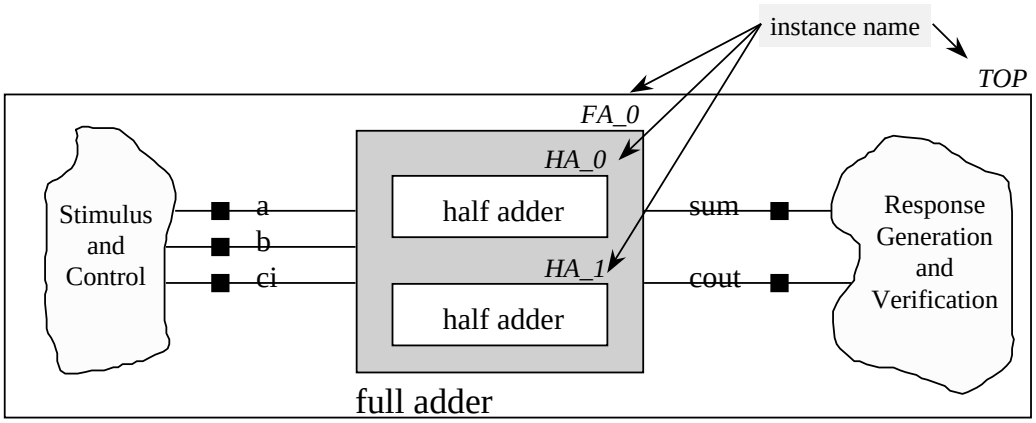
Command	Syntax	Description
continue	.	continue the simulation
step	;	step a single statement
trace-step	,	step and trace a single statement
where	:	print current location

Example

```
C1 > ,
L10 "mux21.v": wire i1 >>> NET = St1
C1 > :
Line 12, file "mux21.v", module (testfixture.mux)
Scope is (testfixture)
C1 >
```

Scopes in Verilog

- **A scope represent a hierarchical level.**



four scopes:

TOP
TOP.FA_0
TOP.FA_0.HA_0
TOP.FA_0.HA_1

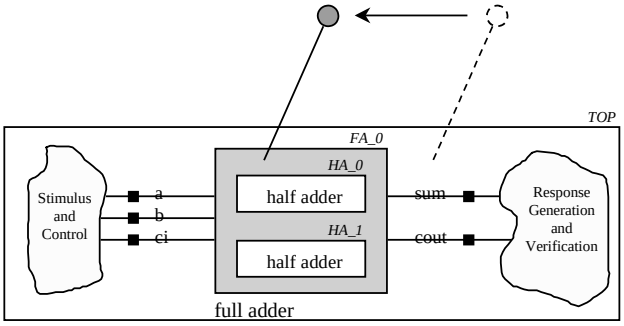
Typical Tasks in Interactive Mode (cont.)

Traversing the Design Hierarchy

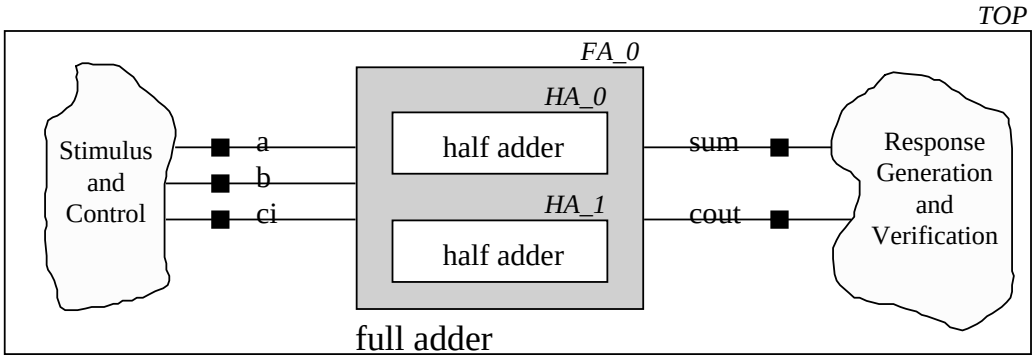
C1 > $showscopes;
Directory of scopes at current scope level:
module (fadder), instance (FA_0)

Current scope is (top)
Highest level modules:
top

C2 > $scope(FA_0);
C3 >



An Example of Traversing the Design



EQUIVALENT $list commands

$list (TOP.FA_0.HA_1);	$scope (TOP.FA_0.HA_1); $list;	$scope (TOP.FA_0); $list (HA_1);

Typical Tasks in Interactive Mode (cont.)

Display signal values with *$showvars* system task.

- *$showvars(<name>)* can display the value of a specific register or a net.

```
C1 > $showvars(bus);
bus[1] (top) wire = St1
   HiZ <- (top): bufif0 (bus[1], data[1], read);
   St1  <- (top.U0): bufif1 (DATA[1], intBus[1], read);
bus[0] (top) wire = St1
   HiZ <- (top): bufif0 (bus[0], data[0], read);
   St1  <- (top.U0): bufif1 (DATA[0], intBus[0], read);
C2 >
```

- *$showvars* can display the informations of all variables in current scope.

Typical Tasks in Interactive Mode (cont.)

Setting Breakpoints

- breakpoints can be set on specific condition happens.

```
C1 >  forever @(posedge clk) $stop;
C2 >  .
.....
C5 > if ( en1 | en2 )
>  @(posedge clk)
>  $stop;
C6 >
......
C9 >  $finish;
```

Always stop the simulation at posedge of clk.

If en1 or en2 = 1, then wait for posedge of clk and stop the simulation.

- type a period "." to resume the simulation.

Typical Tasks in Interactive Mode (cont.)

Circuit Patching

- *force* and *release* can be used to interactively patch the design.
- These commands allows the drivers on nets or statement to be overriden for a controlled perior of time.

```
C1 >  force o1 = i1 | i2;
C2 >  ......
......
C14 >  release o1;
C15 >  $reset;  // reset the simulation to time zero.
```

- The *$list_forces* system task lists the currently active force statements.

Typical Tasks in Interactive Mode (cont.)

Tracing simulation activity

- The *$settrace* and *$cleartrace* system tasks turn simulation tracing on and off.

```
C1 > forever begin
> @(write) $settrace;  #2 $cleartrace;  $stop;  end
C2 > .
C1: #2
L8 "ram.v": bufif1  >>> GATE = StX
L9 "ram.v": bufif1  >>> GATE = StX
L8 "drive.v": bufif0  >>> GATE = HiZ
.....
C1: $cleartrace;
C1: $stop at simulation time 2
C2 >
```


Interactive Command History

- **The *$history* system task lists previously executed interactive commands. You can execute a command again by entering the command number.**

```
C6 > $history;

Command history:
C1  $db_step(10);
C2  $list;
C3  $scope(U0);
C4  $list;
C5  $showvars(DATA);
C6* $history;
C7 >  ;
C7 >  5
```

step a statement and then observe the new value of net *DATA*.

Input Commands from a Text File

- **You can input a list of Verilog interactive commands from a command file.**
  - input the commands in interactive mode:

    C1 > $input(“ command.vlog “);
  - input the commands with -i command line option:

    unix % verilog design.v test.v -i command.vlog

A command file example. →

```
$db_step(10);
$list;
$scope(U0);
$list;
$showvars(DATA);
$history;
,
5
```


Saving a Verilog Simulation

- **Verilog can save the simulation data structure to a permanent file.**

```
module checkpoint;
  initial
//  Save the simulation data structures after 500 ns.
    #500 $save("saved.dat");
//  Incrementally save the simulation after every 100000 ns.
  initial
  begin
    #100000 $incsave( "inc1.dat" );
    #100000 $incsave( "inc2.dat" );
  end
endmodule
```

Restarting a Verilog Simulation

- **You can restart the simulation from the time of save.**
  - use *$restart* in interactive mode to restart from an already saved file.

    C1 > $restart("save.dat");

    C2 >
  - use the *-r* command line option to restart from an already saved file.

    unix % verilog -r save.dat

Summary

- **Entering Verilog-XL interactive mode.**
- **Interactive mode commands.**
- **Interactive debugging.**

Chapter 10

Modeling Memories

Modeling Memories

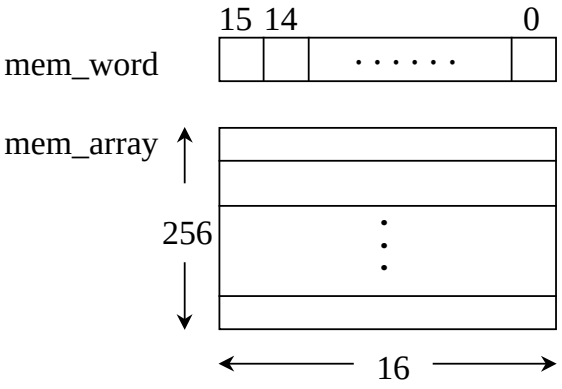
- **How to model memories with Verilog HDL**
  - ROM
  - RAM

Memories Declaration

- **The Verilog HDL models memories as an array of registers.**

```
reg [7:0] mema [ 0:255];            // 256 x 8-bit memory array
reg [7:0] PLA ['hFFFE:'hFFFF];  // 2 x 8-bit memory array
```

```
parameter wordsize = 16,
          memsize = 256;
reg [wordsize-1:0]
  mem_word,
  mem_array [memsize-1:0];
```



Memory Addressing

A memory element is addressed by giving the location of the memory array.

```
.....
reg [8:1] mem_word, mema [0:255];
.....
// assign a value to memory element
mema[0] = 8'b0010_0011;
.....
// display the content of 6th memory element
$displayb(mema[5]);
.....
// display the msb of the 6th memory element
mem_word = mema[5];
$displayb( mem_word[8] );
```

Modeling a ROM

```
module ROM(DOUT, ADDR, CLK);
  output [5:0] DOUT;
  input   CLK;
  input   [3:0] ADDR;
  reg [5:0] DOUT, rom_core [15:0];

  initial begin
    rom_core[0] = 6'b001100;
    ......
    rom_core[15] = 6'b100100;
    end

  always @(posedge clk)
    DOUT = rom_core[ADDR];
endmodule
```

specify the ROM code.

addressing the ROM.

Loading a Memory Array

- **You can also use *$readmem* system task to load values from a text file into memory array.**

```
$readmem<base>("<file_name>", <mem_name>, <start>?, <finish>?);

reg [7:0] mem [1:256];

initial $readmemh("mem.data", mem);
initial $readmemb("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 256);
```

File Format for $readmemb and $readmemh

```
.....
reg [0:7] mem [0:1023];
.....
$readmemb("memfile.txt", mem);
```

UNIX text file memfile.txt

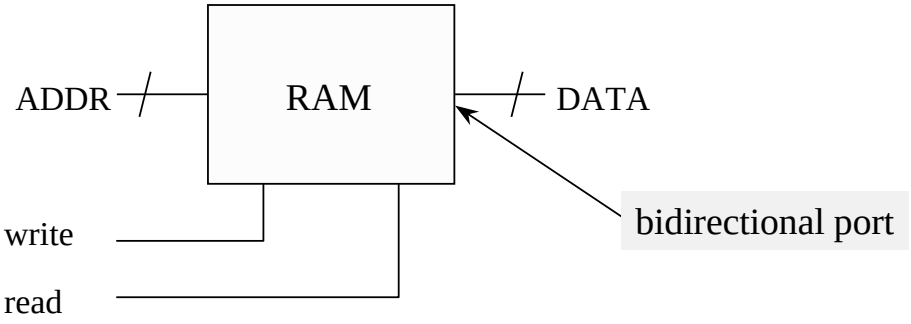
```
0000_0000
0110_0001 0011_0010
// addresses 3 to 255 are not defined
@100
1111_1100
// address 257 to 1022 are not defined
@3FF
1110_0010
```

Memory contents

00000000	0
01100001	1
00110010	2
	⋮
11111100	256
	⋮
11100010	1023

Modeling a RAM

- **Modeling an asynchronous RAM.**



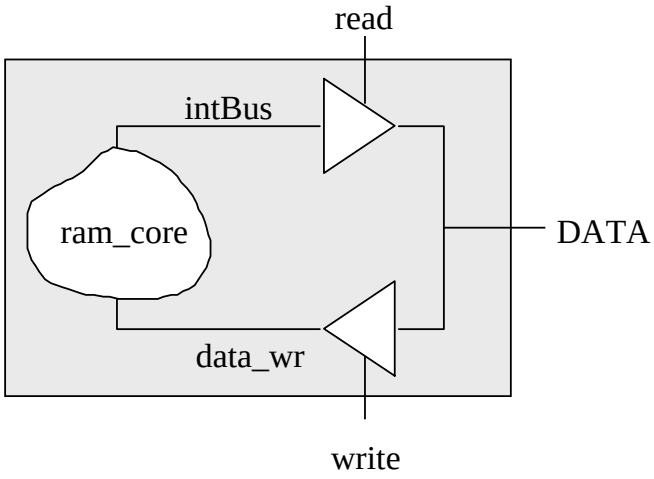
read	write	RAM mode	DATA
1	0	read mode	RAM[ADDR]
0	1	write mode	write data
0	0	disable	high Z
1	1	ERROR	ERROR

Modeling Bidirectional Ports

- **The bidirectional port needs to be declared using the keyword *inout*.**
- **Inout ports must follow the following rules**
  - The inout port must be driven by a net but not a register.
  - The inout port must drive a net but not a register.
- **Logic gates needs to be built around the *inout* port to ensure proper operation.**

Modeling Bidirectional Ports (cont.)

```
module ram2x2( DATA, ADDR, read, write );
  inout [1:0] DATA;
  input [1:0] ADDR;
  input read, write;
  reg [1:0] intBus, ram_core [3:0];
  wire [1:0] data_wr;

  bufif1 ( DATA[0], intBus[0], read );
  bufif1 ( DATA[1], intBus[1], read );
  bufif1 (data_wr[0], DATA[0], write);
  bufif1 (data_wr[1], DATA[1], write);

  always @(posedge read)  // read mode
    assign intBus = ram_core[ADDR];

  always @(posedge write)    // write mode
      ram_core[ADDR] = data_wr;
endmodule
```



Modeling Bidirectional Ports (cont.)

```
module ram2x2( DATA, ADDR, read, write );
  inout [1:0] DATA;
  input [1:0] ADDR;
  input read, write;
  reg [1:0] ram_core [3:0];

  assign DATA[1:0] = (read == 1) ? ram_core [ADDR] : 2'bz;

  always @(posedge write)
    ram_core [ADDR] = DATA;
endmodule
```


Summary

- **Modeling a read only memory (ROM).**
- **Modeling a random access memory (RAM).**
- **Handling bidirectional ports in Verilog.**

Chapter 11

Tasks and Functions

User Defined Tasks and Functions

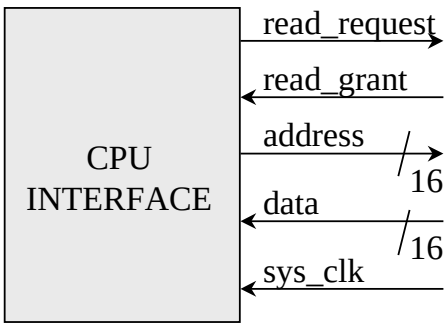
- **Learn the use of Verilog tasks and functions.**
- **Learn the named event and blocks.**
- **Enabling and disabling named events and tasks.**

Verilog Tasks and Functions

- **Both functions and tasks let you execute common procedures from different places in a description.**
- **They make it easier to read and debug your Verilog source descriptions.**
- **Differences:**

Functions	Tasks
A function cannot have timing controls.	A task can contain delays.
A function cannot enable a task.	A task can enable other tasks and functions.
A function must have at least one input argument.	A task can have zero or more arguments of any types.
A function returns a single value.	A task does not return a value.

Specifications of a CPU Interface




```
module cpu_interface(<port>);
  <port_declaration>;
  reg [16:1] IR, PC, address;

  always @(posedge sys_clk)
    begin
      if(read_request == 1)
        //  call the read task
      //  call function to swap bits
      //  call named event
    end
  //  abort read
endmodule
```

Specifications of a CPU Interface (cont.)

- **CPU asserts read_request and waits for read_grant.**
- **When read_grant is asserted, the CPU places the address on the address bus and reads the data.**
- **After reading the data, CPU deasserts the read_request and drives the address to high impedance.**
- **The bits of the data need to be swapped.**
- **If read_grant is deasserted while read_request is asserted, abort the read.**

Verilog Task

```
.....
always @(posedge sys_clk)
  begin
    if (read_request == 1)
      begin
        read_mem(IR, PC);
        // Event and function calls
      end
  end

task read_mem;
  output [15:0] data_in;
  input [15:0] addr;
  @(posedge read_grant)
    begin
      ADDRESS = addr;
      #15 data_in = data;
    end
endtask
......
```

Verilog Function

```
always @(posedge sys_clk)
  begin
    ......
    IR = swap_bits( IR );
    ......
  end

function [16:1] swap_bits;
  input [16:1] in_vec;
  reg [15:0] temp_reg;
  integer i;
  begin
    for ( i=16; i>=1; i=i-1 )
      temp_reg[16-i] = in_vec[i]
    swap_bits = temp_reg;
  end
endfunction
```


Named Event

```
event read_complete;
......
  always @(posedge sys_clk)
    begin
      if(read_request == 1)
        begin
          // task and function calls
          -> read_complete;
        end
    end
  always @(read_complete)
    begin
      read_request = 1'b0;
      ADDRESS = 16'bZ;
      $display("Data received, read is complete");
    end
```

Disabling Named Blocks and Tasks

```
.....
always @(posedge sys_clk)
  begin
    if (read_request == 1)
      read_mem( IR, PC );
  end
  .....
always @(negedge read_grant)
  if (read_request == 1)
    disable read_mem;
.....
task read_mem;
  .....
  -> read_complete;
endtask
.....
```


User-Defined Function Example

```
function [31:0] power_2;
  input [31:0] operand;
  integer i;

  begin
    power_2 = 1;
    for(i=0; i<operand; i=i+1)
      power_2 = 2 * power_2;
  end
endfunction
```

User-Defined Function Example (cont.)

```
function [31:0] factorial;
  input [3:0] operand;
  reg [3:0] index;

  begin
    factorial = operand ? 1 : 0;
    for (index = 2; index <= operand; index = index + 1)
      factorial = index * factorial;
  end
endfunction
```


Summary

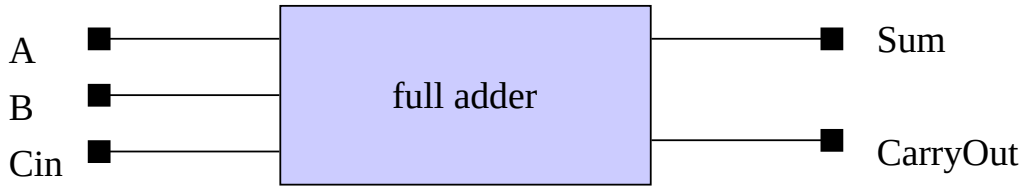
- **User defined tasks and functions.**
- **User defined event**
- **Disable user defined tasks and named block.**

Chapter 12

Specify Blocks

Specify Blocks

- **What is a specify block?**
  - specify block let us add timing specifications to paths across a module.

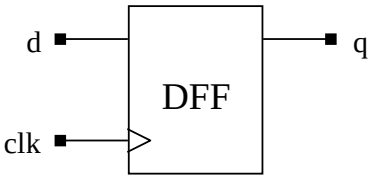


Tlh_A_to_Sum = 1.2
Thl_A_to_Sum = 2.0
.....

Tlh_B_to_Sum = 1.5
Thl_B_to_Sum = 2.2
.....

Terms and Definitions

- **Module Path**
- **Path Delay**
- **Timing Check**



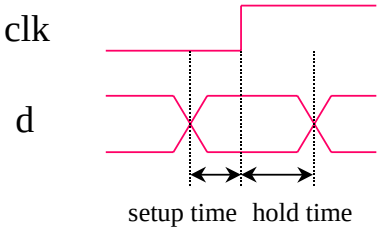
module path

d → q
clk → q

path delay

Tlh_clk_q = 4.0
Thl_clk_q = 5.0

timing check



Features of Specify Block

- **In specify block you can do the following modeling tasks:**
  - describe various paths across the module.
  - assign delays to those paths.
  - perform timing checks.
- **Specify block syntax**

  specify
  <specify_item>*
  endspecify
- **You can declare parameters in specify block.**

  specparam <parameter_assignment><, <parameter_assignment>>*

Example

```
module DFF (q, d, clk);
  input d, clk;
  output q;
  reg notifier;

  UDP_DFF       G4(q, d, clk, notifier);
  specify
    specparam
      InCap$d = 0.024;
      ......
      (cp => q) = (0.390, 0.390);

      $setup( d, posedge cp, Tsetup$d_cp, notifier);
      $hold(posedge cp,  d, Thold$d_cp, notifier);
      .........
  endspecify
endmodule
```

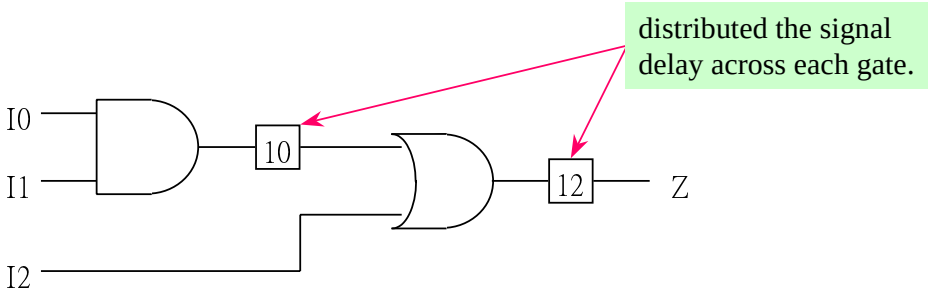
specify parameter

declare module path delays

perform timing check

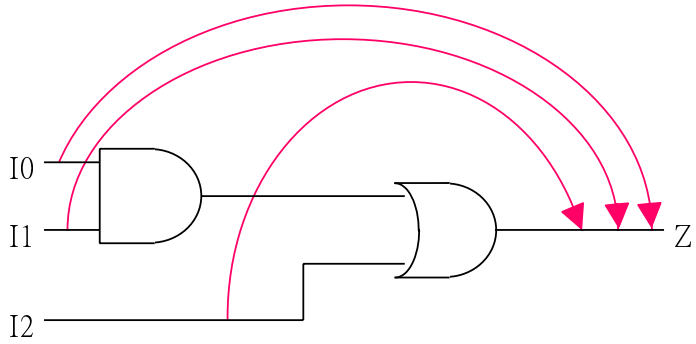
Delay Modeling Options

- **Verilog HDL can describe two types of delays:**
  - Module path delays.
  - distributed delays.
- **Distributed delays:**



Delay Modeling Options (cont.)

- **Module path delays:**



module path delay from I0 to Z = 22
module path delay from I1 to Z = 22
module path delay from I2 to Z = 12

Module Path Declaration

- **Module path declaration**
  - parallel connection
    (source => destination)
  - full connection
    (source *> destination)
- **Example**

(I0, I1, I2 *> Z) // full connection
is equivalent to
(I0 => Z) // parallel connection
(I1 => Z)
(I2 => Z)

Specify Module Path Delays

- **Syntax :**

  (module_path) = (<module_path_delay_specification>);

- **one delay value**

  (I0 => Z) = 12; // specify all transition delay value

- **two delay value**

  (I0 => Z) = (12, 10); // specify rising and falling transistion delay values

- **three delay value**

  (I0 => Z) = (12, 10, 8); // specify rising transistion, falling transistion, and Z transistion delay values

Specify Module Path Delays (cont.)

- **six delay value**

  (I0 => Z) = (12, 10, 8, 8, 9, 8); // specify 0→1, 1→0, 0→Z, Z→1, 1→Z, and Z→0 transistion delay values

- **specify delay in minimum : typical : maximum format.**

  (I0 => Z) = (9 : 12 : 15, 9 : 10 : 12, 6 : 8 : 13)

Examples

```
module andor ( Z, I0, I1, I2);
  output Z;
  input   I0, I1, I2;

  and  G0(net1, I0, I1);
  or    G1(Z, net1, I2);

  specify
    (I0 => Z) = 19 : 22 : 29;
    (I1 => Z) = (22, 19);
    (I2 => Z) = (9 : 12 : 15, 7 : 10 : 13);
  endspecify
endmodule
```

```
module andor ( Z, I0, I1, I2);
  output Z;
  input   I0, I1, I2;

  and  G0(net1, I0, I1);
  or    G1(Z, net1, I2);

  specify
    (I0, I1, I2 *> Z) = 19 : 22 : 29;
  endspecify
endmodule
```


Declarating Parameters in Specify Block

- **Syntax**

  specparam <parameter_assignment>;

- **Example**

```
module andor ( Z, I0, I1, I2);
  output Z;
  input   I0, I1, I2;

  and  G0(net1, I0, I1);
  or    G1(Z, net1, I2);

  specify
    specparam  tRise_I_Z = 22,
               tFall_I_Z = 18;
    ( I0, I1, I2 *> Z ) = ( tRise_I_Z,  tFall_I_Z );
  endspecify
endmodule
```

declare module path delay values as parameters in specify block.

