Verilog HDL Overview

Prof. Chien-Nan Liu
Dept. of Electrical Engineering
National Central University

Tel: (03)4227151 ext:34534
Fax: (03)4255830
E-mail: jimmy@ee.ncu.edu.tw
URL: http://www.ee.ncu.edu.tw/~jimmy

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

Why Use an HDL ?

- Hard to design directly for complex systems
- Formal description using HDL
  - Verify the specification through simulation or verification
  - Easy to change
  - Enable automatic synthesis
- Allow architectural tradeoffs with short turnaround
- Reduce time for design capture
- Encourage focus on functionality
- Shorten the design verification loop

*HDL = Hardware Description Language

2-3

Hardware Description Language

- Have high-level language constructs to describe the functionality and connectivity of the circuit
- Can describe a design at some levels of abstraction
  - Behavioral, RTL, Gate-level, Switch
- Can describe functionality as well as timing
- Can be used to model the concurrent actions in real hardware
- Can be used to document the complete system design tasks
  - testing, simulation … related activities
- Comprehensive and easy to learn

2-4

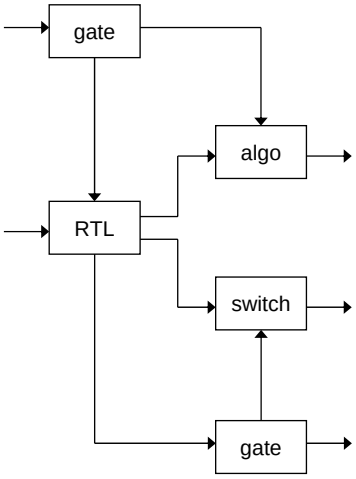
Verilog History

- Gateway Design Automation
  - Phil Moorbr in 1984 and 1985
- Verilog-XL, “XL algorithm”, 1986
  - Fast gate-level simulation
- Verilog logic synthesizer, Synopsys, 1988
  - Top-down design methodology
- Cadence Design Systems acquired Gateway, 1989
  - A proprietary HDL
- Open Verilog International (OVI), 1991
  - Language Reference Manual (LRM)
- The IEEE 1364 working group, 1994
- Verilog became an IEEE standard
  - December, 1995

2-5

What is Verilog HDL ?

- Hardware description language
- Mixed level modeling
  - Behavioral
    - Algorithmic
    - Register transfer
  - Structural
    - Gate
    - Switch
- Single language for design and simulation
- Built-in primitives and logic functions
- User-defined primitives
- Built-in data types
- High-level programming constructs



2-6

Books

- Palnitkar S.," Verilog HDL: A Guide to Digital Design and Synthesis", Prentice Hall, NJ, 1996. (ISBN: 0-13-451675-3)
- Thomas D. and P. Moorby, "The Verilog Hardware Description Language", Kluwer Academic, MA, 1991. (ISBN: 0-7923-9126-8)
- Sternheim E., E. Singh, and Y. Trivedi, "Digital Design with Verilog HDL", Automata Publishing Company, CA, 1990. (ISBN: 0-9627488-0-3)

Official Language Document:

- "Verilog Hardware Description Language Reference Manual", IEEE Std 1364-1995, IEEE.

2-7

Outline

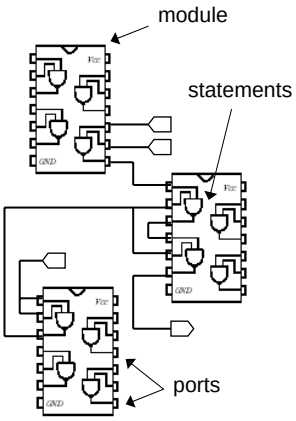
- Introduction
- Language elements
  - Modules
  - Lexical conventions
  - Data types
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

2-8

Basic Unit -- Module

- Modules communicate externally with input, output and bi-directional ports
- A module can be instantiated in another module

module module_name (port_list);
declarations:
  port declaration (input, output, inout, …)
  data type declaration (reg, wire, parameter, …)
  task and function declaration
statements:
  initial block
  always block — Behavioral
  module instantiation
  gate instantiation
  UDP instantiation — Structural
  continuous assignment — Data-flow
endmodule