Restrictions on Module Paths

- **All destinations of a module path must be driven by a gate.**

```
module INA(out,in);
  output out;
  reg    out;
  input  in;

  always @(in) out = ~in;

  specify
    specparam  R_Ramp$in$out = 2,
               F_Ramp$in$out = 1;
    (in => out) = (R_Ramp$in$out, F_Ramp$in$out);
  endspecify
endmodule
```

Verilog Error Message
Error! Register encountered in specify block statement

modified :

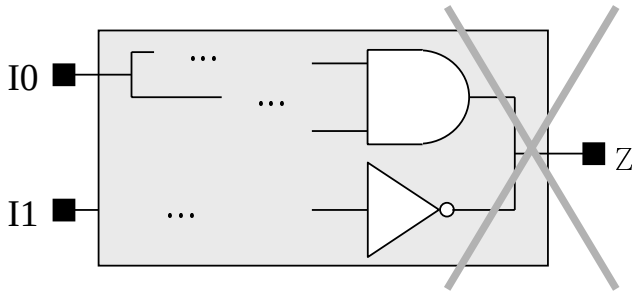
```
.....
reg not_1;
.....
always @(in) out_1 = ~in;
buf (out, out_1);
.....
```


Restrictions on Module Paths (cont.)

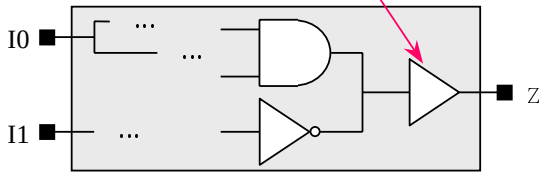
- **Path destinations can have only one driver inside the module.**

Error! Multiple path delays defined to node <path_destination>;
Path delay outputs must have only one driver within the module.

```
.....
specify
  ......
  (I0 => Z) = (tRise$I0$Z, tFall$I0$Z);
  (I1 => Z) = (tRise$I1$Z, tFall$I1$Z);
endspecify
```



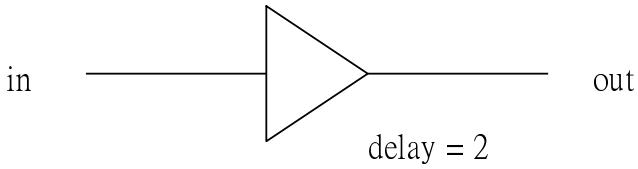
You can remove this error by adding a buffer between the module path destination and the driver.

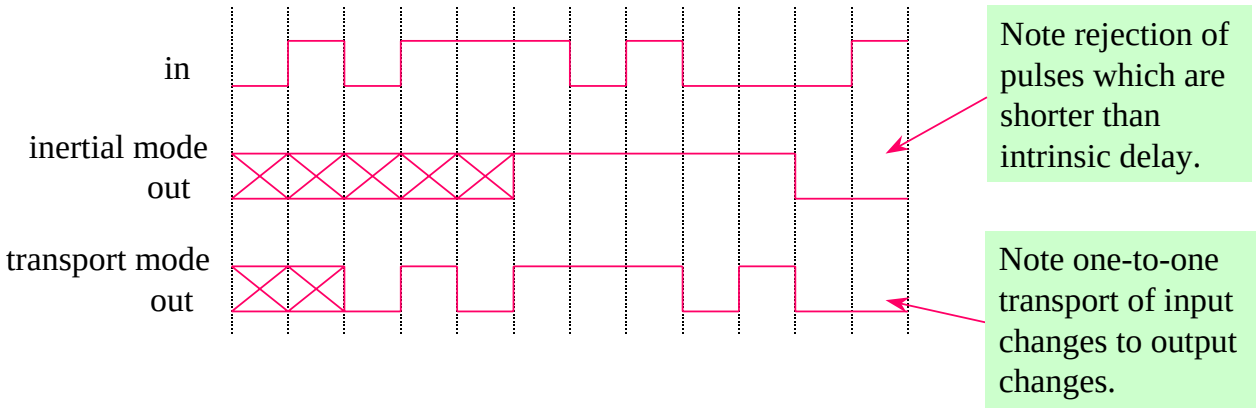


Inertial vs. Transport Delay Models

- **Verilog by default supports the *inertial delay* mode.**
- **In *inertial delay* mode, Verilog does not transmit the pulses with a duration shorter than the element's delay.**
- **In *transport delay* mode, Verilog transmit pulses of shorter duration than the element's delay.**
- **You can invoke Verilog-XL with +*transport_path_delays* command line option to simulate your design with *transport delay* model.**

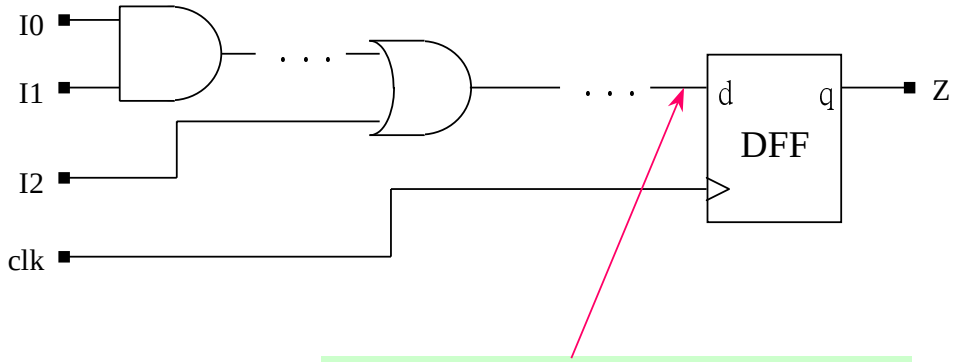
Inertial vs. Transport Delay Models (cont.)





Timing Checks in Verilog

- **Use timing checks to verify the timing of your design.**



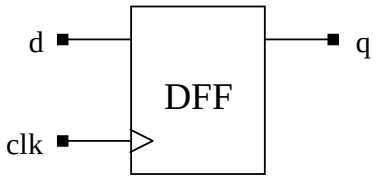
Due to the long path from primary input to *DFF*, the signal in *d* input of *DFF* may violate the setup time or hold time limitation of *DFF*.

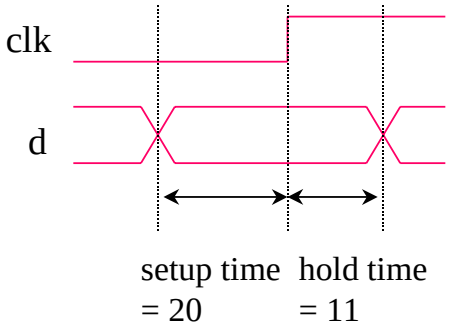
Timing Checks in Verilog (cont.)

- **Timing checks performed by Verilog-XL are**
  - setup
  - hold
  - pulse width
  - clock period
  - skew
  - recovery
- **If there are timing violation in your design, Verilog-XL reports a warning and does not affect the output of the module.**

Timing Checks in Verilog (cont.)

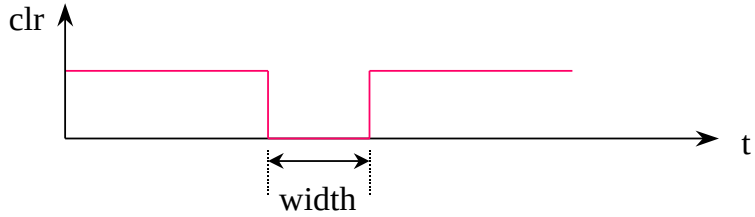
- **Setup time and hold time checks:**
  - $setup (data, posedge clk, 20, notifier);
  - $hold (posedge clk, data, 11, notifier);
  - $setuphold (posedge clk, data, 20, 11, notifier);



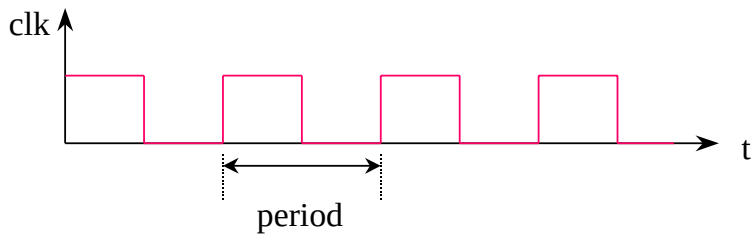


Timing Checks in Verilog (cont.)

- **Pulse width check**
  - $width(negedge clr, 12, 0, notifier);

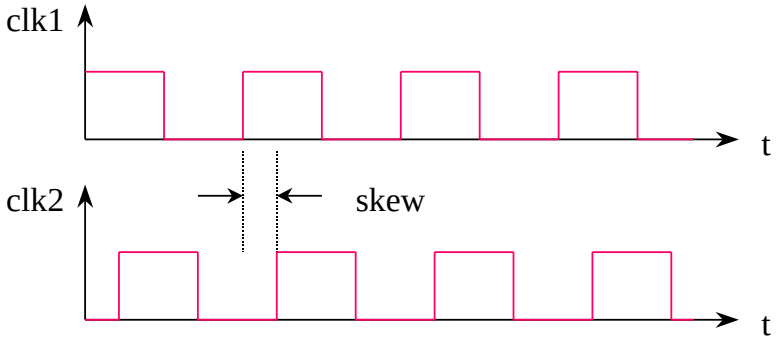


- **Clock period check**
  - $period(posedge clk, 15, notifier);



Timing Checks in Verilog (cont.)

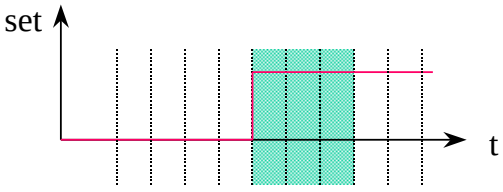
- **Skew check**
  - $skew(posedge clk1, posedge clk2, 14, notifier);



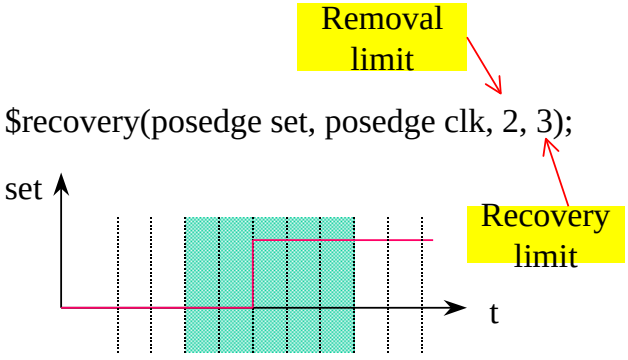
Timing Checks in Verilog (cont.)

- **Recovery check**

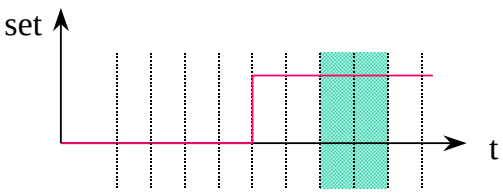
$recovery(posedge set, posedge clk, 3);



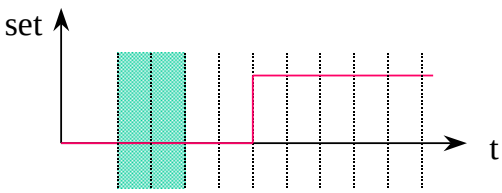
$recovery(posedge set, posedge clk, 2, 3);



$recovery(posedge set, posedge clk, -2, 4);



$recovery(posedge set, posedge clk, 4, -2);



if *posedge clk* does not appear in this interval, Verilog-XL reports a timing violation.

Notifiers in Timing Checks

- **The expected behavior of some logic might turn the output to undefined when a timing violation occurs.**
- **When a timing-check violation occurs, Verilog will toggle the *notifier* register.**

```
reg notifier;
.....
$setup (data, posedge clk, 20, notifier);
.....
```

- **To turn the output to unknow, you can**
  - specify an additional *notifier* port in sequential UDP.
  - use *notifier* to force UDP's output to turn to unknow when *notifier* toggles.

Summary

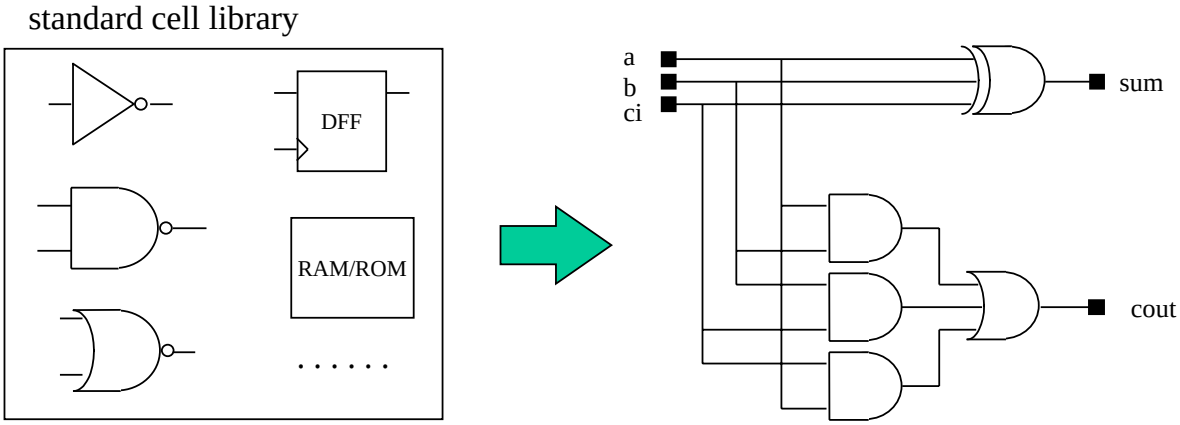
- **Specify block**
  - specify module path delays
  - specify parameters
  - timing checks

Chapter 13

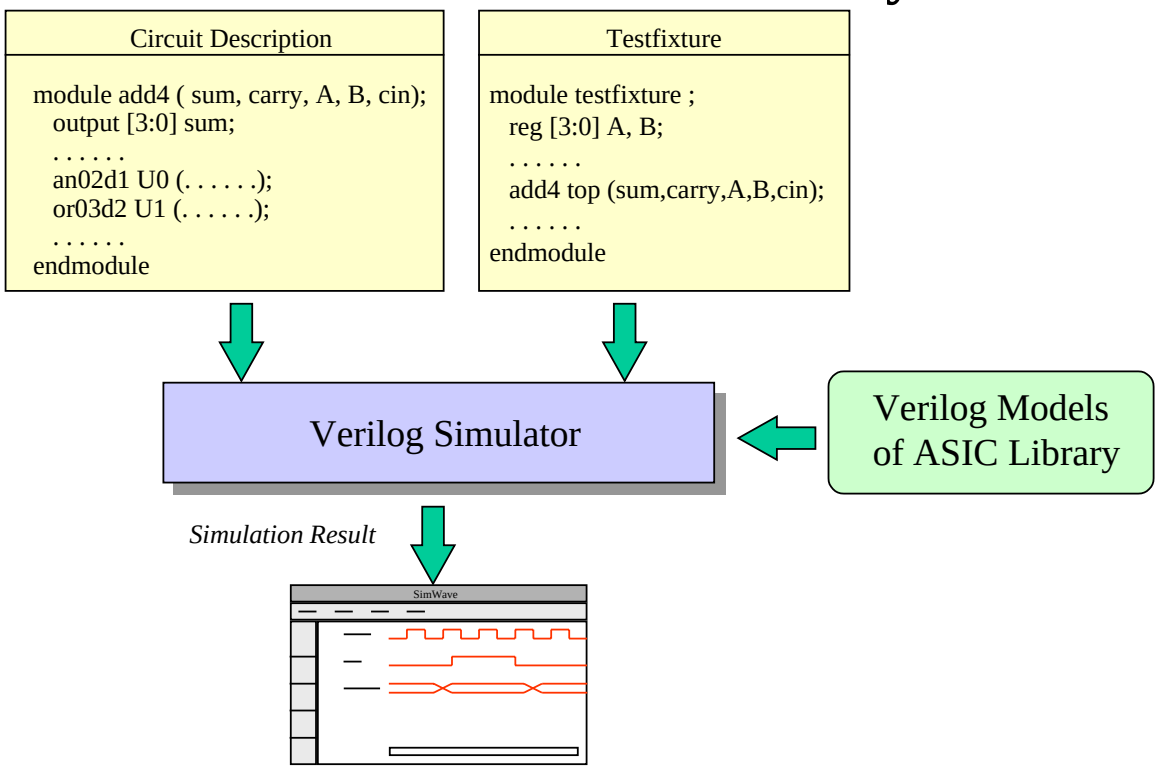
Modeling ASIC Libraries

Overview

- **Circuit designs are often made up of components from a library, such as**
  - standard cells from a specific ASIC vendor.
  - TTL components from 7400 series library.



Simulation with ASIC Library



Modeling ASIC Libraries

- **Each cell in the library has the following parts:**
  - functional
  - timing and timing constraints (setup time, hold time, etc.)
  - technology
- **How to create their Verilog models?**
  - You must describe the function of your cells.
  - You must describe the timing informations of your cells.

Functional Modeling of Library Cells

- **Functionality of a cell can be described at two levels:**
  - structural
  - behavior

  avoid mixing these constructs
- **Structural description**
  - use Verilog primitives to model combinational cells.
  - use User-Defined Primitives (UDPs) to model sequential cells
- **Behavioral description**
  - used to model RAMs, ROMs, and other macro cells.

Verilog Model Example --- Combinational

- **The Verilog model of Compass cell *an02d1*.**

```
`timescale 1 ns / 10 ps
module an02d1 (z, a1, a2);
  input a1, a2;
  output z;

  and              G2(z, a1, a2);

  specify
    specparam cell_count    = 1.500000;
      ......
    // Pin-to-pin delays
    (a1=>z)=(0.111:0.170:0.274, 0.150:0.230:0.370);
    (a2=>z)=(0.117:0.180:0.290, 0.130:0.200:0.322);
  endspecify
endmodule
```

specify *an02d1*'s function.

specify *an02d1*'s timing information.

User Defined Primitives

- User Defined Primitives (UDPs) provide a way to let users define their own primitive elements.
- The behavior of UDPs is described in a truth table.
- UDPs can represent sequential as well as combinational elements.

UDP Features

- UDPs can have only one output.
- The state of UDP's output can be 0, 1, or X. Z logic value is not supported.
- Combinational UDPs can have 1 to 10 inputs.
- Sequential UDPs can have 1 to 9 inputs.
- All ports must be scalar.
- UDPs does not support bidirectional ports.
- The memory requirements increase dramatically as the number of inputs increase from 5.

Declaration of UDP

- **The syntax of UDP definition is as follows:**

primitive <UDP_name> (<port_name_list>);
<port_declaration>;

table
<table_definition>
endtable
endprimitive

Decalre the truth table of this User Defined Primitive.

table definition

```
<input_compostion_1>  :  <out_value_1>
<input_compostion_2>  :  <out_value_2>
<input_compostion_3>  :  <out_value_3>
......
```


Combinational UDP Example

2 to 1 multiplexer

This is the name of the primitive.

The output port must be the first port.

Table columns are inputs in order declared in primitive statement, colon, output, followed by a semicolon.

Any combinations of inputs which is not specified in the table will produce an X at the output.

These entries reduce pessimism.

```
primitive MUX_2_1(o, a, b, s);
  output o;
  input  a, b, s;
  table
  // a b s : o
    0 ? 1 : 0 ;
    1 ? 1 : 1 ;
    ? 0 0 : 0 ;
    ? 1 0 : 1 ;
    0 0 x : 0 ;
    1 1 x : 1 ;
  endtable
endprimitive
```

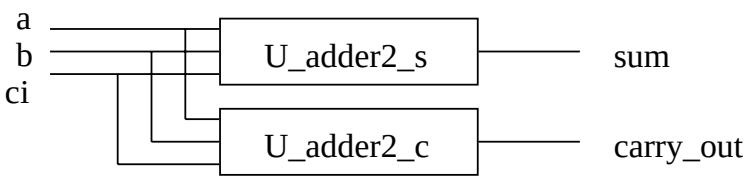
Combinational UDP Example (cont.)

You can implement a full adder with two combinational UDPs.

```
primitive U_adder2_s(s, a, b, ci);
  output s;
  input  a, b, ci;
  table
  // a  b  ci  :  s
     0  0  0   :  0;
     0  0  1   :  1;
     0  1  0   :  1;
     0  1  1   :  0;
     1  0  0   :  1;
     1  0  1   :  0;
     1  1  0   :  0;
     1  1  1   :  1;
  endtable
endprimitive
```

```
primitive U_adder2_c(co, a, b, ci);
  output co;
  input  a, b, ci;
  table
  // a  b  ci  :  co
     1  1   ?  :   1;
     1  ?   1  :   1;
     ?  1   1  :   1;
     0  0   ?  :   0;
     0  ?   0  :   0;
     ?  0   0  :   0;
  endtable
endprimitive
```

If your macro cell has two or more output, then you can use two or more UDPs to model it.



Level-Sensitive Sequential UDPs

level-sensitive latch

```
primitive latch (q, clock, data);
  output q;
  reg q;
  input clock, data;
  initial q = 1'b1;
  table
  // clock data   q    q+
       0     1  :  ?  :  1 ;
       0     0  :  ?  :  0 ;
       1     ?  :  ?  :  - ;
  endtable
endprimitive
```

you must declare a storage element to store the state variable.

The output is initialize to 1'b1;

- means no change

next state

current state

Edge-Sensitive Sequential UDPs

edge-sensitive D flip-flop

```
primitive d_edge_ff(q, clock, data);
  output q; reg q;
  input  clock, data;

  table
  // clock  data  q    q+
    (01)     0  :  ?  :  0 ;
    (01)     1  :  ?  :  1 ;
    (0x)     1  :  1  :  1 ;
    (0x)     0  :  0  :  0 ;
  // ignore negative edge of clock
    (x0)     ?  :  ?  :  - ;
  // ignore data changes on steady clock
       ?     (??) :  ?  :  - ;
  endtable
endprimitive
```

(01) means a rising edge on input.
(10) means a falling edge on input.

Summary of Symbols

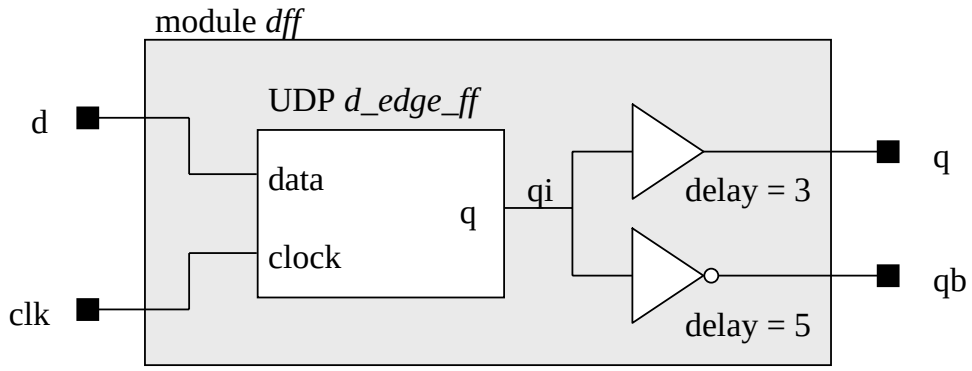
- **In the following is the summary of symbols that are valid in the table part of a UDP definition.**

Symbol	Interpretation	Notes
-	no change	can only be given in the output field of a sequential UDP
?	iteration of 0, 1, and x	cannot be given in output field
b	iteration of 0 and 1	cannot be given in output field
r	same as (01)	rising edge on input
f	same as (10)	falling edge on input
p	iteration of (01), (0x), (x1)	potential positive edge on input
n	iteration of (10), (1x), (x0)	potential negative edge on input
*	same as (??)	any value change on input

Instantiate an UDP

```
module dff(q, qb, clk, d);
  output q, qb;
  input   clk, d;

  d_edge_ff  G0(qi, clk, d);
  buf  #3  G1(q, qi);
  not  #5  G2(qb, qi);
endmodule
```

Use an pre-defined UDP to describe the function of an edge-triggered D flip-flop.



Verilog Model Example --- Sequential

```
module dfntnb (q, qn, d, cp);
  input d, cp;
  output q, qn;
  reg notifier;

  not             G3(qn, N8);
  UDP_DFF         G4(q, d, cp, N6, N6, notifier);
  pullup          G6(N6);
  UDP_DFF         G8(N8, d, cp, N6, N6, notifier);

  specify
    . . . . . .
    (posedge cp => (q +: d)) = (0.390:0.600:0.966, 0.390:0.600:0.966); // Pin-to-pin delays
    . . . . . .
    $setup( d, posedge cp, Tsetup$d_cp, notifier);      // setup timing checks
    . . . . . .
  endspecify
endmodule
```

Use two UDPs to describe the function of sequential cell *dfntnb*.

specify timing constraints

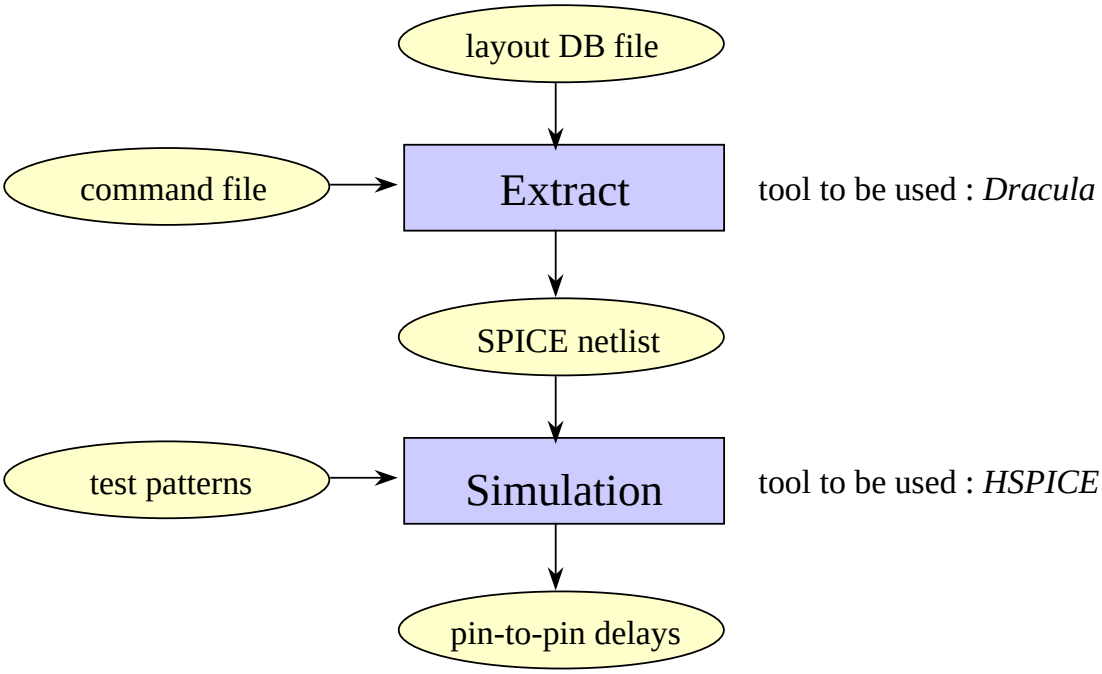
Modeling Timing for Library Cells

- **The timing of a cell may consist of**
  - delay modeling
  - timing checks

Delay Modeling

- **Two methods to model delays in Verilog**
  - distributed delays
  - path delays in specify block
- **Most ASIC library cells are modeled using path delays**
  - easy to model
  - functional description can be simpler
  - path delays are more accurate
  - modular cells

How to get the Cell's Timing Information



Example

```
module INA(Z,I);
    output Z;
    input I;

    not g1(Z,I);

    specify
      specparam A0lh$I$Z = 0.287;
      specparam A0hl$I$Z = 0.184;
      ......

      ( I *> Z ) = ( A0lh$I$Z,A0hl$I$Z );
    endspecify
endmodule
```

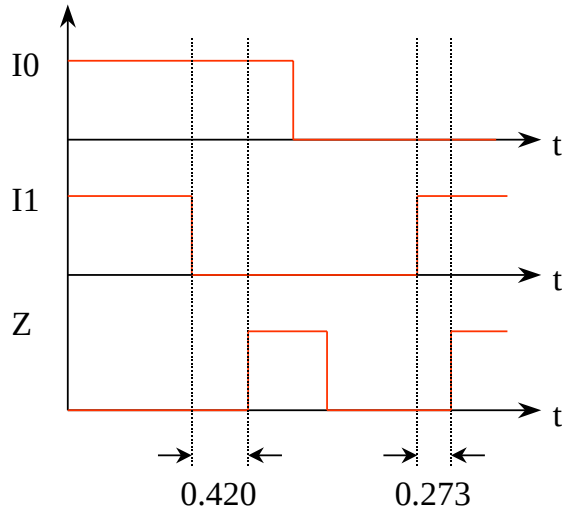
functional description

specify pin-to-pin delay

State-Dependent Path Delays

- **The path delays through the cells can be dependent on the state of the cell's inputs.**

```
module XOR2(Z, I0, I1);
  output Z;
  input  I0, I1;

  xor G0(Z, I0, I1);
  specify
    if (   I1) (I0 => Z) = (0.350, 0.330);
    if ( ~I1) (I0 => Z) = (0.273, 0.273);
    if (   I0) (I1 => Z) = (0.420, 0.420);
    if ( ~I0) (I1 => Z) = (0.273, 0.273);
  endspecify
endmodule
```



Notifiers in Timing Checks

- **The expected behavior of some logic might turn the output to undefined when a timing violation occurs.**
- **You can combine the *notifier* register with *UDPs* to achieve the above behavior.**

```
module dff(q, qb, clk, d);
  output q, qb;
  input  clk, d;
  reg    notifier;

  d_edge_ff  G0(qi, clk, d, notifier);
  buf  #3  G1(q, qi);
  not  #5  G2(qb, qi);
  specify
    $setup (d, posedge clk, 12, notifier);
    $hold (posedge clk, d, 5, notifier);
  endspecify
endmodule
```

declare the notifier register

when timing violation occurs, Verilog-XL will toggle the *notifier* register and it force the *qi* to become to unknow.

Notifiers in Timing Checks (cont.)

```
primitive d_edge_ff(q, clock, data, notifier);
  output q; reg q;
  input   clock, data, notifier;

  table
  // clock  data  notifier   q    q+
    (01)     0       ?     : ? : 0 ;
    (01)     1       ?     : ? : 1 ;
    (0x)     1       ?     : 1 : 1 ;
    (0x)     0       ?     : 0 : 0 ;
    (x0)     ?       ?     : ? : - ;
      ?    (??)      ?     : ? : - ;
  // when notifier changes, send output to unknow
      ?      ?       *     : ? : x ;
  endtable
endprimitive
```

An additional port is specified in this UDP. This port will force the UDP's output to unknow value whenever this input port change values.

Whenever *notifier* changes values, the next state of *q* will become to a unknow state.

Summary

- **ASIC library**
- **Verilog models of an ASIC library**
- **user-defined primitives (UDPs)**
- **notifiers in timing check**

Chapter 14

Simulation with COMPASS Cell Library

Highlight of COMPASS Cell Library

- **For TSMC 0.6um 1P3M process.**
- **5-volt power supply.**
- **High performance core cell library.**
- **High density core cell library.**
- **I/O pad library.**
- **Compiler cells (ROM, RAM, datapath).**

Highlight of COMPASS Cell Library (cont.)

- **Two types of core cell library**
  - high performance cell library : *cb60hp231d*
  - high density cell library : *cb60hd231d*

- **Contents of Compass core cell library**
  - combinational logic
    – inv, buf, 3-state buf, clock buf, nand, nor, and, or, aoi, oai,
  - sequential logic
    – latch, SR latch, D flip-flop, JK flip-flop, scan latch, ...
  - special function
    – decoder, multiplexer, adder, substractor, counter

Highlight of COMPASS Cell Library (cont.)

- **I/O cell library name : *cb60io420d***
  - 70 I/O cells.
- **Contents of *cb60io420d***
  - input pad, output pad, bi-directional pad, crystal oscillator pad, ...
  - power pad

What Does CIC Support?

- **On-line document**
  - get it from CIC's ftp site

  ftp.cic.edu.tw://

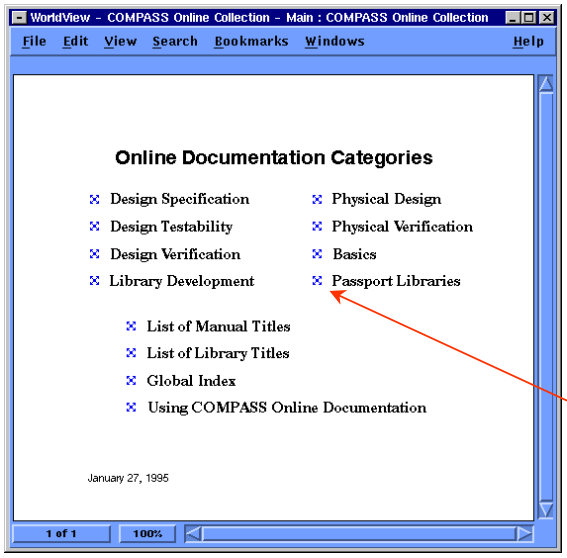
  pub/doc/manual/CompassOnline/cmpdoc_sun_v3.tar.Z
- **Design kit**
  - Synopsys design library
  - Verilog simulation model
  - Verilog delay calculator
  - OPUS database
  - Dracula command file

Usage of COMPASS Cell Library

- **To use Compass cell library to build up your design, you must**
  - read COMPASS on-line document to understand what cells do you have.
  - decide which cell do you want to use.
  - refer to the Verilog simulation model and build up your design.
  - use COMPASS provided Verilog simulator to simulate your design.

COMPASS On-line Document

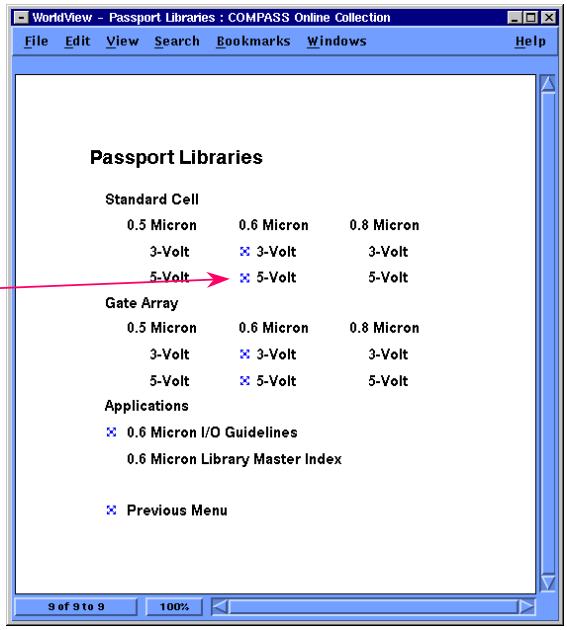
- **Type *iview* under *UNIX* prompt to invoke COMPASS on-line document.**



click *Passport Libraries* to display the detail categories of COMPASS passport libraries.

COMPASS On-line Document (cont.)

Click *5-Volt* to see detail categories of COMPASS 5-Volt cell library.

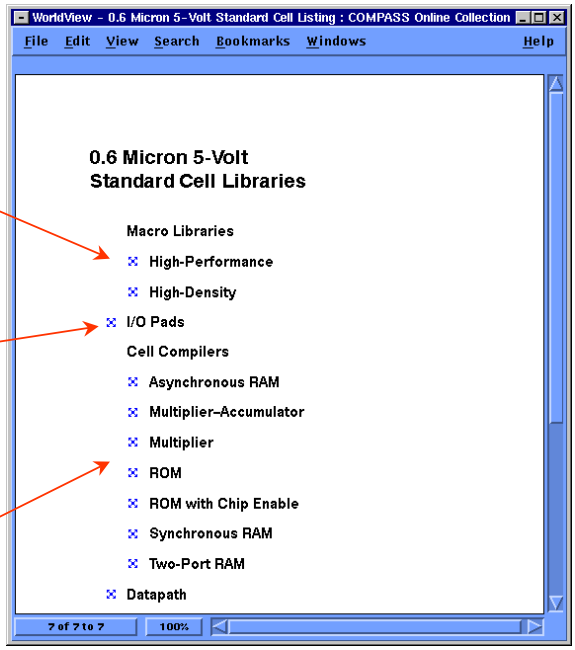


COMPASS On-line Document (cont.)

For detail informations of core cell libraries, click here.

For detail informations of I/O pads libraries, click here.

For detail informations of compiler cells, click here.



Verilog Simulation Model

- **COMPASS supports the following Verilog simulation models (under the *<your_path>/CIC_CBKIT_V4/Verilog* directory)**
  - linear model
    cb60hp231d.vmd
    cb60hd231d.vmd
    cb60io420d.vmd
  - non-linear model (ISM model)
    cb60hp231d.ismvmd
    cb60hd231d.ismvmd
    cb60io420d.ismvmd
  - user-defined primitives
    <your_path>/CIC_CBKIT_V4/Verilog/cb60hp231d/cells/support/udps.vmd
    <your_path>/CIC_CBKIT_V4/Verilog/cb60hd231d/cells/support/udps.vmd

Build up Your Design

- **Connect each COMPASS cell correctly by**
  - port order mapping

in cb60hp231d.vmd:

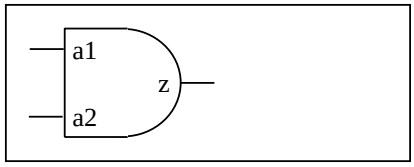
```
......
module an02d1 (z, a1, a2);
   input a1, a2;
......
```

in your design file:

```
......
  an02d1 U0(and_o, and_i1, and_i2);
......
```

  - name mapping

in on-line document



in your design file:

```
......
  an02d1 U0(.z(and_o), .a1(and_i1), .a2(and_i2));
......
```


CIC support two Simulation Method

- **Simulation with COMPASS Delay Calculator.**
- **Simulation with Cadence Central Delay Calculator.**

Simulation with COMPASS Delay Calculator

- **COMPASS supported Verilog delay calculator.**
  - PLI-based delay calculator.
  - Simulate your gate-level design more accurately.
  - Verilog execution file linked with Compass delay calculator

    <Your_path>/CIC_CBKIT_V4/Verilog/bin/vlog_mercury

- **Setup simulation evnironment for COMPASS cell library.**
  - read the *readme* file

    <your_path>/CIC_CBKIT_V4/Verilog/00.README

Prepare for Your Testfixture file

- **Add the following system task in your testfixture file to invoke COMPASS delay calculator.**

  $VTOOLS_decal("in_file", "product.xch");

- **Example testfixture file**

```
module testfixture;
  . . . . . .
  initial
    $VTOOLS_decal("in_file", "product.xch");
  . . . . . .
endmodule
```

Simulation Control File

- ***in_file* is the Verilog delay calculator input control file, it must be in the same directory with stimulus.**
- **A sample *in_file* is**

```
calc_mode estimate;
library_name cb60hp231d;   // or cb60hd231d
process typical;    //  can be set as slow, typical, fast
temperature 25;    //   can be set between -50 ~ 150
voltage 5.0;          //   can be set between 4.5 ~ 5.5
num_sites 1000;  /*   it determines which wire load model to use,
                      1000 is for less than 1000 gates design,
                      2000 is for 1000 ~ 2000 gates design,
                      the others can be set are 4000, 8000, 16000, 30000, 46000,
                      77000, 110000, 163000, 198000, 246000.   */
```

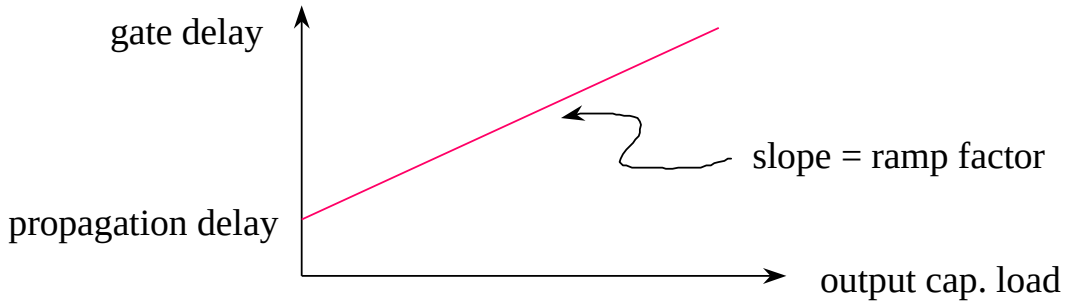

Gate Level Simulation
- Compass 0.6um Cell Library

- **Simulation with COMPASS Verilog simulation model.**
  - linear model
    - – simulation speed is faster when compared with non-linear model simulation.
    - – more inaccurate in circuit timing.
  - non-linear model (ISM model)
    - – more accurate in timing than linear model simulation.
    - – simulation speed will be greatly slow down.
- **Use the *+ism* command line option to switch the simulation mode from linear model simulation to non-linear model simulation.**

Gate Level Simulation
- Compass 0.6um Cell Library (cont.)

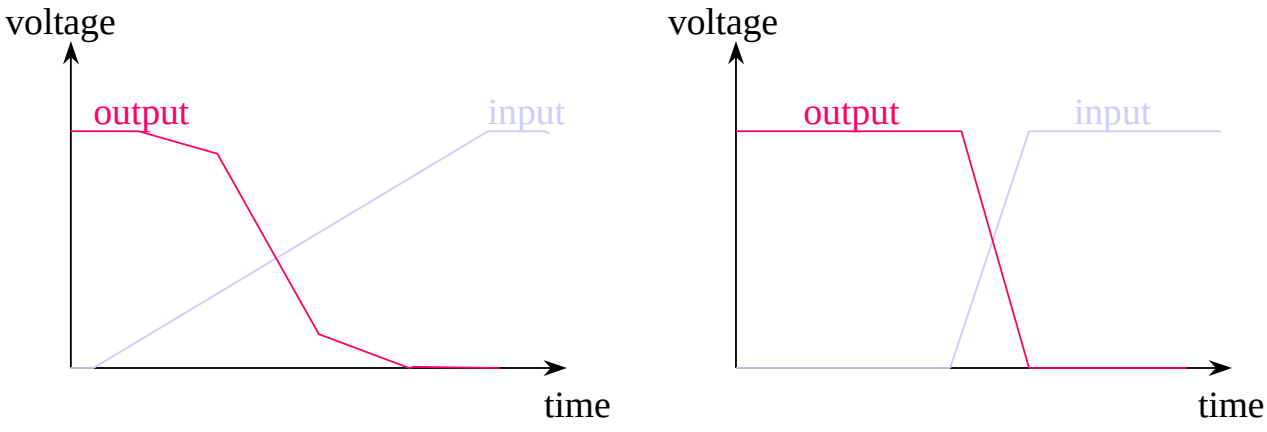
- **Linear model**
  - it use prop/ramp model to estimate gate delay.
  - In this model,

    gate_delay = Propagation_delay + Ramp_factor * Load



Gate Level Simulation
- Compass 0.6um Cell Library (cont.)

- **ISM model**
  - It take input slope into account to estimate the gate delay.
  - More accurate than linear model, but needs more CPU time.



Gate Level Simulation
- Compass 0.6um Cell Library (cont.)

- **Use the following command to run linear model simulation.**

  vlog_mercury design.v testfixture.v
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60hp231d/cells/support/udps.vmd
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60hp231d.vmd
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60io420d.vmd

- **Use the following command to run ISM model simulation.**

  vlog_mercury design.v testfixture.v
  +ism
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60hp231d/cells/support/udps.vmd
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60hp231d.ismvmd
  -v <your_path>/CIC_CBKIT_V4/Verilog/cb60io420d.ismvmd

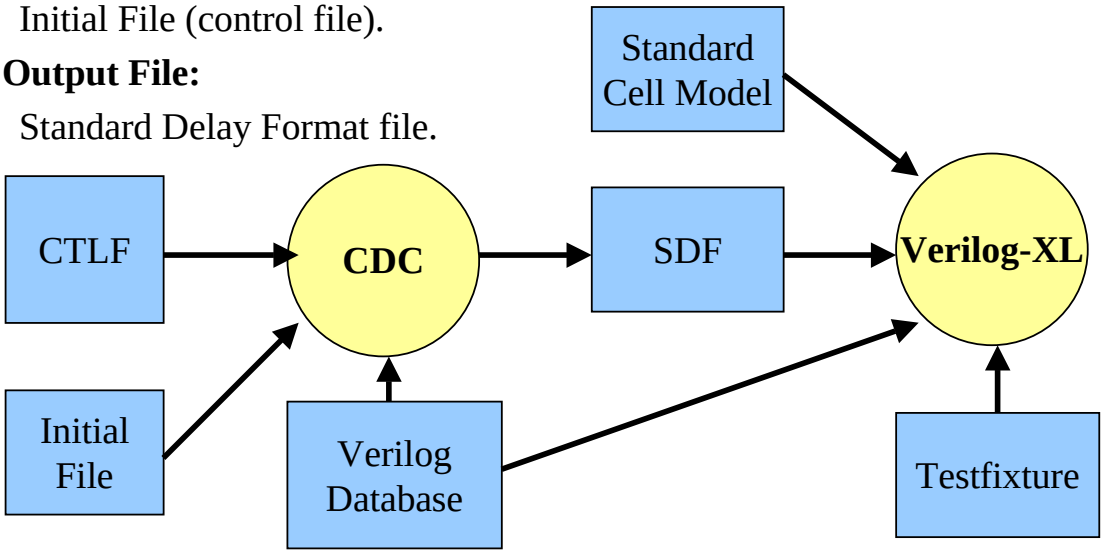
Simulation with Central Delay Calculator

- **Input File:**

  A compiled Timing Library Format library.
  Netlist connectivity.
  Initial File (control file).

- **Output File:**

  Standard Delay Format file.



Preparing the Input

- **Connectivity Data:**
  - Verilog file of your design.
- **Testfixture**
- **Cell Model:**
  - Verilog file of the standard cell library.
- **Timing Data:**
  - CTLF file of the standard cell library.
- **Initialization Data:**
  - A file that specifies the parameters for delay calculation.

Prepare the Testfixture

- **Add the following system task in your testfixture file to invoke CDC before simulation starts.**

  $cdc(module, initial_file);

- **Example testfixture file**

```
module testfixture;
  . . . . . .
  initial
    $cdc(U0, “dlc.init”);
  . . . . . .
endmodule
```

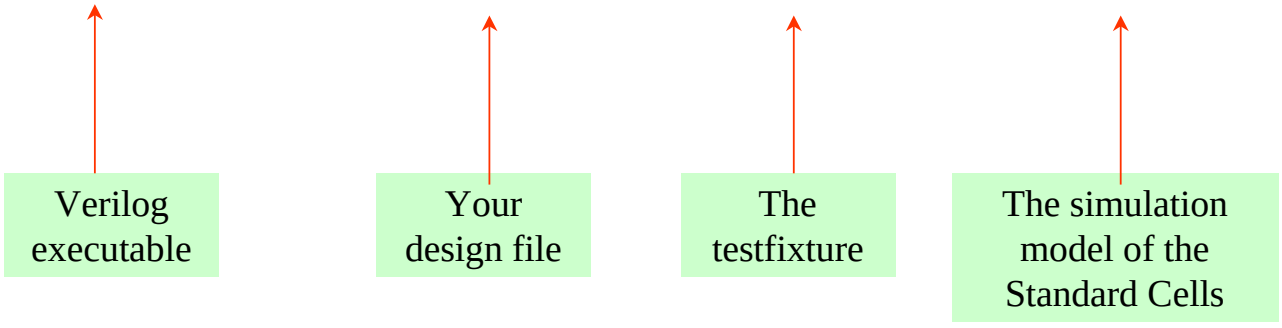
Initial file

- **A sample dlc.init is**

```
#This is the initializtation information of the CADENCE CDC.
CTLF_FILE "timing.ctlf"
SDF_TARGET "VERILOG"
SDF_VERSION 2.0
REPORT_INTRINSIC_IO
ESTIMATE_PARASITIC
HIER_CHAR "."
DEFAULT_OUTCAP (1.0 2.0 3.0)
PROC_VAR (0.92 1.0 1.33)
VOLTAGE (4.5 5.0 5.5)
TEMPERATURE (-50.0 25.0 150.0)
```


Running Simlation with CDC

```
vlog_cdc  +lic_ncv  my_design.v  my_testfixture.v  -v  cb60func.v
```



Summary

- **COMPASS on-line document.**
- **Using COMPASS cells.**
- **Linear and non-linear simulation.**
- **Using COMPASS delay calculator.**
- **Using Cadence Central Delay Calculator.**

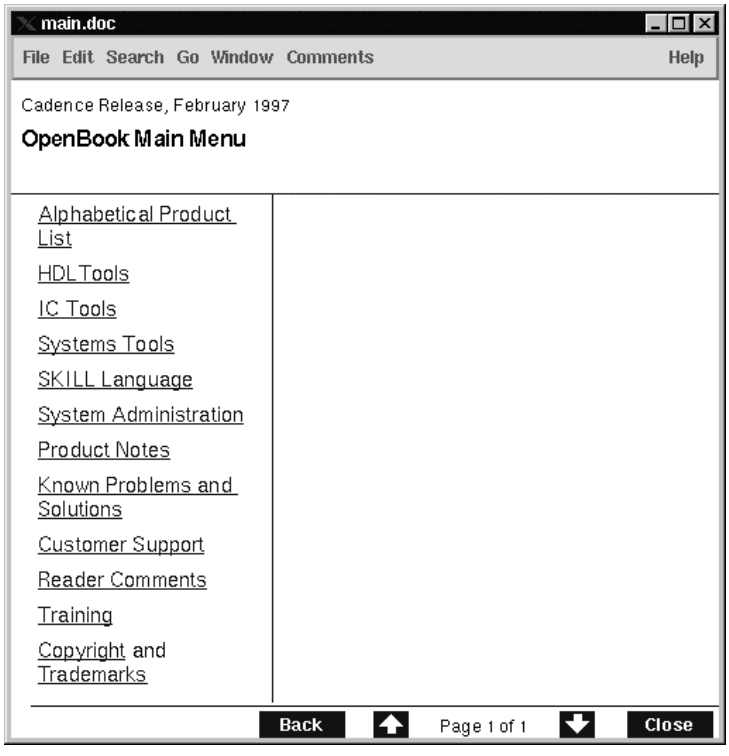
Appendix A

Useful Information

When You Have Problem

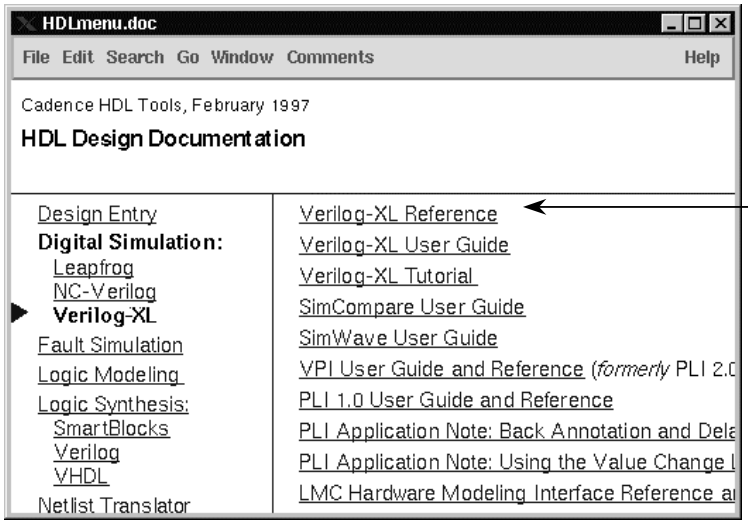
- **How to get help?**
  - use *openbook*

    unix % openbook



When You Have Problem (cont.)