2-9

An Example

```
module FA_MIX (A, B, CIN, SUM, COUT);
  input A,B,CIN;
  output SUM, COUT;
  reg COUT;
  reg T1, T2, T3;
  wire S1;

  xor X1 (S1, A, B); // Gate instantiation.

  always @ (A or B or CIN)   // Always Block
  begin
    T1 = A & CIN;
    T2 = B & CIN;
    T3 = A & B;
    COUT = (T1 | T2 | T3);
  END
  assign SUM = S1 ^ CIN;   // Continuous assignment
endmodule
```

2-10

Structural Hierarchy Description Style

- Direct instantiation and connection of models from a separate calling model
  - Form the structural hierarchy of a design
- A module may be declared anywhere in a design relative to where it is called
- Signals in the higher “calling” model are connected to signals in the lower “called” model by either:
  - Named association
  - Positional association

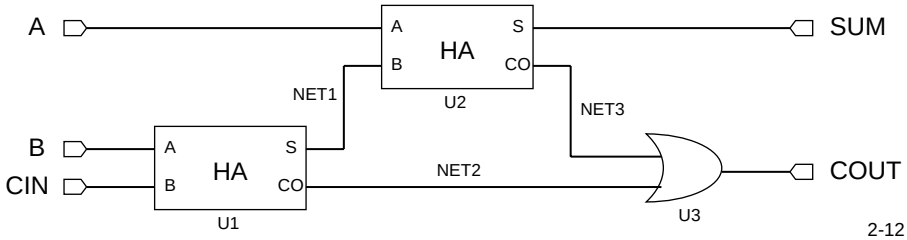
2-11

Structural Hierarchy Description Style

- Example: (Full Adder)

```
module FULL_ADD (A, B, CIN, SUM, COUT);
   input A, B, CIN;
   output SUM, COUT;
   wire NET1, NET2, NET3;
   HA  U1(NET1, NET2, B, CIN);   /* positional association */
   HA  U2(.S(SUM), .CO(NET3), .A(A), .B(NET1));   /* named */
   OR2  U3(COUT, NET2, NET3);
endmodule
```

instance name

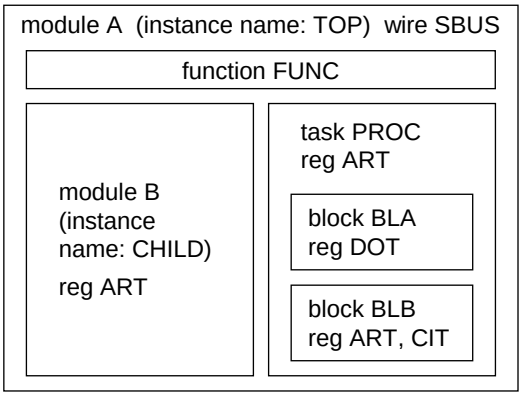


2-12

Hierarchical Path Name

- Every identifier has a unique hierarchical path name
- Period character (**.**) is the separator
- New hierarchy is defined by: module instantiation, task definition, function definition, named block

TOP.SBUS
TOP.CHILD.ART
TOP.PROC.ART
TOP.PROC.BLB.CIT
TOP.PROC.BLA.DOT



2-13

Lexical Conventions

- Verilog is a free-format language
  - Like C language
- White space (blank, tab, newline) can be used freely
- Verilog is a **case-sensitive** language
- Identifiers
  - User-provided names for Verilog objects in the descriptions
  - Legal characters are “a-z”, “A-Z”, “0-9”, “_”, and “$”
    - First character has to be a letter or an “_”
  - Example: Count, _R2D2, FIVE$
- Keywords
  - Predefined identifiers to define the language constructs
  - All keywords are defined in lower case
  - Cannot be used as idenfiers
  - Example: **initial, assign, module**

2-14

Lexical Conventions

- Comments: two forms

```
/* First form: can
     extend over many
        lines */
// Second form: ends at the end of this line
```

- Strings
  - Enclosed in double quotes and must be specified in one line
    - “Sequence of characters”
  - Accept C-liked escape character
    - \n = newline
    - \t = tab
    - \\ = backslash
    - \” = quote mark (“)
    - %% = % sign