- **In *openbook*, click** HDL tools → Verilog-XL



Verilog-XL reference

- **You can find Verilog-XL reference manual in here.**

When You Have Problem (cont.)

- **You can ask problems on network and CIC's staffs will answer it for you.**
  - News

    unix % setenv NNTPSERVER news.cis.nctu.edu.tw

    unix % tin

    then search for *cic*.
  - World Wide Web

    http://www.cic.edu.tw
  - E-Mail

    leejy@mbox.cic.edu.tw

Verilog HDL

Lab Book
Version 9807

Chip Implementation Center

蔡慶宏
(03) 577-3693 # 144
chtsai@mbox.cic.edu.tw

李杰原
(03) 577-3693 # 154
leejy@mbox.cic.edu.tw

Table of Content

Verilog Labs

Lab1 Set up Your Environment

- Lab 1-1. Set up Your Environment
- Lab 1-2. Using Openbook

Lab1-1. Set up Your Environment

Set up your UNIX search path.

1. Add the following lines into your *.cshrc* file.

   set path = (/usr/cadence/tools/bin $path)
   set path = (/usr/cadence/tools/dfII/bin $path)

2. Source your *.cshrc* file.

   unix % source ~/.cshrc <enter>

3. Test the result of environment setup .

   unix % which verilog
   /usr/cadence/tools/bin/verilog
   unix % verilog

Prepare for the Verilog lab data.

Before you going on the Verilog lab, you must first get the lab data from CIC's ftp site :

1. Connect to CIC's ftp site.

   unix% ftp ftp.cic.edu.tw
   Guest login ok, send your complete e-mail address as password.
   Password:
   230 Guest login ok, access restrictions apply.
   ftp>

2. Get the lab data.

```
ftp> bin
ftp> cd <Verilog_lab_data_path>
ftp> get vlog_lab_data_10.tar
ftp> bye
```

3. Extract the lab data.

```
unix% tar xvf vlog_lab_data_10.tar
```

After you have extracted the lab data, there should be a new directory *vlog_lab_1.0* appears. All labs are performs under this directory.

Summary

In this lab you have setup your Verilog-XL simulation environment by

1. Adding search path into your *.cshrc* file.
2. Getting Verilog lab data from CIC's ftp site.

Lab 1-2. Using the OpenBook

Objective :

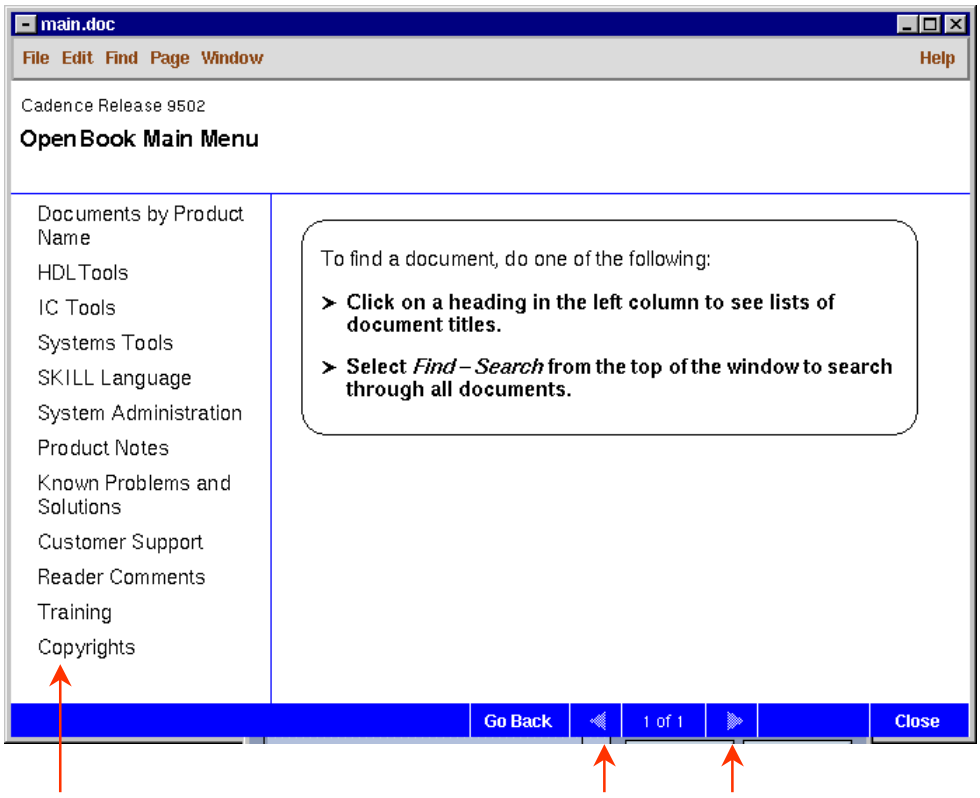
In this lab, you will learn how to use *OpenBook* online document to find out useful informations.

Starting the OpenBook

To start the *OpenBook*, you can enter openbook in a xterm window.

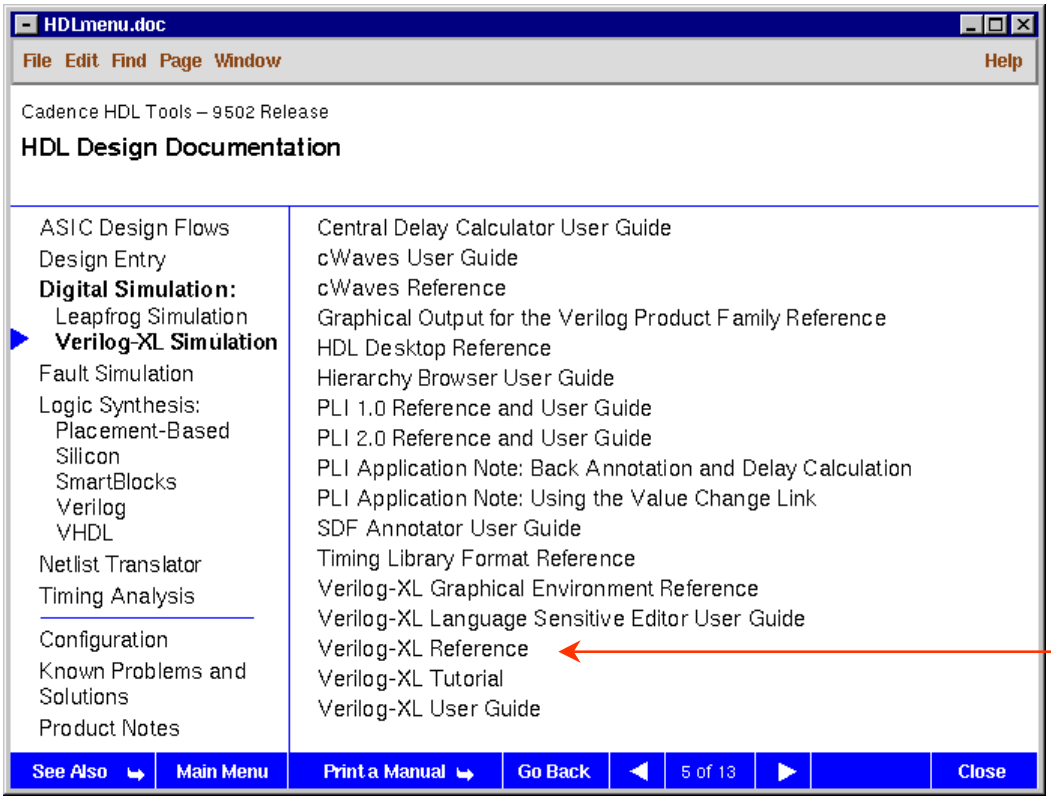
```
unix% openbook&
```

The *OpenBook* window will appears on your X window.



Finding Verilog related manuals from the topics page.

1. Use left most mouse botton to click **HDL Tools** from the *OpenBook* main menu, then select Verilog-XL simulation from the *OpenBook* topics field. You can see all Verilog related manuals appears on the right side of *OpenBook* window.
2. Use the left most mouse botton to select *Verilog-XL Reference* manual. You can see the *Verilog-XL Reference* manual opens in another window.



3. Use mounse botton to click **Introduction** in the *Verilog-XL reference* manual. The content of *Verilog-XL reference* manual will change to chapter 1.

Traverse the *OpenBook*.

1. Traverse *Verilog-XL* reference manual by clicking the *previous page arrow* and the *next page arrow*.
2. Use the left most mouse botton to click **Page** in the top tool bar of *OpenBook* window, Select **Go to Page...** , then type a number in the **Page Number** field. Now, which page do you see?
3. Click the **Go Back** botton in the bottom tool bar of *Verilog-XL Reference* window. Now, which page do you see?

Tips to find the wanted solutions.

1. In *OpenBook* window, click **Find** → **Search**, a pop up window *FrameViewer - Search* appears. Type "SDF" in the **Query** field and press **Search**. Do you see any difference in the *FrameViewer - Search* window?
2. In the **Query Result** field, you can see a lot of manuals which contain the keyword *SDF*. The goal you want is to limite the keyword searching in a specific area.
3. Click **Limit Search...** botton in the *FrameViewer - Search* window. A new pop up window will appear, which include two field. Move all manual catagories from left field to right field by selecting them and clicking "Disconnect" botton, except the **HDL Tools** catagory. Now, search the keyword *SDF* again. Do you see that the number of manuals appeared in the Query Result field is fewer than before?

Summary

In this lab you have learned the following topics of *OpenBook* :

1. How to traverse the *OpenBook.*
2. How to find Verilog-XL related manuals.
3. Find out the solutions using keyword search.

Lab 2. Quick Start

- Lab 2-1. Verilog-XL Simulation
- Lab 2-2. Using Cadence Wave

Lab 2-1. Verilog-XL Simulation

Objectives

In this lab you will learn how to use Verilog-XL simulator to simulate a Verilog design.

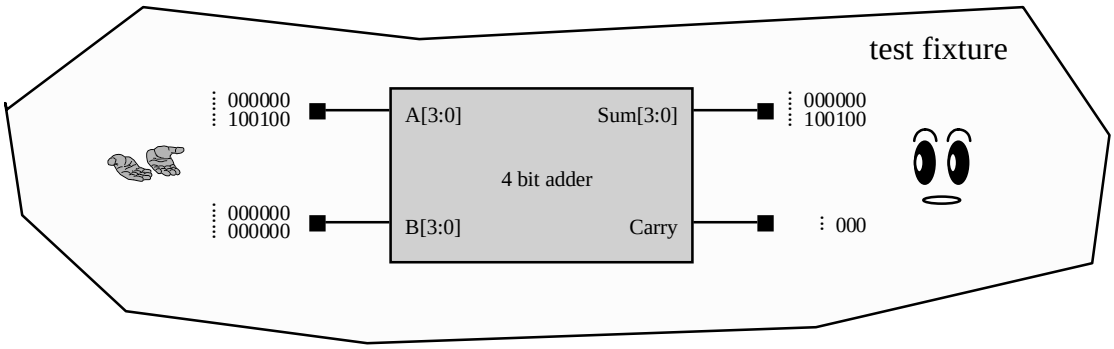
Simulation Circuit in this Lab.

In this lab you will simulate a 4-bit adder. The following is the truth table and simulation architecture of 4-bit adder.

truth table

adder pin / pattern	A		B		Sum		carry
	binary	decimal	binary	decimal	binary	decimal	binary
test pattern 1	0000	0	0000	0	0000	0	0
test pattern 2	0001	1	0000	0	0001	1	0
test pattern 3	0010	2	0000	0	0010	2	0
……	……	……	……	……	……	……	……

block diagram



Simulate a Simple Design

1. Change directory to lab2_sim. There are two files in that directory:

   add4.v ---- Verilog design file of a 4-bit adder.

   testfixture.v ---- test fixture file which is used to simulate the 4-bit adder

2. Simulate the 4-bit adder

   unix% verilog +lic_ncv add4.v testfixture.v

   You will see the following simulation result displayed on your X terminal:

```
Host command: /usr/cadence/tools/verilog/bin/verilog
Command arguments:
   add4.v
   testfixture.v

VERILOG-XL 2.2.1 log file created Jun 16, 1997  11:38:14
VERILOG-XL 2.2.1   Jun 16, 1997  11:38:14
......
......
A = 1111   B = 1110  Result = 11101
A = 1111   B = 1111  Result = 11110
*******************************
  End of simulation with OK result!
*******************************
3611 simulation events + 4737 accelerated events
CPU time: 0.1 secs to compile + 0.1 secs to link + 0.1 secs in
simulation
End of VERILOG-XL 2.2.1   Jun 6, 1997  11:44:32
```

   Observe the simulation result and make sure that the simulation result is correct.

Use a Batch File to Simplify Simulation Procedure.

1. Use text editor or vi to edit a new text file *run.f*, which contains the following lines:

   add4.v

   testfixture.v

   Re-simulate the 4-bit adder design by executing the following commands:

   unix% verilog +lic_ncv -f run.f

   Does the Verilog-XL simulator simulate the same files as the previous one?

Summary

In this lab, you have learned

1. How to simulate a Verilog design.
2. How to use a batch file to simplify simulation procedure.

Lab 2-2. Using SimWave

Objective

In this lab, you will learn how to display your simulation result in graphical environment.

Save Your Simulation Result

Before you viewing your graphical simulation result, you must save them in a waveform database.

1. Use text editor or vi to add the following lines right above the *endmodule* line in *testfixture.v*

```
module testfixture;
.....
initial begin
  $shm_open("waves.shm");
  $shm_probe("AS");
  end
endmodule
```

2. Re-simulate the 4-bit adder.

3. List the files in current UNIX directory and see what files or directory that Verilog-XL newly generated.

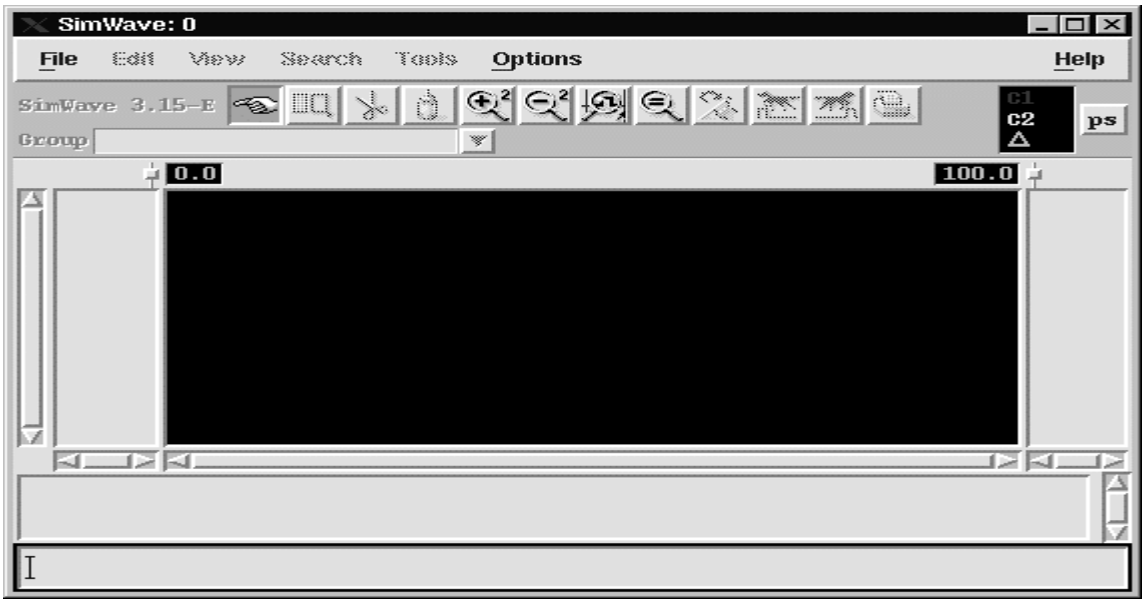
Using SimWave

You have saved your simulation result in a waveform database. Now you will invoke *SimWave* and open the pre-saved waveform database.

1. Starting *SimWave* by executing the commands

   unix% simwave&

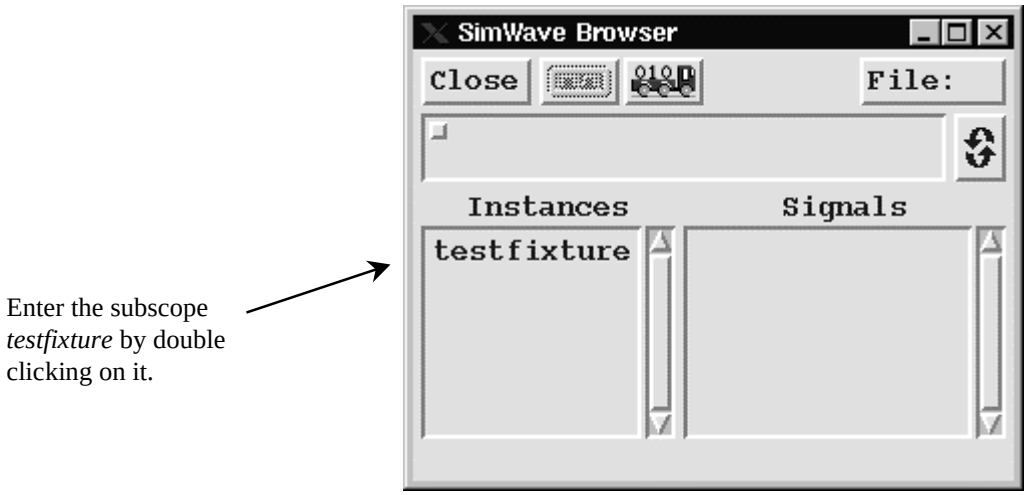
   The *SimWave* window will appears.



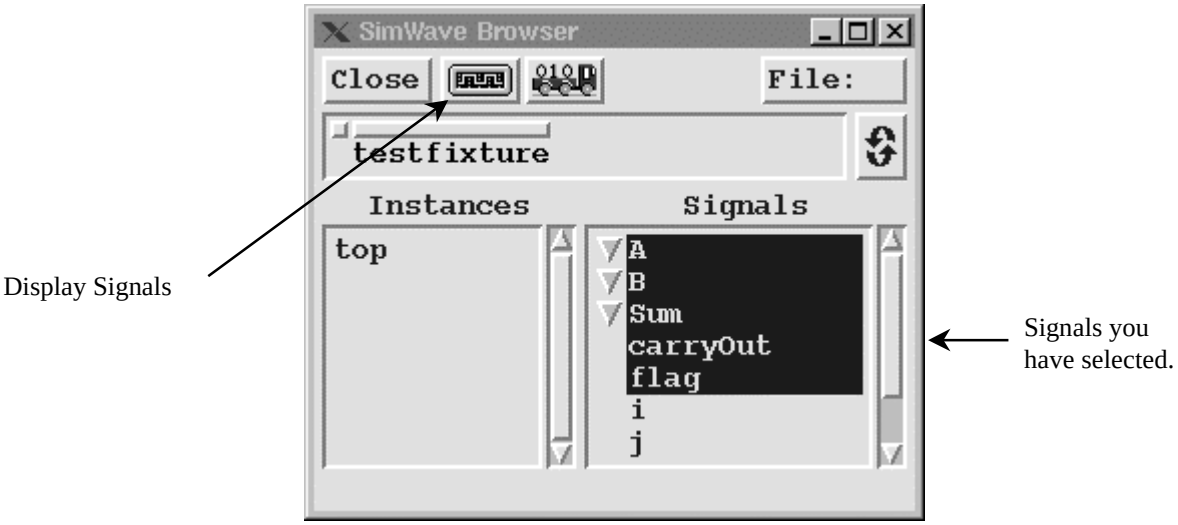
2. In *SimWave* window, click **File --> Database --> Load ...**, a pop up window appears. Select *waves.shm* in the **Directories** field of the pop up window, then press **OK**.

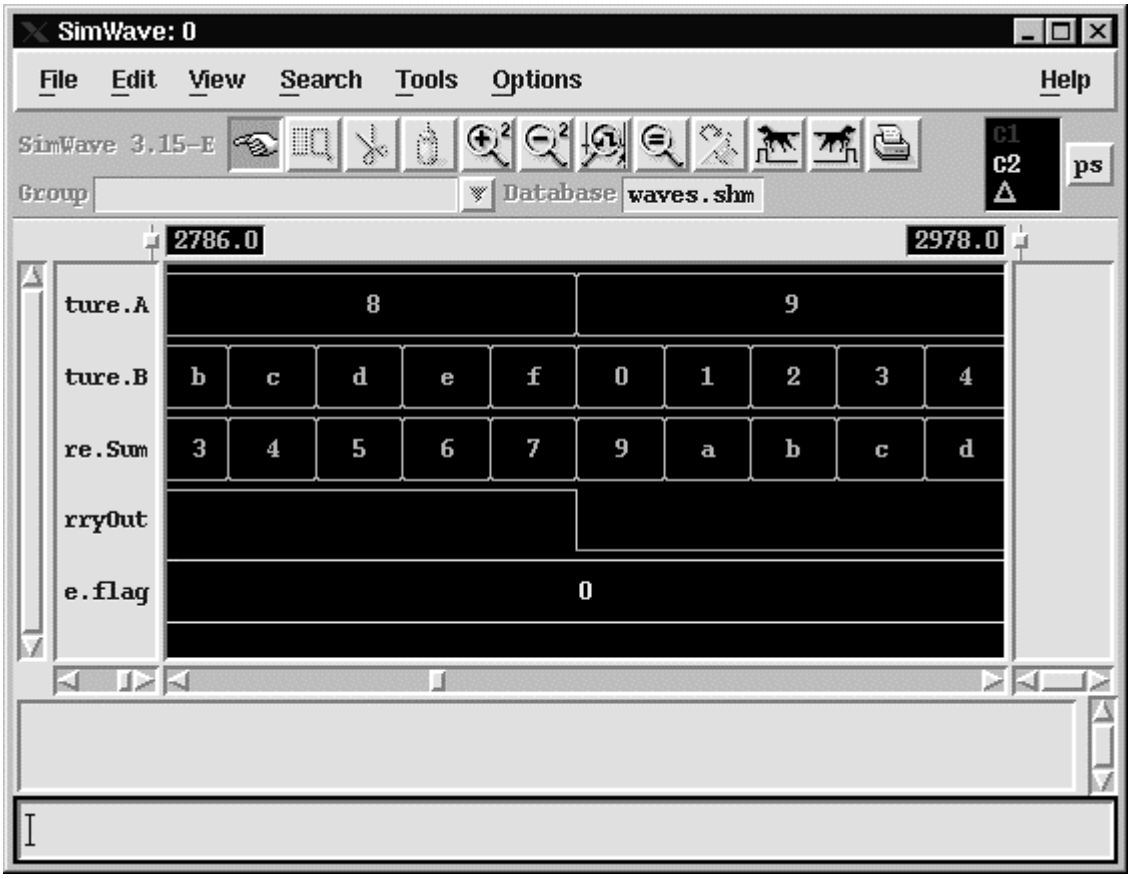
In next steps, you will select those signals you want to watch and display them in the *SimWave* window.

3. In *SimWave* window, click **Edit —> Add Signals...**, the SimWave Browser window opens, which can help you to select the signals you want to monitor. Select the scope *testfixture* by double clicking the *testfixture* in the *Instances* field.



4. Select signals you want to monitor and click the Display Signals button. After you click the button, the waveform will be displayed on *SimWave* window.





Now you have used *SimWave* to display your simulation result. You can try to find out the following *SimWave* features by your self:

- change the radix of displayed signal values.
- change the signal order by using mouse to drag it.
- modify your watching domain by zoom in and zoom out.
- group a set of signal waveforms.

Summary

In this lab, you have learned how to use *SimWave* to show your simulation result.

Lab3. Structural Modeling

- Lab 3-1. Full Adder Example.
- Lab 3-2. 4-bit Adder Example.
- Lab 3-3. Full Adder Example with Cell Library Component.

Lab 3-1. Full Adder Example

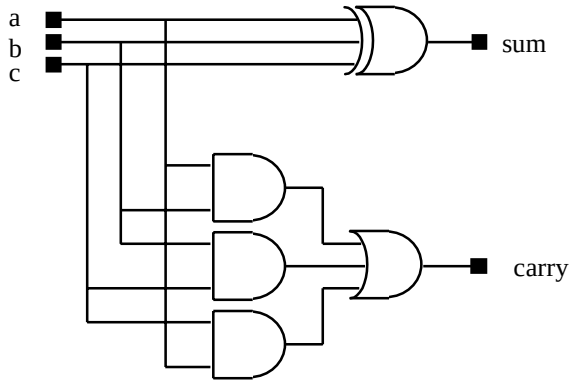
Objective

In this lab, you will learn how to use Verilog primitives to create a full adder's schematic netlist.

Full Adder Schematic

The following graphic is the schematic of a full adder. Try to build a Verilog netlist file which represents the full adder schematic. For convenient, the module name of this full adder is restricted to *f_adder*.

Note : Before you build your first Verilog file, do not forget to change your working directory to *lab3-structural*.



Hint : Build up your Verilog netlist as the following template.

```
module f_adder(sum, carry, a, b, c);
  output  sum, carry;
  input  a, b, c;

        schematic netlist
          description

endmodule
```

Simulate Your First Verilog Design

After you have finished your full adder design, you must simulate it to make sure that its interconnection is correct.

1. Save your Verilog design as *f_adder.v*.
2. Simulate your full adder with pre-exist testfixture *testfixture_fa.v*:

   unix% verilog +lic_ncv f_adder.v testfixture_fa.v
3. The simulation result will be displayed on your X terminal. If your full adder design passes the simulation, *Verilog-XL* will display the following messages:

   ```
   ************************************
      The full adder passes the simulation.
   ************************************
   ```

   On the contrary, if your full adder design does not meet its specification, the *Verilog-XL* will display the following messages:

   ```
   ****************************************
      The full adder does not pass the simulation.
   ****************************************
   ```

   Watche the simulation procedure and make sure that your full adder design indeed match its specification.

Summary

In this lab, you learn how to use Verilog primitive to build up a full adder design.

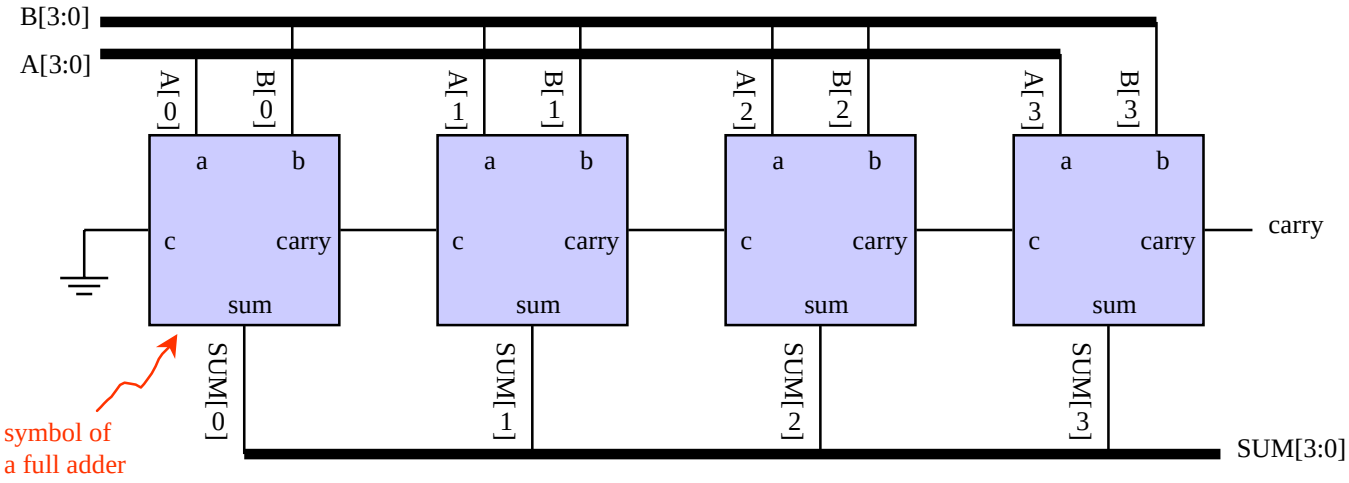
Lab 3-2. 4-bit Adder Example

Objective

In this lab, you will learn how to use the full adder created in the previous section to build a 4-bit adder.

4-bit Adder Schematic

The following graphic is the schematic of a 4-bit adder. Try to build a Verilog netlist file which represents the 4-bit adder schematic. For convenient, the module name of this full adder is restricted to *add4*.



Hint : Build up your Verilog netlist as the following template.

```
module add4 (SUM, carry, A, B);
  output  [3:0] SUM;
  output  carry;
  input   [3:0] A, B;

        schematic netlist
          description

endmodule
```

Simulate Your First Verilog Design

After you have finished your 4-bit adder design, you must simulate it to make sure that its interconnection is correct.

1. Save your Verilog design as *add4.v*.
2. Simulate your full adder with pre-exist testfixture *testfixture_add4.v*:

   unix% verilog +lic_ncv f_adder.v add4.v testfixture_add4.v
3. The simulation result will be displayed on your X terminal. If your 4-bit adder design passes the simulation, *Verilog-XL* will display the following messages:

   ```
   ************************************
      The 4-bit adder passes the simulation.
   ************************************
   ```

   On the contrary, if your 4-bit adder design does not meet its specification, the *Verilog-XL* will display the following messages:

   ```
   ****************************************
      The 4-bit adder does not pass the simulation.
   ****************************************
   ```

   Watche the simulation procedure and make sure that your 4-bit adder design indeed match its specification.

Summary

In this lab, you learn how to use pre-defined verilog module *f_adder* to build up a 4-bit adder design.

Lab 3-3. Full Adder Example with Cell Library Component.

Objective

In lab 3-1, you have used the Verilog primitives to build up your design. But, in most cases, you use the vender provided cell library to create your design. In this lab, you will learn how to use a cell library to build up a full adder design.

Re-Build The Full Adder design with Library Cells

1. Copy the full adder design created in Lab 3-1 to a new file.

   unix% cp f_adder.v f_adder_cb.v

2. Using text editor or vi to edit the file *f_adder_cb.v*. Replace the Verilog primitives with the following rules:

   - replace 3-input *xor* gate with *XO3A*.
   - replace 2-input *and* gate with *AN2A*.
   - replace 3-input *or* gate with *OR3A*.
   - add each instance an instance names if need.

   For example, replace the following Verilog primitive

   xor (sum, a, b, c) ;

   with the following one :

   XO3A U0(sum, a, b, c) ;

3. Re-simulate the full adder design with the command

   unix% verilog +lic_ncv f_adder_cb.v add4.v testfixture_add4.v

   Is there any error messages? If it does, what does these error messages mean?

Simulate The 4-bit Adder Design with Library Cells

The reason why Verilog-XL cannot simulate the new design is that it cannot recognize those cells *XO3A*, *AN2A*, and *OR3A*. Therefore, you must link the cell informations of these cells to Verilog-XL.

1. Re-simulate the 4-bit adder design with the command:

   unix% verilog +lic_ncv -v cells.v f_adder_cb.v add4.v testfixture_add4.v

   Can you simulate your 4-bit adder design this time?
2. Use text editor or vi to view the content of *cells.v* and see what this file contains.

Summary

In this lab you have learned

- how to use a vender provided cell library to define your circuit.
- how to simulate a circuit with library cells.

Lab 4. Behavorial Modeling

- Lab 4-1. Combinational Circuit Example.
- Lab 4-2. Sequential Circuit Example.

Lab 4-1. Combinational Circuit Example

Objective

In this lab you will learn how to utilize Verilog behavioral modeling technique to model a combinational circuit.

Modeling a 8-bit Adder

1. Change directory to lab4-adder.
2. Use text editor or vi to open a new file named *add8.v* and write your Verilog description in this file.
3. Because the property of combinational circuit is : whenever any one circuit input changes logic value, the circuit output will be updated, so you can utilize a Verilog *always* block to describe the behavior of a 8-bit adder.

Hint : Build up your Verilog netlist as the following template.

```
module add8 (SUM, carry, A, B);
  output  [7:0] SUM;
  reg     [7:0] SUM;
  output  carry;
  reg     carry;
  input   [7:0] A, B;

  always @(A or B)
    arithmatic equation
    of SUM and carry

endmodule
```

You can use Verilog arithmatic addition operator + to perform the 8-bit adder's function. For more information about arithmatic operator, you can refer to *Chapter 7 : Assignment*.

Simulate Your Verilog Design

Simulate your 8-bit adder again and make sure that it meets the adder specification.

1. Save your Verilog design as *add8.v*.
2. Simulate your full adder with pre-exist testfixture *testfixture_add8.v*:

   unix% verilog +lic_ncv add8.v testfixture_add8.v

   The testfixture file *testfixture_add8.v* already exists in the directory *lab4-adder*. So you can observe the simulation result and see whether or not your behavioral description of 8-bit adder is correct.

Summary

In this lab, you have

- created a 8-bit adder design.
- learned how to use a *always* block to model a combinational circuit.
- learned how to use Verilog *arithmatic* operator to model your circuit.

Lab 4-2. Sequential Circuit Example

Objective

In this lab you will learn how to model a sequential circuit by using Verilog behavioral modeling mechanism.

Example Circuit Used in This Lab

The example circuit used in this lab is a 4-bit shifter. Figure 4-1 shows the symbol of the 4-bit shifter.

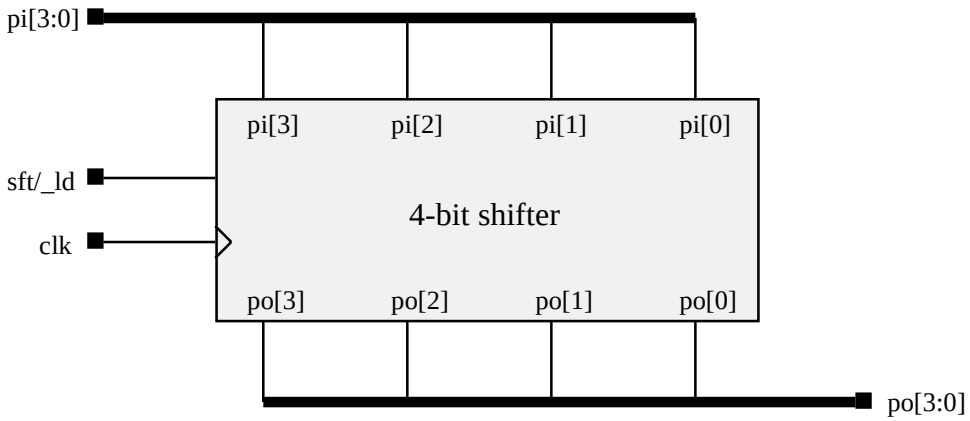


Fig 4-1. Symbol of a 4-bit shifter

The behavior of the 4-bit shifter is listed in the following:

- When *sft/_ld* = 0, the 4-bit shifter loads data into its internal storage element from parallel input *pi[3:0]* on the rising edge of *clk*.
- When *sft/_ld* = 1, the 4-bit shifter shifts left its internal data on the rising edge of *clk*, with one bit per clock cycle.

Modeling the 4-bit shifter

Perform this lab in the *lab4-shifter* directory. This directory contains a testfixture file that can be used to test a 4-bit shifter design.

1. Change your working directory to *lab4-shifter*.
2. Use text editor or vi to open a new file named *shifter4.v* and write your Verilog description in this file.

To model the 4-bit shifter design, you must note that all circuit's behavior (either load or shift) must be synchronized with clock signal *clk*. The following is a template file which you can use to build your 4-bit shifter design:

```
module shifter4 (PO, PI, sft_ld, clk);
  output [3:0] PO;
  reg    [3:0] PO;
  input  [3:0] PI;
  input  sft_ld, clk;

always @(posedge clk)

  behavorial description of
  shift operation and
  load operation

endmodule
```

The shift operation and load operation are synchronized with clock signal *clk*.

Hint : you can use Verilog shift operator (<< and >>) to perform the 4-bit shifter's shift function. For more information about the shift operator, please refer to *Chapter 7 : Assignment*.

Think : In lab 4-1 and lab 4-2, we all utilize the Verilog *always* block to model a combinational circuit and a sequential circuit. *What is the difference in modeling techniques?*

Simulate the 4-bit shifter

Now, you are ready to simulate your 4-bit shifter design.

1. Save your design as *shifter4.v*.
2. Simulate your 4-bit adder design.

   unix% verilog +lic_ncv shifter4.v testfixture_shifter4.v
3. Observe your simulation result. If your 4-bit shifter design meets its specification, Verilog-XL will display the following message:

   ```
   ********************************
     Simulation of 4-bit shifter is OK!
   ********************************
   ```

   If your 4-bit shifter does not meet its specification, Verilog-XL will display the following message:

   ```
   ***********************************
     Simulation of 4-bit shifter is not OK!
   ***********************************
   ```

Summary

In this lab you have

- created a 4-bit shifter design.
- learned how to use *always* block to synchronize a set of circuit behavior.
- learned how to use Verilog *shift* operator to model your circuit.

Lab 5. Interactive Debugging

◆Lab 5-1. Debug an ALU Design

Lab 5-1. Debug an ALU Design

Objective

Verilog-XL provides a text mode debugging environment. Although this debugging environment is not very user friendly, but it is helpful when your design has problem. In this lab, you will learn how to use the interactive debugging environment.

Example Circuit Used in This Lab

The circuit used in this lab is a 4-bit ALU. Figure 5-1 is the symbol of the 4-bit ALU. The function of this ALU is also listed in Table 5-1.

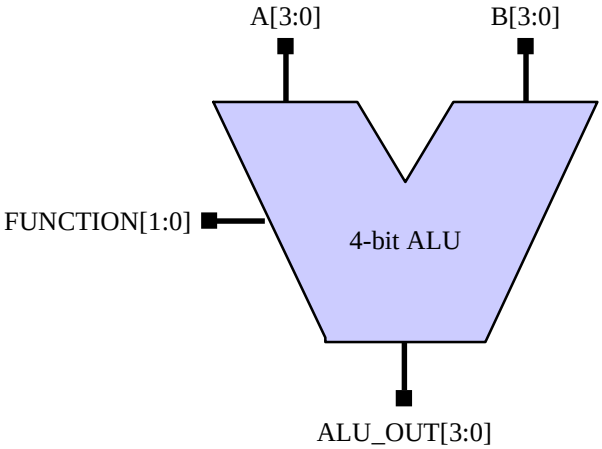


Figure 5-1. Symbol of a 4-bit ALU

FUNCTION[1:0]	ALU operation	ALU_OUT[3:0]
11	arithmatic addition	A + B
10	logical AND	A and B
01	logical OR	A or B
00	logical XOR	A xor B

Table 5-1. Various ALU Operation Mode

Figure 5-2 shows the schematic of 4-bit ALU. In this lab, you will use this schematic to debug the 4-bit ALU design.

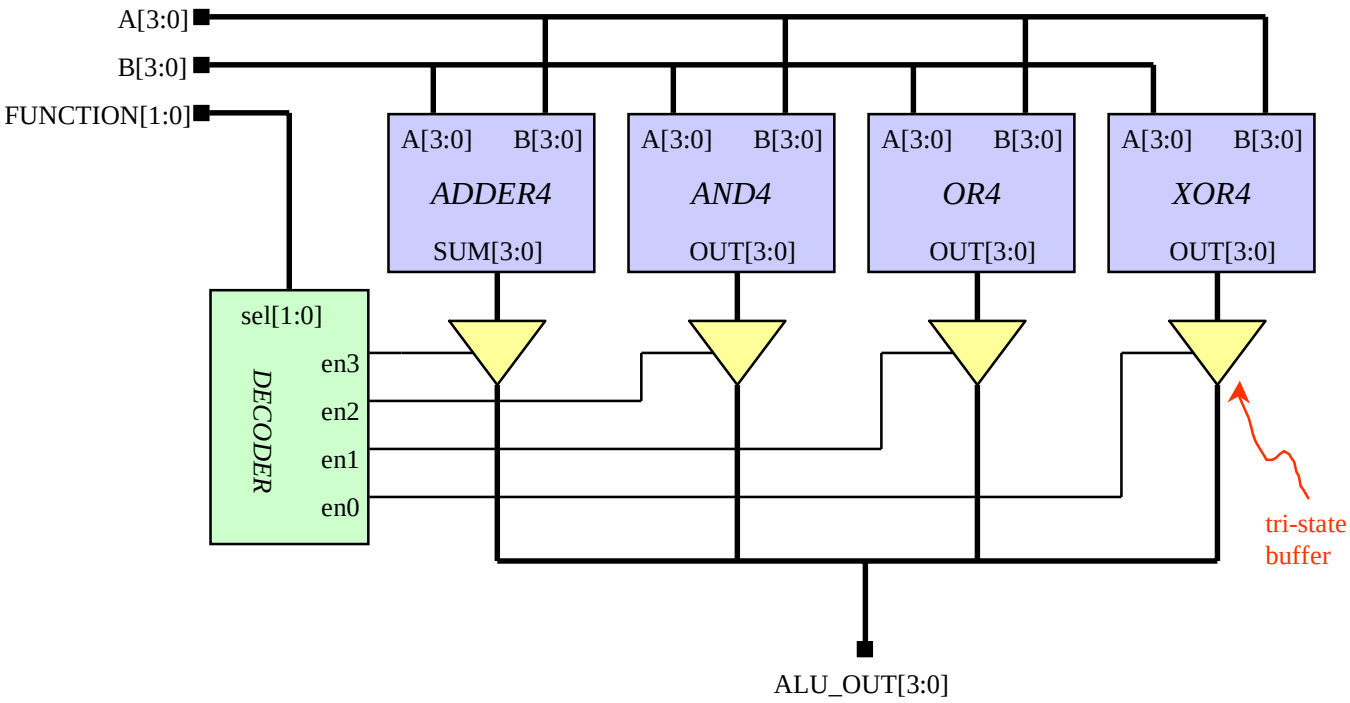


Figure 5-2. Schematic Netlist of 4-bit ALU

First Simulation

In this lab, you will first simulate a bug-embedded ALU design. Then you must clear those bugs by debug the ALU design in Verilog-XL interactive environment.

1. Change your working directory to *lab5-interactive.* In this directory you can see the following files :

alu.v	—>	the Verilog file of 4-bit ALU.
testfixture_alu.v	—>	the testfixture file of 4-bit ALU.
global_define.v	—>	this file contains the global definition of 4-bit ALU design.

2. Simulate the 4-bit ALU design.

   unix% verilog +lic_ncv alu.v testfixture_alu.v

   This testfixture will apply exhaustive test patterns to the 4-bit ALU (that is, all four ALU operation mode will be tested, with exhaustive *A* and *B* input).

   After you simulate the 4-bit ALU, you will see a lot of error messages appear on your X terminal.

   ```
   < Verilog-XL compiling messages >
   function ADD fail !!!  1 +  1 =  X
   function ADD fail !!!  1 +  3 =  X
   function ADD fail !!!  1 +  5 =  X
   . . . . . .
   ```

   It means that this ALU design cannot generat correct output in various operation mode. So you must try to find out the ALU bugs in this lab.

3. Open the SimWave window and see the simulation waveform.

   Unix% simwave &

   Click “File” -> “Database” -> “Load” to load the waves.shm database.

   After you have viewed the simulation waveform, please close the waveform window.

4. Enter Verilog-XL interactive mode by using the *-s* command line option.

```
unix% verilog +lic_ncv -s alu.v testfixture_alu.v
```

Verilog-XL will first compile the *alu.v* and *testfixture_alu.v*, then enter *interactive mode* immediately. You can see the following command prompt :

```
Type ? for help
C1 >
```

5. Type the interactive command *$showscopes* to see the current scope and subscopes.

```
C1 > $showscopes;
Directory of scopes at current scope level:
   module (ALU), instance (top)

Current scope is (testfixture_alu)
Highest level modules:
testfixture_alu                    <-- subscopes
C2 >                               <-- current scope
```

6. Type the interactive command *$showvars* to see the names and values of current scope variables.

```
C2 > $showvars;                    <-- the top most scope
Variables in the current scope:
ALU_OUT[3] (testfixture_alu) wire = StX
    . . . . . .
```

Debug the 4-bit ALU Design

1. Change scope from testfixture_alu to ALU.

```
C3 > $scope( top );
C4 > $showscopes;
Directory of scopes at current scope level:
   module (ADDER4), instance (alu_adder)
   module (AND4), instance (alu_and)
   module (OR4), instance (alu_or)
   module (XOR4), instance (alu_xor)
   module (DECODER), instance (alu_decoder)

Current scope is (testfixture_alu.top)
Highest level modules:
testfixture_alu
C5 >
```

Now you can see 5 subscopes exist in the scope *testfixture_alu.top.* You can refer to Figure 5-3, which can help you to realize the design hierarchy.

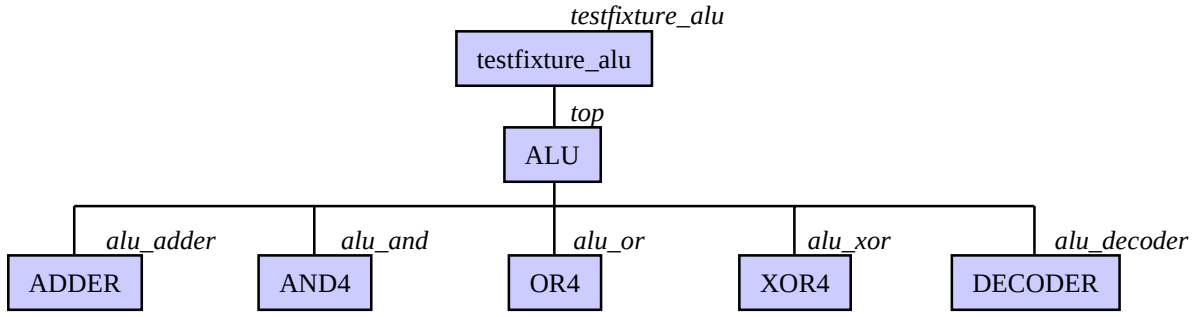


Figure 5-3. Design Hierarchical

2. The first simulation error occurs at time 170. For debugging purpose, we must stop the simulation at time 170.

```
C5 > $db_breakbeforetime(170);
Set break (1) before time 170.
C6 > .
Break (1) occured before time 170.
C6 >
```

Continue simulation process.

Simulation process stops at time 170.

3. Simulate the 4-bit ALU step by step until we see the first simulation error. To advance the simulation one step, you can type a comma :

```
C6 > ,
SIMULATION TIME IS 170
L27 "testfixture_alu.v": #10 >>> CONTINUE
C6 >
```

For convenience, if you want to progress many simulate steps at the same time, you can type many commas in one command line. For example, if you type 10 commas in one command line :

```
C6 > , , , , , , , , , , , , , , ,
L27 "testfixture_alu.v": if(ALU_OUT !== (i + j)) >>> SKIPPING
L34 "testfixture_alu.v": end
L23 "testfixture_alu.v": for(; j <= 15; j = j + 1) >>> j = 32'h1, 1
. . . . . .
```

Continue step by step simulation until you have seen the following two lines :

```
L2 "testfixture_alu.v": wire ALU_OUT[1] >>> XL NET = StX
L2 "testfixture_alu.v": wire ALU_OUT[0] >>> XL NET = StX
```

Up to now, you have traced to the simulation point where first error occurs.

The first simulation error occurs when the *A* and *B* input of ALU equal to *4'b0001* and *4'b0001*, respectively. The ALU operation mode is *arithmatic addition*. So the desired ALU output should be *4'b0010*, while the simulated ALU output is *4'b00XX*.

4. To find out what cause the unknow value occurs in the last two least significant bits, you can use *$showvars* to help you.

```
C6 > $showvars( ALU_OUT[1] );
ALU_OUT[1] (testfixture_alu.top) wire = StX
   HiZ <– (testfixture_alu.top): bufif1 xor_buf1(ALU_OUT[1], XOR_OUT[1], en0);
   St0  <– (testfixture_alu.top): bufif0 or_buf1  (ALU_OUT[1], OR_OUT[1], en1);
   HiZ <– (testfixture_alu.top): bufif1 and_buf1(ALU_OUT[1], AND_OUT[1], en2);
   St1  <– (testfixture_alu.top): bufif1 add_buf1(ALU_OUT[1], ADDER_OUT[1], en3);
C7 >
```

These two gates drive different values to one net and cause an unknow value on that net.

At this time, two logic gates are driving different logic value to the net *ALU_OUT[1]* : buffer *or_buf1* provides logic value 0 and buffer *add_buf1* provides logic value 1. The same situation happens on the net *ALU_OUT[0]*.

5. Use text editor or vi to open the 4-bit ALU design file *alu.v* in another X terminal, then move the cursor to line 66. The following is a part of *alu.v*, which describe the *tri-state* buffers in Figure 5-2.

```
. . . . . .
bufif1 and_buf1(ALU_OUT[1], AND_OUT[1], en2);
bufif1 and_buf0(ALU_OUT[0], AND_OUT[0], en2);

bufif0 or_buf3(ALU_OUT[3], OR_OUT[3], en1);
bufif0 or_buf2(ALU_OUT[2], OR_OUT[2], en1);
bufif0 or_buf1(ALU_OUT[1], OR_OUT[1], en1);
bufif0 or_buf0(ALU_OUT[0], OR_OUT[0], en1);
. . . . . .
```

We can see that there is a design error on the four buffers *or_buf3, or_buf2, or_buf1*, and *or_buf0*. The original buffer type is *bufif0*, but they should be *bufif1*. So modify line 66 to line 70 in *alu.v* as :

```
bufif1 or_buf3(ALU_OUT[3], OR_OUT[3], en1);
bufif1 or_buf2(ALU_OUT[2], OR_OUT[2], en1);
bufif1 or_buf1(ALU_OUT[1], OR_OUT[1], en1);
bufif1 or_buf0(ALU_OUT[0], OR_OUT[0], en1);
```

Save the modified 4-bit ALU design in another file *alu_fixed.v* and continue the Verilog simulation by typing a *period* in interactive prompt :

```
C7 > .
. . . . . .
End of VERILOG-XL 2.2.1   Jun 19, 1997  21:17:04
unix%
```

6. Re-simulate the fixed 4-bit ALU design.

```
unix%  verilog +lic_ncv alu_fixed.v testfixture_alu.v
```

If your modification on *alu.v* is correct, you should see the following message :

```
*****************************************
   Simulate ALU addition function OK!!!
*****************************************

*****************************************
   Simulate ALU logical AND function OK!!!
*****************************************

*****************************************
   Simulate ALU logical OR function OK!!!
*****************************************

*****************************************
   Simulate ALU logical XOR function OK!!!
*****************************************
```

Summary

In this lab, you have learned

- how to enter Verilog-XL interactive mode.
- how to traverse your design hierarchy in interactive mode.
- practice some useful Verilog-XL *interactive command* in interactive mode.
- how to debug design errors in a 4-bit ALU.

Lab 6. Modeling Memory

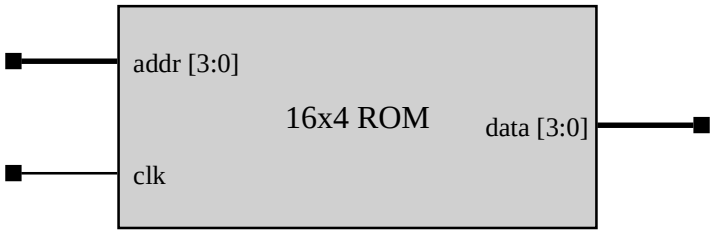
- Lab 6-1. Modeling a ROM.
- Lab 6-2. Modeling a RAM.

Lab 6-1. Modeling a ROM

Objective

In this lab you will learn how to model a ROM with pre-defined ROM code.

Symbol of a 16x4 synchronout ROM.



Model a 16x4 synchronous ROM

Change the working directory to *lab6-rom*. This directory contains one file : *testfixture_srom.v*. In this lab, you will model a 16x4 synchronous ROM.

1. Use text editor or vi to open a file *srom16x4.v* and model this 16x4 synchronous ROM in this file. The ROM code of each address is the complement of its address. Table 6-1 lists the ROM code of each address :

address in decimal	address in binary	ROM code of each address
0	4'b0000	4'b1111
1	4'b0001	4'b1110
. . . . .	. . . . .	. . . . .
15	4'b1111	4'b0000

Table 6-1. ROM code of 16x4 synchronous ROM

2. To model the timing of this example ROM, please refer to Figure 6-1 which shows the access time of this ROM. You must consider the access time when modeling the ROM.

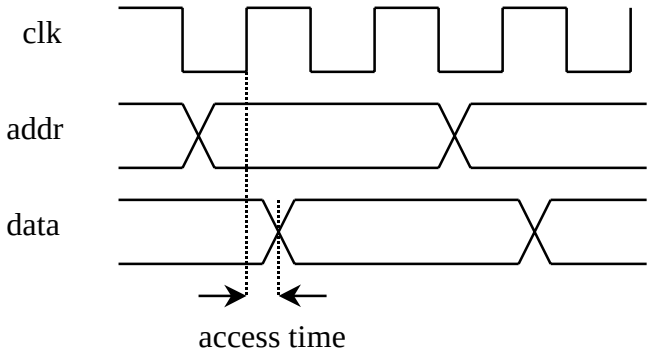


Figure 6-1. Timing diagram of 16x4 synchronous ROM

3. For the convenience of simulation, please use the following template to create your 16x4 synchronous ROM:

```
module srom16x4(data, addr, clk);
  output [3:0] data;
  input   [3:0] addr;
  input  clk;
  reg [3:0] memory [15:0];

  parameter  t_acc = 5;

     ROM code model

endmodule
```

Simulate the 4x16 Synchronous ROM

1. Execute the following simulation command.

```
unix%  verilog +lic_ncv srom16x4.v testfixture_srom.v
```

2. If you have correctly modeled the 16x4 synchronous ROM, you will see the following simulation result :

```
**********************************
   Completed ROM Tests With 0 Errors!
**********************************
```

If the simulation result is not correct, check the following points :

- ROM code is correctly specified.
- ROM output is completely synchronized with clock signal *clk*.

Summary

In this lab you have learned

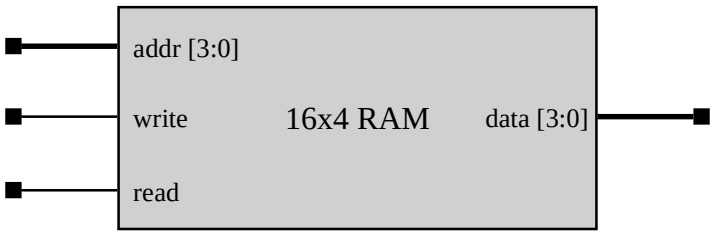
- how to model a synchronous ROM.
- how to model access time of a ROM.

Lab 6-2. Modeling a RAM

Objective

In this lab you will learn how to model a RAM.

Symbol of a 16x4 RAM.



Model a 16x4 RAM

Change the working directory to *lab6-ram*. The *testfixture_ram.v* file already exists in this directory. This file is used to test the RAM module you created.

1. Open a new file called *ram16x4.v* to model the 16x4 RAM.
2. Use the following template to create your 16x4 RAM :

```
module ram16x4(data, addr, write, read);
  inout  [3:0] data;
  input  [3:0] addr;
  input  write, read;
  reg [3:0] memory [15:0];

  parameter  t_acc = 5;

     RAM cell model

endmodule
```

3. The RAM's read/write control should be as the following graph :

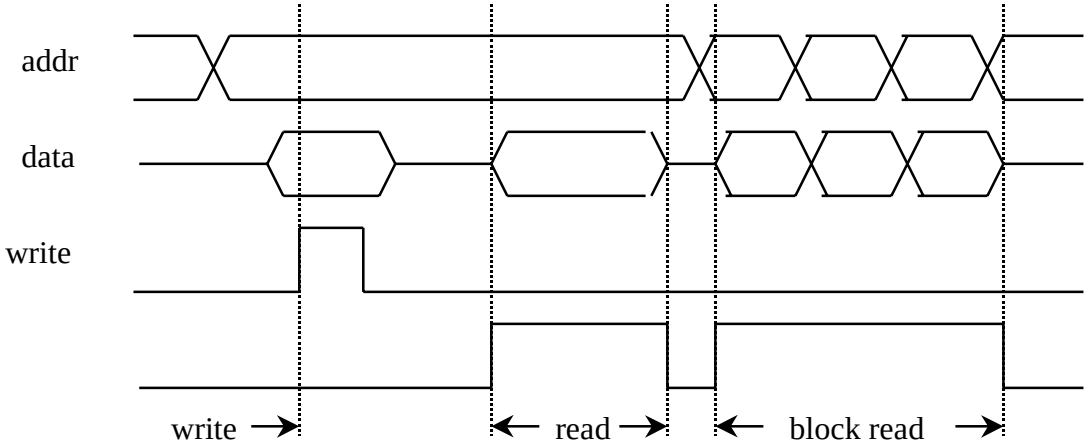


Figure 6-2. Timing diagram of 16x4 RAM

- write cycle : at the positive edge of *write* control signal, the value on the *data* bus should be written into the memory location addressed by the *addr* bus.
- read cycle : when the *read* line is asserted (logic 1), the contents of the memory location addressed by the *addr* bus should be continuously driven to the *data* bus. (Hint : Use a continuous assignment).
- block read : if the *addr* bus changes while *read* is asserted, the contents of the new memory location addressed should be immediately transfered to the *data* bus.
- disable : when the *read* control line is not asserted (logic 0), the memory should place the *data* bus in a high impedance state.

4. Simulate your RAM design by using the *testfixture_ram.v* file.

unix% verilog +lic_ncv ram16x4.v testfixture_ram.v

If you have correctly modeled the 16x4 RAM, you will see the following simulation result :

```
**********************************
  Completed RAM Tests With 0 Errors!
**********************************
```

Summary

In this lab you have

- created a 16-word by 4-bit random access memory.
- learn how to handle bidirectional ports.

LAB 7. Finite-State Machine

◆Lab 7-1. Model a Finite-State Machine

Lab 7-1. Model a Finite-State Machine

Objective

Finite-state machines are widely used as controllers in various logic system. In this lab, you will learn how to use Verilog HDL to model a simple finite-state machine.

An Example FSM

Figure 7-1 shows the symbol and state diagram of the example finite-state machine, which will be used as a controller in lab 8 :

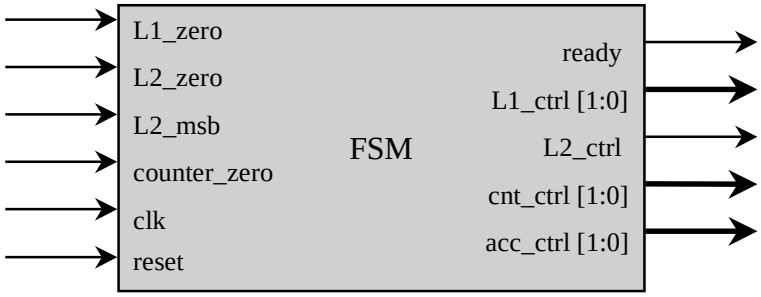


Figure 7-1(a). Symbol of example finite-state machine

This finite-state machine has four input signal *L1_zero*, *L2_zero*, *L2_msb* and *counter_zero*, which represent the status of other circuits (for example, *L1_zero* = 1 represents the content of L1 register is currently zero). The finite-state machine should generate control signals *ready*, *L1_ctrl*, *L2_ctrl*, *cnt_ctrl*, and *acc_ctrl* according to its current state and its input signals.

Figure 7-2 shows the state diagram of the example finite-state machine. The state transition and state encoding are listed in Table 7-1.

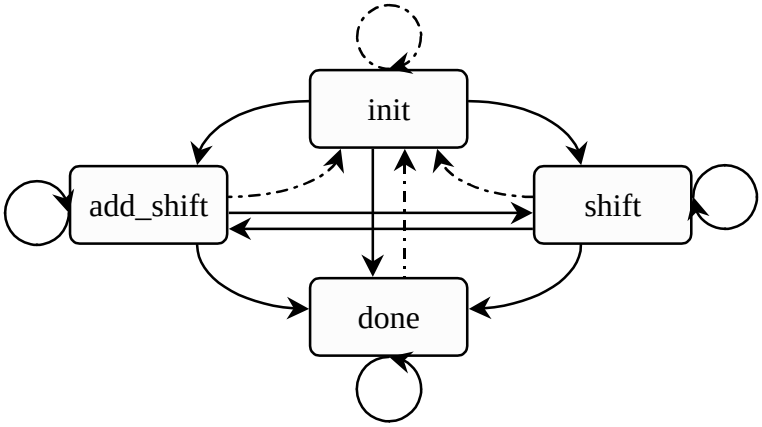


Figure 7-2. state diagram of example finite-state machine

state	reset	L1_zero	L2_zero	L2_msb	counter_zero	state+	ready	L1_ctrl	L2_ctrl	cnt_ctrl	acc_ctrl
all state	0	X	X	X	X	init	X	X	X	X	X
init	1	1	X	X	X	done	0	2'b11	1'b1	2'b00	2'b00
init	1	X	1	X	X	done	0	2'b11	1'b1	2'b00	2'b00
init	1	0	0	0	0	shift	0	2'b11	1'b1	2'b00	2'b00
init	1	0	0	1	0	add_shift	0	2'b11	1'b1	2'b00	2'b00
add_shift	1	X	X	0	0	shift	0	2'b00	1'b0	2;b11	2'b11
add_shift	1	X	X	1	0	add_shift	0	2'b00	1'b0	2;b11	2'b11
add_shift	1	X	X	X	1	done	0	2'b00	1'b0	2;b11	2'b11
shift	1	X	X	0	0	shift	0	2'b01	1'b0	2'b11	2'b11
shift	1	X	X	1	0	add_shift	0	2'b01	1'b0	2'b11	2'b11
shift	1	X	X	X	1	done	0	2'b01	1'b0	2'b11	2'b11
done	1	X	X	X	X	done	1	2'b01	1'b0	2'b10	2'b10

Table 7-1. State Transition Table of example finite-state machine

Model the Example Finite-State Machine

1. Change the working directory to *lab7-fsm*. The following files already exists in this directory :

   testfixture_fsm.v ---> test fixture file for the example finite-state machine.

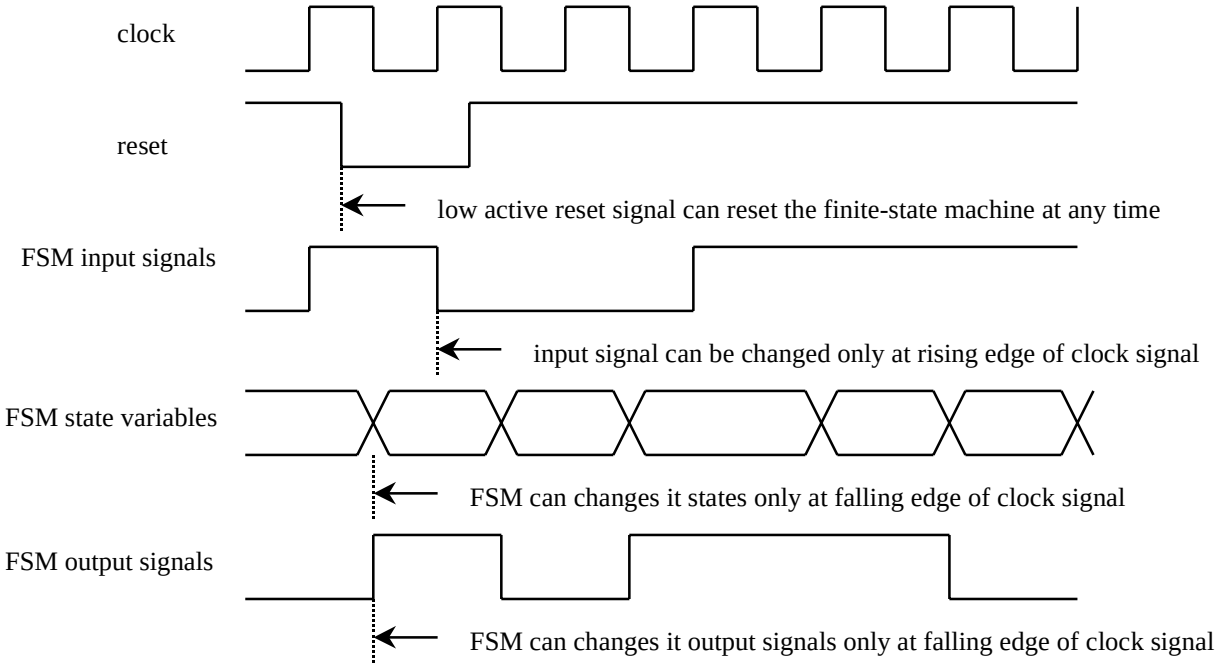
2. Open a new file called *fsm.v* to model the example finite-state machine. To make sure the consistency between finite-state machine model and testfixture file, use the following template to create the finite-state machine model :

```
module fsm(ready, L1_ctrl, L2_ctrl, cnt_ctrl, acc_ctrl, clk,
   L1_zero, L2_zero, L2_msb, counter_zero, reset);
   output ready;
   output [1:0] L1_ctrl;
   output L2_ctrl;
   output [1:0] cnt_ctrl, acc_ctrl;
   input  clk, L1_zero, L2_zero, L2_msb, counter_zero, reset;

   finite-state machine model

endmodule
```

3. For convenience, use the following state encoding to model the example finite-state machine :

State name	state encoding
init	00
add_shift	01
shift	10
done	11

4. Refer the following timing diagram to model the state transistion of the finite-state machine :



Hint. To model this finite-state machine, you can use three procedural blocks to model the following machanism.

1. Use a *always* procedural block to model state transition.

   always at negitive edge of clock,
   if *reset* equals to logic zero, then set current state to *init* state.
   else set current state to *next state* variable.

2. Use a *always* procedural block to evaluate *next state* variable.

   whenever any one of *current state, L1_zero, L2_zero, L2_msb,* and *counter_zero* net changes value, evaluate *next state* variable according to the current logic values of *current state, L1_zero, L2_zero, L2_msb,* and *counter_zero.*

3. Use a *always* procedural block to evaluate output signals *ready*, *L1_ctrl*, *L2_ctrl*, *cnt_ctrl*, and *acc_ctrl.*

   whenever the finite-state machine changes its *current state* variable, evaluate its output signals according to its *current state* variable.

Simulate the finite-state machine

1. Simulate the finite-state machine with pre-exist testfixture file *testfixture_fsm.v.*

   ```
   unix%  verilog +lic_ncv fsm.v testfixture_fsm.v
   ```

2. Open SimWave to verify your simulation result.

Summary

In this lab you have

- understood the mechanism of a finite-state machine.
- created the Verilog model of an example finite-state machine.
- simulated the finite-state machine.

Lab 8.

A Top Down Design Example

- Lab 8-1. Behavior Modeling of a Serial Multiplier
- Lab 8-2. RTL Modeling of a Serial Multiplier

Lab 8-1. Behavioral Modeling of a Serial Multiplier

Objective

Learn how to use Verilog high-level programming language to model the algorithm of a 4-bit serial multiplier.

Example Serial Multiplier

Figure 8-1 shows the symbol of a serial multiplier : a low-active reset signal *reset* is used to reset and start the multiplier; when *reset* goes to logic 0, the serial multiplier will be reseted; when *reset* goes from low to high, the serial multiplier will start to multiple its 4-bit multiplicand *A* and 4-bit multiplicator *B* at next rising edge of *clk*; after the multiplier has finished its operation, it will put the result on its output *MULT* and set *ready* signal to logic 1.

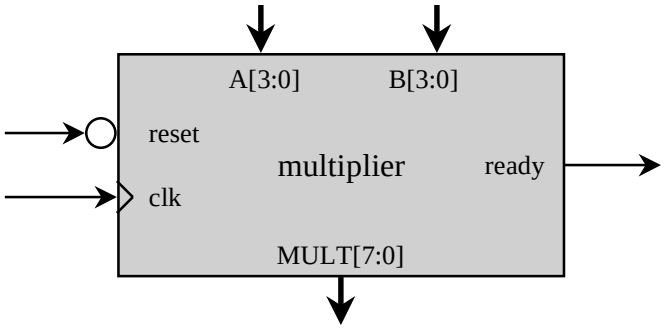


Figure 8-1. Symbol of a 4-bit serial multiplier

Figure 8-2 shows the algorithm of the 4-bit serial multiplier. Later you will use Verilog HDL to model and simulate this algorithm.

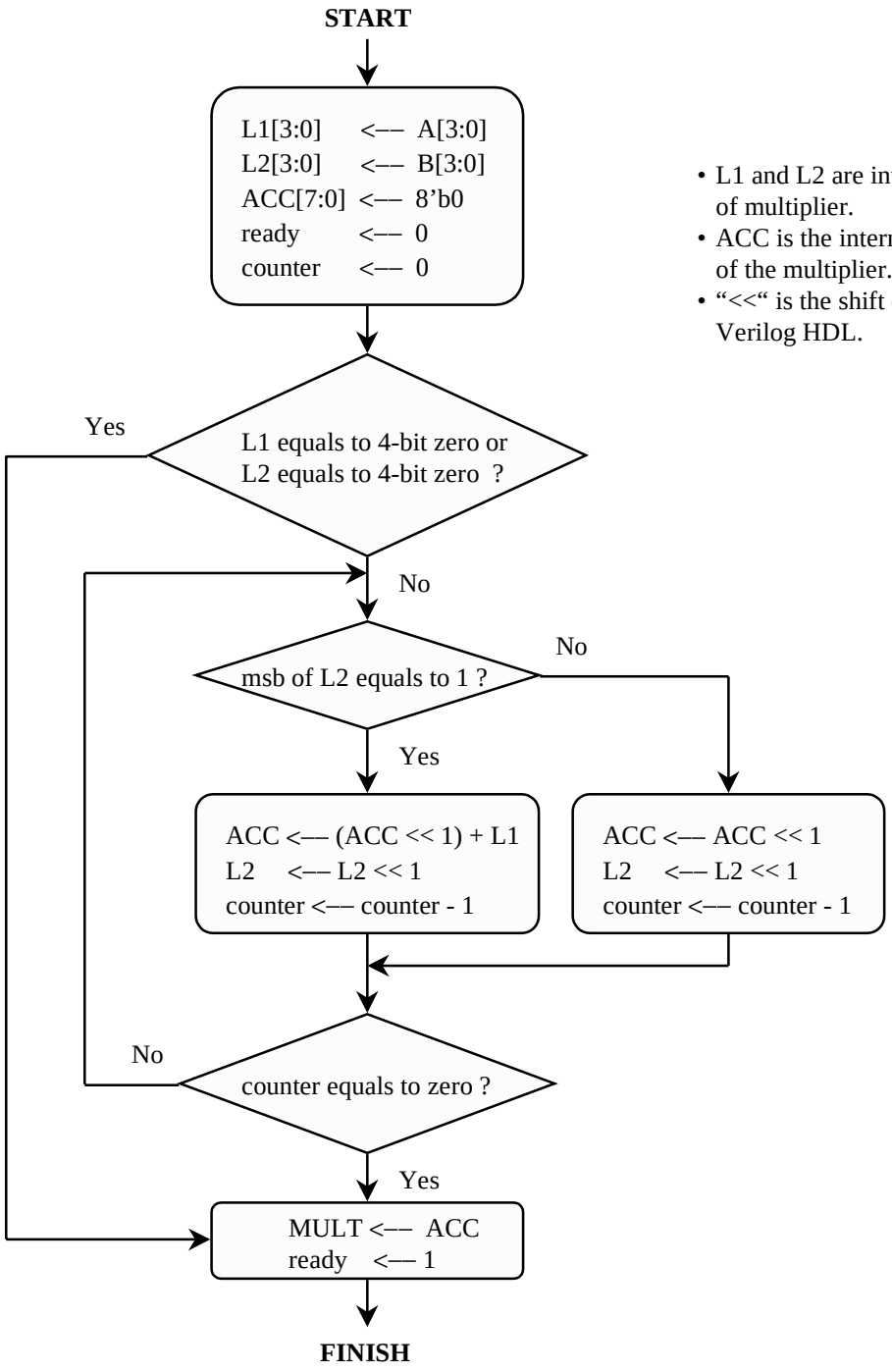
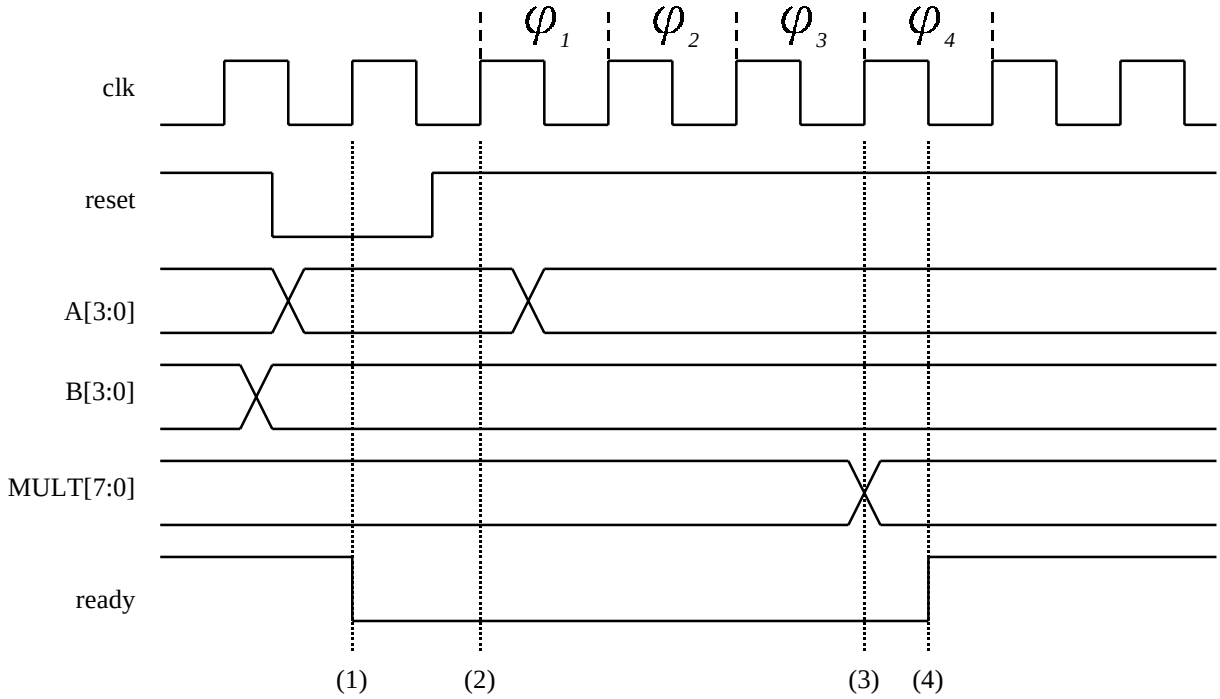


Figure 8-2. Algorithm of 4-bit serial multiplier

Timing Diagram of the 4-bit Serial Multiplier



(1) When *reset* goes from logic 1 to logic 0, serial multiplier should reset its state at the next rising edge of *clk*.

```
L1[3:0]    <-- A[3:0]
L2[3:0]    <-- B[3:0]
ACC[7:0]   <-- 8'b0
ready      <-- 0
counter    <-- 0
```

(2) When *reset* goes high, serial multiplier should start to multiply its two multicand at the next rising edge of *clk*.

(3) The result should appear at the multiplier's output *MULT* at the fourth rising edge of *clk* after *reset* signal goes high if *A* or *B* does not equal to zero.

(4) The *ready* signal should set to logic 1 at the fourth falling edge of *clk* after reset signal goes high..

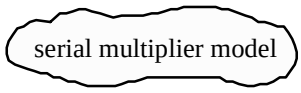
Model The Serial Multiplier

1. Change your working directory to *lab8-mult_beh*. The following files already exist in the *lab8-mult_beh* :

run.f	——>	simulation option file
testfixture_mult.v	——>	testfixture file for 4-bit serial multiplier

2. Use the following template to model the 4-bit serial multiplier :

```
module mult(MULT, ready, A, B, clk, reset);
  output [7:0] MULT;
  output ready;
  input  [3:0] A, B;
  input  clk, reset;
  reg    [7:0] MULT;
  reg    ready;
  reg    [3:0] L1, L2;
  integer counter;
```




```
endmodule
```

3. Simulate the 4-bit serial multiplier :

   unix% verilog +lic_ncv -f run.f

   If you pass the test, you can see the following simulation result displayed on your X terminal.

```
*********************************
    Total pattern tested  =       256
    Total error pattern   =         0
    Total OK pattern      =       256
*********************************
```

Summary

In this lab, you have

- realized the design algorithm of a 4-bit serial multiplier.
- learned how to use Verilog high-level programming language construct to model the 4-bit serial multiplier.
- simulated the 4-bit multiplier design.

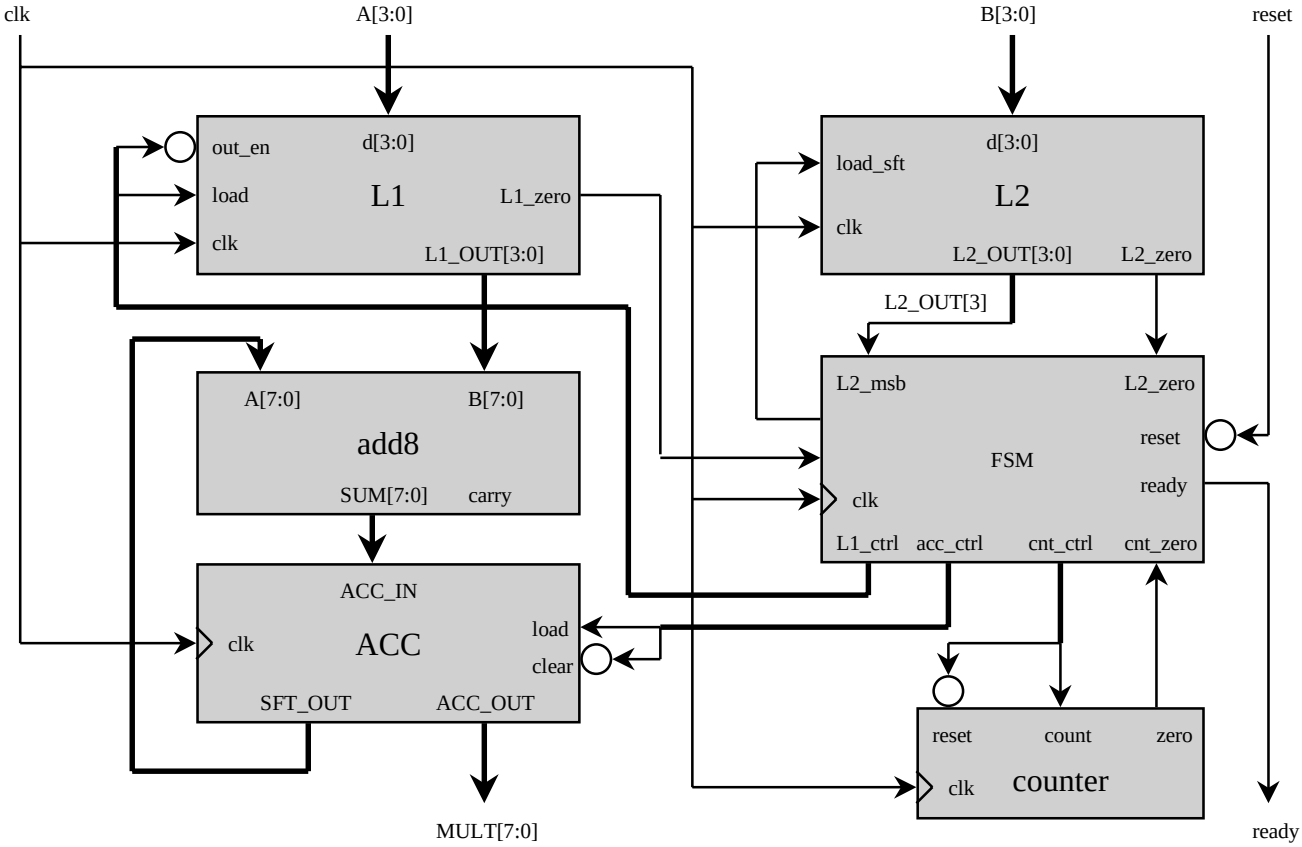
Lab 8-2. RTL Modeling of a Serial Multiplier

Objective

In this lab you will first partition the 4-bit serial multiplier into several small logic block, then model these blocks in *register-transfer logic* (RTL) level and simulate them. Finally, you will assemble these blocks into a 4-bit serial multiplier and simulate it.

Partition the 4-bit multiplier

Figure 8-3 shows the schematic of a 4-bit multiplier, which contains several small logic block with clear function definitions.



Construct the 4-bit serial multiplier

1. Change your working directory to *lab8-mult_rtl*. The following files already exists in the directory :

L1.v	——> Verilog RTL description of module *L1*.
L2.v	——> Verilog RTL description of module *L2*.
acc.v	——> Verilog RTL description of module *accumulator*.
counter.v	——> Verilog RTL description of module *counter*.
clk_gen.v	——> Verilog behavorial description of *clock generator*.
mult.v	——> Verilog structural level description of module *mult*.
run.f	——> simulation run file.
testfixture_mult.v	——> testfixture file of 4-bit serial multiplier.

   To complete the whole structure of 4-bit serial multiplier, you still need to copy finite-state machine description *fsm.v* from directory *lab7-fsm*, and the 8-bit adder description *add8.v* from directory *lab4-adder*.

```
unix% cp ../lab4-adder/add8.v .
unix% cp ../lab7-fsm/fsm.v .
```

2. Simulate the 4-bit serial multiplier :

```
unix% verilog +lic_ncv -f run.f
```

   If you pass the simulation, you will see the following messages :

```
*********************************
    Total pattern tested  =      256
    Total error pattern   =        0
    Total OK pattern      =      256
*********************************
```

Summary

In this lab you have

- utilize the finite-state machine created in *lab7* to build the RTL model of the 4-bit serial multiplier.
- simulate the 4-bit serial multiplier.

Appendix A

Solution Files for Labs

Lab2. Quick Start

add.v

```
/*********************************
  This file includes all modules
  needed to construct the 4-bit
  adder.
            Verilog LAB v1.0
*********************************/

/*****************************
  A half-adder built with
  Verilog primitives.
*****************************/
module ha(s,c,a,b);
  output s,c;
  input  a,b;

  not g0(ab,a),
      g1(bb,b);
  and g2(g2o,a,bb),
      g3(g3o,ab,b),
      g4(c,a,b);
  or  g5(s,g2o,g3o);
endmodule


/*****************************
     full-adder design
*****************************/
module fa(s,cout,a,b,cin);
  output s,cout;
  input  a,b,cin;

  ha I0(s,I0c,I1s,cin),
     I1(I1s,I1c,a,b);
  or g1(cout,I0c,I1c);
endmodule


/*****************************
   4-bit adder built with
   four pre-defined half
   -adder module.
*****************************/
module add4(Sum,carry,A,B);
  output  [3:0] Sum;
  output  carry;
  input   [3:0] A,B;
  supply0 cin;

  fa I0(Sum[0],I0c,A[0],B[0],cin),
     I1(Sum[1],I1c,A[1],B[1],I0c),
     I2(Sum[2],I2c,A[2],B[2],I1c),
     I3(Sum[3],carry,A[3],B[3],I2c);
endmodule
```

testfixture.v

```
/*************************************
  Testfixture file of a 4-bit adder
              Verilog LAB v1.0
*************************************/

module testfixture;
  reg  [3:0] A,B;
  wire [3:0] Sum;
  reg  [4:0] summary;
  integer i, j, result, flag;

// instantiate the 4-bit adder.
  add4 top(Sum,carryOut,A,B);

// Initialize all registers.
  initial
    begin
      A = 0;
      B = 0;
      flag = 0;
    end

// Apply patterns and observe result.
  initial
    begin
    // Here we apply full test patterns by two "for" loops.
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
      A = i; B=j;
      result = i+j;
      #19 summary = {carryOut,Sum};
          // Observe the output response.
          if(summary === result)
            $display("A = %b   B = %b  Result = %b",A,B,summary);
          else
            begin
              $display("A = %b   B = %b  Result = %b      NOT OK!!!",A,B,summary);
              flag = 1;
            end
      #1;
      end
    if (flag == 1)
      begin
        $display("****************************************");
        $display(" End of simulation with NOT OK result!");
        $display("****************************************");
      end
    else
      begin
        $display("*************************************");
        $display(" End of simulation with OK result!");
        $display("*************************************");
      end
    end
```



```
// Open a new waveform database "waves.shm".
  initial begin
    $shm_open("waves.shm");
    $shm_probe("AS");
    end
endmodule
```

Lab3. Structural Modeling

add4.v

```
/************************************
  4-bit adder built with four
  pre-defined half -adder module.
            Verilog LAB v1.0
************************************/

module add4(SUM,carry,A,B);
  output carry;
  output [3:0] SUM;
  input  [3:0] A,B;
  supply0 carryin;

  f_adder fa0(SUM[0],carry0,A[0],B[0],carryin),
          fa1(SUM[1],carry1,A[1],B[1],carry0),
          fa2(SUM[2],carry2,A[2],B[2],carry1),
          fa3(SUM[3],carry, A[3],B[3],carry2);
endmodule
```


cells.v

```
/************************************
  This file is the Verilog model
  of an example ASIC cell library
  which contains 3 cells :
    XO3A : 3-input XOR gate
    AN2A : 2-input AND gate
    OR3A : 3-input OR gate
            Verilog LAB v1.0
************************************/

module XO3A(z, i1, i2, i3);
  output z;
  input  i1, i2, i3;

  xor #1 (z, i1, i2, i3);
endmodule

module AN2A(z, i1, i2);
  output z;
  input  i1, i2;

  and #1 (z, i1, i2);
endmodule

module OR3A(z, i1, i2, i3);
  output z;
  input  i1, i2, i3;

  or #1 (z, i1, i2, i3);
endmodule
```

f_adder.v

```
/***********************************
  A full adder design built with
  Verilog primitive cells.
            Verilog LAB v1.0
***********************************/

module f_adder(sum,carry,a,b,c);
  output sum,carry;
  input  a,b,c;

  xor g1(sum,a,b,c);
  and g2(g2o,a,b),
      g3(g3o,b,c),
      g4(g4o,c,a);
  or  g5(carry,g2o,g3o,g4o);
endmodule
```


f_adder_cb.v

```
/***********************************
  A full adder design built with
  the cells of an example ASIC
  cell library.
            Verilog LAB v1.0
***********************************/

module f_adder(sum,carry,a,b,c);
  output sum,carry;
  input  a,b,c;

  XO3A g1(sum,a,b,c);
  AN2A g2(g2o,a,b),
       g3(g3o,b,c),
       g4(g4o,c,a);
  OR3A g5(carry,g2o,g3o,g4o);
endmodule
```

testfixture_add4.v

```
/************************************
  Testfixture file of a 4-bit adder
              Verilog LAB v1.0
************************************/

module testfixture_add4;
  reg  [3:0] A, B;
  wire [3:0] SUM;
  wire carry;
  integer i, j, flag;

// instantiate the 4-bit adder.
  add4 top(SUM,carry,A,B);

  initial
    flag = 0;

// Apply patterns and observe result.
  initial begin
    $display(" Simulation Result");
    $display("-------------------");
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
        A = i; B = j;
        #10 if ({carry, SUM} != (A + B))
              begin
                flag = 1;
                $display("   %d + %d = %d   NOT OK!!!", A, B, {carry, SUM});
              end
            else
              $display("   %d + %d = %d", A, B, {carry, SUM});
      end
    $display("");
    if (flag == 1)
      begin
      $display("************************************************");
      $display(" The 4-bit adder does not pass the simulation.");
      $display("************************************************");
      end
    else
      begin
      $display("****************************************");
      $display(" The 4-bit adder passes the simulation.");
      $display("****************************************");
      end
    end
endmodule
```


testfixture_fa.v

```
/************************************
  Testfixture file of a half adder
              Verilog LAB v1.0
************************************/

module testfixture_fa;
  reg  a, b, c, flag;
  reg  [3:0] i;
  wire sum, carry;

  f_adder top(sum,carry,a,b,c);

  initial
    flag = 0;

  initial begin
//  $display("***************************");
//  $display("   Simulation Result.");
//  $display("***************************");
    $display("   c   a   b  carry sum");
    $display("  --- --- --- ----- ---");
    for (i=0; i<=7; i=i+1)
      begin
        {c, a, b} = i[2:0];
        #10 $display("   %b   %b   %b    %b    %b", c, a, b, carry, sum);
        if ({carry, sum} != (a + b + c))
          flag = 1;
      end
    $display("");
    if (flag == 1)
      begin
      $display("************************************************");
      $display(" The full adder does not pass the simulation.");
      $display("************************************************");
      end
    else
      begin
      $display("****************************************");
      $display(" The full adder passes the simulation.");
      $display("****************************************");
      end
    end
endmodule
```

Lab4-1. Behavorial Modeling --- Combinational Circuit Example

add8.v

```
/*********************************
  A 8-bit adder modeled with
  Verilog behavioral statement.
            Verilog LAB v1.0
*********************************/

module add8(SUM, carry, A, B);
  output [7:0] SUM;
  reg    [7:0] SUM;
  output carry;
  reg    carry;
  input  [7:0] A, B;

  always @(A or B)
    {carry, SUM} = A + B;
endmodule
```


testfixture_add8.v

```
/*************************************
  Testfixture file of a 8-bit adder
            Verilog LAB v1.0
*************************************/

module testfixture_add8;
  reg  [7:0] A, B;
  wire [7:0] SUM;
  wire carry;
  integer i, j, flag;

  add8 top(SUM,carry,A,B);

  initial
    flag = 0;

  initial begin
    $display(" Snap Shot of Simulation Result");
    $display("--------------------------------");
    for (i=0; i<=255; i=i+1)
    for (j=0; j<=255; j=j+1)
      begin
        A = i; B = j;
        #10 if ({carry, SUM} != (A + B))
            begin
              flag = 1;
              $display("   %d + %d = %d   NOT OK!!!", A, B, {carry, SUM});
            end
          else
            if ((i % 79 == 0) && (j % 61 == 0))
              $display("   %d + %d = %d", A, B, {carry, SUM});
      end
    $display("");
    if (flag == 1)
      begin
      $display("************************************************");
      $display(" The 8-bit adder does not pass the simulation.");
      $display("************************************************");
      end
    else
      begin
      $display("*******************************************");
      $display(" The 8-bit adder passes the simulation.");
      $display("*******************************************");
      end
    end
endmodule
```

Lab4-2. Behavorial Modeling --- Sequential Circuit Example

shifter4.v

```
/***************************************
  A 4-bit shifter modeled with Verilog
  behavioral statement.
                Verilog LAB v1.0
***************************************/

module shifter4(PO, PI, sft_ld, clk);
  output [3:0] PO;
  reg    [3:0] PO;
  input  [3:0] PI;
  input  sft_ld, clk;

  always @(posedge clk)
    if (sft_ld == 1'b0)
      PO = PI;
    else
      PO = PO << 1;
endmodule
```


testfixture_shifter4.v

```
/***************************************
  Testfixture file of a 4-bit shifter
              Verilog LAB v1.0
***************************************/

module testfixture_shifter;
  wire [3:0] PO;
  reg  [3:0] PI, tmp;
  reg  sft_ld, clk;
  integer i, flag;

  shifter4 top(PO, PI, sft_ld, clk);

  /*********************************
    un-mark this section if you
    want to see waveform output.
  *********************************/
//  initial begin
//    $shm_open("waves.shm");
//    $shm_probe("AS");
//    end

  initial begin
    clk = 0;
    flag = 0;
    end

  always #10 clk = ~clk;

  initial begin
    for (i=0; i<16; i=i+1)
      begin
        PI = i[3:0];
        sft_ld = 0;      /* load data */
        #25 sft_ld = 1;      /* start to shift data */
            tmp = i[3:0];
        #15 if (PO != tmp << 1)
              begin
                $display("Shift left 1 bit error, PI = %b, PO = %b", PI, PO);
                flag = 1;
              end
            else
              $display("Shift %b left 1 bit = %b", PI, PO);
        #20 if (PO != tmp << 2)
              begin
                $display("Shift left 2 bit error, PI = %b, PO = %b", PI, PO);
                flag = 1;
              end
            else
              $display("Shift %b left 2 bit = %b", PI, PO);
        #20 if (PO != tmp << 3)
              begin
                $display("Shift left 3 bit error, PI = %b, PO = %b", PI, PO);
                flag = 1;
              end
```

```
        else
          $display("Shift %b left 3 bit = %b", PI, PO);
      #20 if (PO != tmp << 4)
          begin
            $display("Shift left 4 bit error, PI = %b, PO = %b", PI, PO);
            flag = 1;
          end
        else
          $display("Shift %b left 4 bit = %b", PI, PO);
    end
  if (flag == 0)
    begin
      $display("\n");
      $display("*****************************************");
      $display("  Simulation of 4-bit shifter is OK!");
      $display("*****************************************");
      $display("\n");
    end
  else
    begin
      $display("\n");
      $display("*********************************************");
      $display("  Simulation of 4-bit shifter is not OK!");
      $display("*********************************************");
      $display("\n");
    end
  $finish;
  end
endmodule
```


Lab5. Interactive Debugging

alu.v

```
/************************************
  This file includes the modules
  used to construct the 4-bit ALU.
          Verilog LAB v1.0
************************************/
module ADDER4(SUM, A, B);
  output [3:0] SUM;
  reg    [3:0] SUM;
  input  [3:0] A, B;

  always @(A or B)
    SUM = A + B;
endmodule

/************************************
     4-bit AND function design
************************************/
module AND4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A & B;
endmodule

/************************************
     4-bit OR function design
************************************/
module OR4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A | B;
endmodule

/************************************
     4-bit XOR function design
************************************/
module XOR4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A ^ B;
endmodule
```

```
/************************************
    A 2-to-4 decoder design
************************************/
module DECODER(en3, en2, en1, en0, sel);
  output en3, en2, en1, en0;
  input  [1:0] sel;

  `include "global_define.v"

  assign {en3, en2, en1, en0} = (sel == `sel_add) ? 4'b1000 :
                                (sel == `sel_and) ? 4'b0100 :
                                (sel == `sel_or ) ? 4'b0010 :
                                (sel == `sel_xor) ? 4'b0001 :
                                4'bx;
endmodule

/*****************************************
  Structural model of a 4-bit ALU design
*****************************************/
module ALU(ALU_OUT, A, B, FUNCTION);
  output [3:0] ALU_OUT;
  input  [3:0] A, B;
  input  [1:0] FUNCTION;
  wire   [3:0] ADDER_OUT, AND_OUT, OR_OUT, XOR_OUT;

  ADDER4  alu_adder(ADDER_OUT, A, B);
  AND4    alu_and(AND_OUT, A, B);
  OR4     alu_or(OR_OUT, A, B);
  XOR4    alu_xor(XOR_OUT, A, B);
  DECODER alu_decoder(en3, en2, en1, en0, FUNCTION);

  bufif1 add_buf3(ALU_OUT[3], ADDER_OUT[3], en3);
  bufif1 add_buf2(ALU_OUT[2], ADDER_OUT[2], en3);
  bufif1 add_buf1(ALU_OUT[1], ADDER_OUT[1], en3);
  bufif1 add_buf0(ALU_OUT[0], ADDER_OUT[0], en3);

  bufif1 and_buf3(ALU_OUT[3], AND_OUT[3], en2);
  bufif1 and_buf2(ALU_OUT[2], AND_OUT[2], en2);
  bufif1 and_buf1(ALU_OUT[1], AND_OUT[1], en2);
  bufif1 and_buf0(ALU_OUT[0], AND_OUT[0], en2);

  bufif0 or_buf3(ALU_OUT[3], OR_OUT[3], en1);
  bufif0 or_buf2(ALU_OUT[2], OR_OUT[2], en1);
  bufif0 or_buf1(ALU_OUT[1], OR_OUT[1], en1);
  bufif0 or_buf0(ALU_OUT[0], OR_OUT[0], en1);

  bufif1 xor_buf3(ALU_OUT[3], XOR_OUT[3], en0);
  bufif1 xor_buf2(ALU_OUT[2], XOR_OUT[2], en0);
  bufif1 xor_buf1(ALU_OUT[1], XOR_OUT[1], en0);
  bufif1 xor_buf0(ALU_OUT[0], XOR_OUT[0], en0);
endmodule
```


alu_fixed.v

```
/************************************
  This file includes the modules
  used to construct the 4-bit ALU.
          Verilog LAB v1.0
************************************/
module ADDER4(SUM, A, B);
  output [3:0] SUM;
  reg    [3:0] SUM;
  input  [3:0] A, B;

  always @(A or B)
    SUM = A + B;
endmodule

/************************************
    4-bit AND function design
************************************/
module AND4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A & B;
endmodule

/************************************
    4-bit OR function design
************************************/
module OR4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A | B;
endmodule

/************************************
    4-bit XOR function design
************************************/
module XOR4(OUT, A, B);
  output [3:0] OUT;
  input  [3:0] A, B;

  assign OUT = A ^ B;
endmodule
```

```
/************************************
    A 2-to-4 decoder design
************************************/
module DECODER(en3, en2, en1, en0, sel);
  output en3, en2, en1, en0;
  input  [1:0] sel;

  `include "global_define.v"

  assign {en3, en2, en1, en0} = (sel == `sel_add) ? 4'b1000 :
                                (sel == `sel_and) ? 4'b0100 :
                                (sel == `sel_or ) ? 4'b0010 :
                                (sel == `sel_xor) ? 4'b0001 :
                                4'bx;
endmodule

/*****************************************
  Structural model of a 4-bit ALU design
*****************************************/
module ALU(ALU_OUT, A, B, FUNCTION);
  output [3:0] ALU_OUT;
  input  [3:0] A, B;
  input  [1:0] FUNCTION;
  wire   [3:0] ADDER_OUT, AND_OUT, OR_OUT, XOR_OUT;

  ADDER4  alu_adder(ADDER_OUT, A, B);
  AND4    alu_and(AND_OUT, A, B);
  OR4     alu_or(OR_OUT, A, B);
  XOR4    alu_xor(XOR_OUT, A, B);
  DECODER alu_decoder(en3, en2, en1, en0, FUNCTION);

  bufif1 add_buf3(ALU_OUT[3], ADDER_OUT[3], en3);
  bufif1 add_buf2(ALU_OUT[2], ADDER_OUT[2], en3);
  bufif1 add_buf1(ALU_OUT[1], ADDER_OUT[1], en3);
  bufif1 add_buf0(ALU_OUT[0], ADDER_OUT[0], en3);

  bufif1 and_buf3(ALU_OUT[3], AND_OUT[3], en2);
  bufif1 and_buf2(ALU_OUT[2], AND_OUT[2], en2);
  bufif1 and_buf1(ALU_OUT[1], AND_OUT[1], en2);
  bufif1 and_buf0(ALU_OUT[0], AND_OUT[0], en2);

  bufif1 or_buf3(ALU_OUT[3], OR_OUT[3], en1);
  bufif1 or_buf2(ALU_OUT[2], OR_OUT[2], en1);
  bufif1 or_buf1(ALU_OUT[1], OR_OUT[1], en1);
  bufif1 or_buf0(ALU_OUT[0], OR_OUT[0], en1);

  bufif1 xor_buf3(ALU_OUT[3], XOR_OUT[3], en0);
  bufif1 xor_buf2(ALU_OUT[2], XOR_OUT[2], en0);
  bufif1 xor_buf1(ALU_OUT[1], XOR_OUT[1], en0);
  bufif1 xor_buf0(ALU_OUT[0], XOR_OUT[0], en0);
endmodule
```


global_define.v

```
/************************************************
  This file includes global variable definition.
                Verilog LAB v1.0
************************************************/
`define                 sel_add                 2'b11
`define                 sel_and                 2'b10
`define                 sel_or                  2'b01
`define                 sel_xor                 2'b00
```

testfixture_alu.v

```
/*****************************************
  The testfixture file of the 4-bit ALU.
                  Verilog LAB v1.0
*****************************************/

module testfixture_alu;
  wire [3:0] ALU_OUT;
  reg  [3:0] A, B;
  reg  [1:0] FUNCTION;
  integer i, j, flag;

  `include "global_define.v"

  ALU top(ALU_OUT, A, B, FUNCTION);

  initial begin
    $shm_open("waves.shm");
    $shm_probe("AS");
    end

  initial begin
    /*********************
      test ADD function
    *********************/
    flag = 0;
    FUNCTION = `sel_add;
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
    if (i + j <16)
      begin
        A = i; B = j;
        #10 if (ALU_OUT !== (i+j))
              begin
                $display("Function ADD fail!!! %d + %d = %d", A, B, ALU_OUT);
                flag = 1;
              end
//            else
//              $display("%d + %d = %d", A, B, ALU_OUT);
      end
      $display("\n");
    if (flag == 1)
      begin
        $display("************************************************");
        $display("  Simulate ALU addition function failed!!!");
        $display("************************************************");
      end
    else
      begin
        $display("********************************************");
        $display("  Simulate ALU addition function OK!!!");
        $display("********************************************");
      end
```



```
    /*********************
      test AND function
    *********************/
    flag = 0;
    FUNCTION = `sel_and;
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
        A = i; B = j;
        #10 if (ALU_OUT !== (i & j))
              begin
                $display("Function ADD fail!!! %b and %b = %b", A, B, ALU_OUT);
                flag = 1;
              end
//            else
//              $display("%b and %b = %b", A, B, ALU_OUT);
      end
      $display("\n");
    if (flag == 1)
      begin
        $display("************************************************");
        $display(" Simulate ALU logical AND function failed!!!");
        $display("************************************************");
      end
    else
      begin
        $display("********************************************");
        $display(" Simulate ALU logical AND function OK!!!");
        $display("********************************************");
      end

    /*********************
      test OR function
    *********************/
    flag = 0;
    FUNCTION = `sel_or;
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
        A = i; B = j;
        #10 if (ALU_OUT !== (i | j))
              begin
                $display("Function OR fail!!! %b or %b = %b", A, B, ALU_OUT);
                flag = 1;
              end
//            else
//              $display("%b or %b = %b", A, B, ALU_OUT);
      end
      $display("\n");
    if (flag == 1)
      begin
        $display("************************************************");
        $display(" Simulate ALU logical OR function failed!!!");
        $display("************************************************");
      end
```

```
    else
      begin
        $display("*****************************************");
        $display(" Simulate ALU logical OR function OK!!!");
        $display("*****************************************");
      end

    /*******************
      test XOR function
    *******************/
    flag = 0;
    FUNCTION = `sel_xor;
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
        A = i; B = j;
        #10 if (ALU_OUT !== (i ^ j))
              begin
                $display("Function XOR fail!!! %b xor %b = %b", A, B, ALU_OUT);
                flag = 1;
              end
//          else
//            $display("%b xor %b = %b", A, B, ALU_OUT);
      end
        $display("\n");
    if (flag == 1)
      begin
        $display("************************************************");
        $display(" Simulate ALU logical XOR function failed!!!");
        $display("************************************************");
      end
    else
      begin
        $display("*****************************************");
        $display(" Simulate ALU logical XOR function OK!!!");
        $display("*****************************************");
      end
    $display("\n");
    end
  endmodule
```


Lab6-1. Modeling Memory --- Modeling a ROM

srom16x4.v

```
/************************************************
  A 16x4 Read-Only Memory --- Verilog LAB v1.0
************************************************/

`timescale 1ns/10ps
module srom16x4(data, addr, clk);
  output [3:0] data;
  input  [3:0] addr;
  input  clk;
  reg [3:0] data, memory [15:0];

  parameter t_acc = 5;

  initial begin
    memory[0]  = 4'b1111;
    memory[1]  = 4'b1110;
    memory[2]  = 4'b1101;
    memory[3]  = 4'b1100;
    memory[4]  = 4'b1011;
    memory[5]  = 4'b1010;
    memory[6]  = 4'b1001;
    memory[7]  = 4'b1000;
    memory[8]  = 4'b0111;
    memory[9]  = 4'b0110;
    memory[10] = 4'b0101;
    memory[11] = 4'b0100;
    memory[12] = 4'b0011;
    memory[13] = 4'b0010;
    memory[14] = 4'b0001;
    memory[15] = 4'b0000;
    end

  always @(posedge clk)
    #t_acc data = memory[addr];
endmodule
```

testfixture_srom.v

```
/************************************
  Testfixture file of the 16x4 ROM
              Verilog LAB v1.0
************************************/

`timescale 1ns/10ps
module testfixture_srom;
  wire [3:0] data;
  reg  [3:0] addr;
  reg  clk;
  integer i, err_count;

  srom16x4 top(data, addr, clk);

  initial begin
    $timeformat(-9, 1, " ns", 5);
    clk = 0;
    err_count = 0;
    end

  always #10 clk = ~clk;

  initial begin
    $display("\nBegin to read data from ROM .....");
    for (i=0; i<=15; i=i+1)
      begin
        #5  addr = i;
        #11 if (data !== ~i[3:0])
            begin
              $display("Error at time %t : ROM address=%b Expected Data=%b Acquired Data=%b",
                       $time, addr, ~i[3:0], data);
              err_count = err_count + 1;
            end
        #4;
      end
    $display("****************************************");
    $display("  Completed ROM Tests With %0d Errors!", err_count);
    $display("****************************************");
    $finish;
    end
endmodule
```


Lab 6-2. Modeling Memory --- Modeling a RAM

ram16x4.v

```
/*****************************************************
   A 16x4 Random-Access Memory --- Verilog LAB v1.0
*****************************************************/

`timescale 1ns/10ps
module ram16x4(data, addr, read, write);
  inout [3:0] data;
  input [3:0] addr;
  input read, write;
  reg   [3:0] MEM [15:0];

  assign data = (read == 1'b1) ? MEM[addr] : 4'bz ;

  always @(posedge write)
    #1 MEM[addr] = data;
endmodule
```

testfixture_ram.v

```
/************************************
  Testfixture file of the 16x4 RAM
            Verilog LAB v1.0
************************************/

`timescale 1ns/10ps
module testfixture_ram;
  wire [3:0] ram_data, read_data;
  reg  [3:0] addr, data;
  reg  read, write;
  integer i, err_count;

  ram16x4 top(ram_data, addr, read, write);

  bufif1 rd_buf3(read_data[3], ram_data[3], read);
  bufif1 rd_buf2(read_data[2], ram_data[2], read);
  bufif1 rd_buf1(read_data[1], ram_data[1], read);
  bufif1 rd_buf0(read_data[0], ram_data[0], read);

  bufif1 wr_buf3(ram_data[3], data[3], write);
  bufif1 wr_buf2(ram_data[2], data[2], write);
  bufif1 wr_buf1(ram_data[1], data[1], write);
  bufif1 wr_buf0(ram_data[0], data[0], write);

  initial begin
    read  = 0;
    write = 0;
    err_count = 0;
    $display("Now write data into RAM cells .....");
    for (i=0; i<16; i=i+1)
      begin
        addr = i;
        data = ~i[3:0];
        #10 write = 1;
        #10 write = 0;
      end
    read = 1;
    $display("Now read data from RAM cells .....");
    for (i=0; i<16; i=i+1)
      begin
        #10 addr = i;
        //#10 $display("addr = %d  read_data = %b", addr, read_data);
        #10 if (read_data !== ~i[3:0])
              begin
                $display("Error at time %t : RAM address=%b Expected Data=%b Acquired Data=%b",
                  $time, addr, ~i[3:0], ram_data);
                err_count = err_count + 1;
              end
      end
```



```
    $display("************************************");
    $display("  Completed RAM Tests With %0d Errors!", err_count);
    $display("************************************");
    $finish;
    end

  initial begin
    $timeformat(-9, 2, " ns", 10);
    $shm_open("waves.shm");
    $shm_probe("AS");
    end
endmodule
```

Lab7. Finite-State Machine

fsm.v

```
/***************************************
  A finite-state machine design which
  will be used in the next lab.
                Verilog LAB v1.0
***************************************/
module fsm(ready, L1_ctrl, L2_ctrl, cnt_ctrl, acc_ctrl, clk,
  L1_zero, L2_zero, L2_msb, counter_zero, reset);
  output ready;
  output [1:0] L1_ctrl;
  output L2_ctrl;
  output [1:0] cnt_ctrl, acc_ctrl;
  input  clk, L1_zero, L2_zero, L2_msb, counter_zero, reset;
  reg    ready;
  reg    [1:0] L1_ctrl;
  reg    L2_ctrl;
  reg    [1:0] cnt_ctrl, acc_ctrl;
  reg    [1:0] curr_state, next_state;

  `include "/users/cic/chtsai/vlog/lab/lab-mult_rtl/parts/fsm/global_def.v"

  always @(negedge clk)
    if (reset == 0)
      curr_state = init;
    else
      curr_state = next_state;

  always @(curr_state or L1_zero or L2_zero or L2_msb or counter_zero)
    case (curr_state)
      init    : if ((L1_zero == 1) || (L2_zero == 1))
                  next_state = done;
                else if (L2_msb == 1)
                  next_state = add_sft;
                else if (L2_msb == 0)
                  next_state = shift;
      add_sft : if (counter_zero == 1)
                  next_state = done;
                else if (L2_msb == 1)
                  next_state = add_sft;
                else if (L2_msb == 0)
                  next_state = shift;
      shift   : if (counter_zero == 1)
                  next_state = done;
                else if (L2_msb == 1)
                  next_state = add_sft;
                else if (L2_msb == 0)
                  next_state = shift;
      done    : next_state = done;
    endcase

  always @(curr_state)
    case (curr_state)
```



```
      init    : begin
                  L1_ctrl  = `L1_load;
                  L2_ctrl  = `L2_load;
                  cnt_ctrl = `cnt_reset;
                  acc_ctrl = `acc_clear;
                  ready    = 1'b0;
                end
      add_sft : begin
                  L1_ctrl  = `L1_outL1;
                  L2_ctrl  = `L2_shift;
                  cnt_ctrl = `cnt_count;
                  acc_ctrl = `acc_add;
                  ready    = 1'b0;
                end
      shift   : begin
                  L1_ctrl  = `L1_outzero;
                  L2_ctrl  = `L2_shift;
                  cnt_ctrl = `cnt_count;
                  acc_ctrl = `acc_add;
                  ready    = 1'b0;
                end
      done    : begin
                  L1_ctrl  = `L1_outzero;
                  L2_ctrl  = `L2_shift;
                  cnt_ctrl = `cnt_idle;
                  acc_ctrl = `acc_idle;
                  ready    = 1'b1;
                end
    endcase
endmodule
```

global_def.v

```
/*************************************
  define the state vector of FSM
*************************************/
parameter       init                            = 2'b00;
parameter       add_sft                         = 2'b01;
parameter       shift                           = 2'b10;
parameter       done                            = 2'b11;

/*************************************
  define the output control signal
  of FSM
*************************************/
`define                         L1_load                         2'b11
`define                         L1_outL1        2'b00
`define                         L1_outzero      2'b01
`define         L2_load                         1'b1
`define                         L2_shift        1'b0
`define                         cnt_reset       2'b00
`define                         cnt_count       2'b11
`define                         cnt_idle        2'b10
`define                         acc_clear       2'b00
`define                         acc_add                         2'b11
`define                         acc_idle        2'b10
```


testfixture_fsm.v

```
/*************************************************
  Testfixture file of the FSM.  Please verify the
  state transition by using SimWave.
                        Verilog LAB v1.0
*************************************************/
module testfixture_fsm;
  `include "global_def.v"
  wire ready;
  wire [1:0] L1_ctrl;
  wire L2_ctrl;
  wire [1:0] cnt_ctrl ,acc_ctrl;
  reg  L1_zero, L2_zero, L2_msb, counter_zero, reset;

  CK  ckgen(clk);
  fsm top(ready, L1_ctrl, L2_ctrl, cnt_ctrl, acc_ctrl, clk,
          L1_zero, L2_zero, L2_msb, counter_zero, reset);

  initial begin
    $shm_open("waves.shm");
    $shm_probe("AS");
    end

  initial begin
    reset = 1; L1_zero = 0; L2_zero = 0; L2_msb = 1; counter_zero = 0;
    $display("Reset FSM.....");
    #5  reset = 0;
    #50 reset = 1;
    #75 counter_zero = 1;

    $display("Test \(init\)->\(add_shift\)->\(add_shift\)->\(add_shift\)->\(add_shift\)->\(done\) flow");
    #35 reset = 0;
       counter_zero = 0;
    #30 reset = 1;
       L2_msb = 0;
    #75 counter_zero = 1;

    $display("Test \(init\)->\(shift\)->\(shift\)->\(shift\)->\(shift\)->\(done\) flow");
    #35 reset = 0;
       counter_zero = 0;
    #30 reset = 1;
    #75 counter_zero = 1;

    $display("Test \(init\)->\(add_shift\)->\(shift\)->\(add_shift\)->\(shift\) -> \(done\) flow");
    #35 reset = 0;
       counter_zero = 0;
    #30 reset = 1;
       L2_msb = 1;
    #15 L2_msb = 0;
    #20 L2_msb = 1;
    #20 L2_msb = 0;
       counter_zero = 1;
    #100 $finish;
    end
endmodule
```

Lab8-1. Top Down Design Example --- Behavorial Modeling of a Serial Multiplier

mult.v

```
/*********************************************
  Behavioral description of a 4-bit serial
  multiplier.
                Verilog LAB v1.0
*********************************************/

module mult(MULT, ready, A, B, clk, reset);
  output [7:0] MULT;
  output ready;
  input  [3:0] A, B;
  input  clk, reset;
  reg    [7:0] ACC, MULT;
  reg    ready;
  reg    [3:0] L1, L2;
  integer counter;
  event  start;

  always @(negedge reset)
    begin
      @(posedge clk);
      L1 = A;
      L2 = B;
      ACC = 0;
      ready = 0;
      counter = 4;
      @(posedge reset)
      -> start;
    end

  always @(start)
    begin : mult_block
    if (reset == 1'b1)
      if ((L1 == 4'b0) || (L2 == 4'b0))
        begin
          @(posedge clk);
          ACC = 8'b0;
          MULT = ACC;
          ready = 1'b1;
        end
      else
        begin
          while (counter !== 0)
            begin
              @(posedge clk);
              if (L2[3] == 1'b1)
                ACC = (ACC << 1) + L1;
              else
                ACC = ACC << 1;
              L2 = L2 << 1;
              counter = counter - 1;
            end
          MULT = ACC;
          @(negedge clk) ready = 1'b1;
        end
    end
endmodule
```


testfixture_mult.v

```
/*****************************************************
  Testfixture file of the 4-bit serial multiplier.
                Verilog LAB v1.0
*****************************************************/

module testfixture_mult;
  wire [7:0] MULT;
  reg  [3:0] A, B;
  reg  reset, clk;
  integer i, j;
  integer ok_count, err_count, total;

  mult top(MULT, ready, A, B, clk, reset);

  initial begin
    $shm_open("waves.shm");
    $shm_probe("AS");
    end

  initial begin
    ok_count = 0; err_count = 0; total = 0;
    reset = 1;
    clk = 0;
    end

  always #10 clk = ~clk;

  initial begin
    for (i=0; i<=15; i=i+1)
    for (j=0; j<=15; j=j+1)
      begin
        #5  reset = 0;
        A = i; B = j;
        #20 reset = 1;
        @(posedge ready);
        if(MULT != A*B)
          begin
            $display("Error in time %t : %d * %d = %d", $time, A, B, MULT);
            err_count = err_count + 1;
          end
        else
          ok_count = ok_count + 1;
        total = total + 1;
        end
    $display("\n");
    $display("*********************************************");
    $display("       Simulation Statistic Report");
    $display("*********************************************");
    $display("  Total pattern tested = %d", total);
    $display("  Total error pattern  = %d", err_count);
    $display("  Total OK pattern     = %d", ok_count);
    $display("*********************************************");
    $display("\n");
    #100 $finish;
    end
endmodule
```

Lab 8-2. Top Down Design Example --- RTL Modeling of a Serial Multiplier

acc.v

```
/*************************************************************
  Behavorial description of Accumulator --- Verilog LAB v1.0
*************************************************************/

module ACC(ACC_OUT, SFT_OUT, ACC_IN, clk, clear, load);
  output [7:0] ACC_OUT, SFT_OUT;
  reg    [7:0] ACC_OUT, SFT_OUT;
  input  [7:0] ACC_IN;
  input  clk, clear, load;

  always @(clear)
    if (clear == 1'b0)
      begin
        assign ACC_OUT = 8'b0;
        assign SFT_OUT = 8'b0;
      end
    else
      begin
        deassign ACC_OUT;
        deassign SFT_OUT;
      end

  always @(ACC_IN or clk or load)
    if ((clk == 1) && (load == 1)) ACC_OUT = ACC_IN;

  always @(ACC_OUT or clk)
    if (clk == 0) SFT_OUT = ACC_OUT << 1;
endmodule
```


clk_gen.v

```
/*********************************
  A 50MHz clock signal generator
            Verilog LAB v1.0
*********************************/
module CK(clk);
  output clk;
  reg    clk;

  initial clk = 0;
  always #10 clk = ~clk;
endmodule
```

counter.v

```
/***********************************
  The behavorial description of a
  4-down-to-0 counter.
              Verilog LAB v1.0
***********************************/

module counter(zero, reset, count, clk);
  output zero;
  input  reset, count, clk;
  reg    [3:0] count_value;
  reg    zero;

  always @(reset)
    if (reset == 1'b0)
      assign count_value = 4'd4;
    else
      deassign count_value;

  always @(posedge clk)
    if (count == 1)
      count_value = count_value - 1;

  always @(count_value)
    if (count_value == 4'b0000)
      zero = 1'b1;
    else
      zero = 1'b0;
endmodule
```


L1.v

```
/************************************************************
  Behavorial description of L1 latch --- Verilog LAB v1.0
************************************************************/

module L1(L1_OUT, L1_zero, d, clk, load, out_en);
  output [3:0] L1_OUT;
  output L1_zero;
  input  [3:0] d;
  input  clk, load, out_en;
  reg    [3:0] L1_reg;
  reg    L1_zero;

  always @(posedge clk)
    if (load == 1) L1_reg = d;

  assign L1_OUT = (out_en == 1'b0) ? L1_reg : 4'b0000;

  initial L1_zero = 0;

  always @(L1_reg)
    if (L1_reg === 4'b0000)
      L1_zero = 1;
    else
      L1_zero = 0;
endmodule
```

L2.v

```
/*************************************************************
  Behavorial description of L2 latch --- Verilog LAB v1.0
*************************************************************/

module L2(L2_OUT, L2_zero, d, clk, load_sft);
  output [3:0] L2_OUT;
  output L2_zero;
  input  [3:0] d;
  input  clk, load_sft;
  reg    [3:0] L2_OUT;
  reg    L2_zero;

  always @(posedge clk)
    if (load_sft == 1)
      L2_OUT = d;
    else
      L2_OUT = L2_OUT << 1;

  initial L2_zero = 0;

  always @(L2_OUT)
    if (L2_OUT === 4'b0000)
      L2_zero = 1;
    else
      L2_zero = 0;
endmodule
```


mult.v

```
/*************************************************************
  Structural description of the 4-bit serial multiplier.
                        Verilog LAB v1.0
*************************************************************/

module mult(PRODUCT, ready, A, B, clk, reset);
  output [7:0] PRODUCT;
  output ready;
  input  [3:0] A, B;
  input  clk, reset;
  wire   [3:0] L1_OUT, L2_OUT;
  wire   [1:0] L1_ctrl;
  wire   [7:0] ADDER_SUM, ACC_SFT_OUT;
  wire   [1:0] cnt_ctrl, acc_ctrl;

  L1   L1_LATCH(L1_OUT[3:0], L1_zero, A[3:0], clk, L1_ctrl[1], L1_ctrl[0]);
  L2   L2_LATCH(L2_OUT[3:0], L2_zero, B[3:0], clk, L2_ctrl);
  add8 ADDER(ADDER_SUM[7:0], , ACC_SFT_OUT[7:0], {4'b0, L1_OUT[3:0]});
  ACC  ACCUMULATOR(PRODUCT[7:0], ACC_SFT_OUT[7:0], ADDER_SUM[7:0], clk,
                   acc_ctrl[1], acc_ctrl[0]);
  fsm  FSM(ready, L1_ctrl[1:0], L2_ctrl, cnt_ctrl[1:0], acc_ctrl[1:0], clk,
           L1_zero, L2_zero, L2_OUT[3], counter_zero, reset);
  counter COUNTER(counter_zero, cnt_ctrl[1], cnt_ctrl[0], clk);
endmodule
```