2-15

Lexical Conventions

- System tasks / function
  - Execute the built-in tasks and functions
  - Frequently used system tasks / functions
    - **$time**: report the current simulation time
    - **$display**: display the values of signals
    - **$monitor**: continuously monitor the values of signals
    - **$stop**: stop the simulation
    - **$finish**: quit the simulation
- Compiler directive:
  - Remain active through the rest of compilation until they are overridden or deactived
  - Frequently used compiler directives
    - **`define** <name><macro_text>:
      <name> will substitute <macro_text> at compile time
    - **`include** <file_name>:
      include the contents of the file named as <file_name>

2-16

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

Integer Numbers

- Sized integers
  - For unsigned integers only
  - Representation form: [<size>] ‘ <base><value>
    where
    <size> is the size in bits (can be ignored)
    <base> can be b(binary), o(octal), d(decimal), or h(hexadecimal)
    <value> is any legal number in selected base and X(unknown), Z(high-impedence) (binary only)
  - Example: `o721 (9-bit octal), 4’d2 (4-bit decimal), 4’bz (4-bit z, z extended)
- Unsized integers
  - Signed decimal integers in two’s complement form
  - Example: 32 (decimal 32), -15 (decimal -15)

Real Numbers

- Decimal notation
  - 11.2
  - 1.572
  - 0.1
- Scientific notation

235.1e2	23510.0	
3.6E2	360.0	(e is the same as E)
5E-4	0.0005	

- Must have at least one digit on either side of decimal
- Stored and manipulated in double precision (usually 64-bits)

2-19

Value Set

- 0: logic-0 / FALSE
- 1: logic-1 / TRUE
- x: unknown / don’t care, can be 0, 1 or z.
- z: high-impedance

2-20

Data Types

- Nets
  - Connects between structural elements
  - Values come from its drivers
    - Continuous assignment
    - Module or gate instantiation
  - If no drivers are connected to net, default value is Z
- Registers
  - Represent abstract data storage elements
  - Manipulated within procedural blocks
  - The value in a register is saved until it is overridden
  - Default value is X

2-21

Net Types

- **wire, tri**: standard net
- **wor, trior**: wired-or net
- **wand, triand**: wired-and net
- **trireg**: capacitive
  - If all drivers at z, previous value is retained
- **tri1**: pull up (if no driver, 1)
- **tri0**: pull down (if no driver, 0)
- **supply0**: ground
- **supply1**: power
- A net that is not declared defaults to a 1-bit wire

```
wire reset;
wor [7:0] DBUS;
supply0 GND;
```

2-22

Register Types

- **reg**: any size, unsigned
- **integer**: 32-bit signed (2's complement)
- **time**: 64-bit unsigned
- **real, realtime**: 64-bit real number
  - Defaults to an initial value of 0
- Examples:

```
reg CNT;
reg [31:0] SAT;
integer A, B, C;   // 32-bit
real SWING;
realtime CURR_TIME;
time EVENT;
```

2-23

Parameters

- Constant
- Can be modified at compilation time
  - Use ***defparam*** statement
- Examples:

```
parameter LINE_LENGTH = 132, ZLL_X_S = 16'bx;
parameter BIT = 1, BYTE = 8, PI = 3.14;
parameter SROBE_DELAY = (BYTE + BIT) / 2;
parameter TQ_FILE = "/home/jimmy/TEST/add.tq";
```

- Common usage
  - Specify delays and widths

2-24

Memories

- Array of registers
- No multiple dimensions

  reg [3:0] MY_MEM [0:63]; // 64 4-bit registers
- Entire memory cannot be assigned a value in a single assignment

  reg [1:5] DIG; // 5-bit register

  DIG = 00000

  reg BOG [1:5]; // 5 1-bit register

  {BOG[1], BOG[2], …, BOG[5]} = 00000;
- Can load memory by using a system task

  $readmem<base>(“<filename>”, <memory_name>, <start_addr>, <finish_addr>);

  where <base> can be b(binary) or h(hexadecimal)

Outline

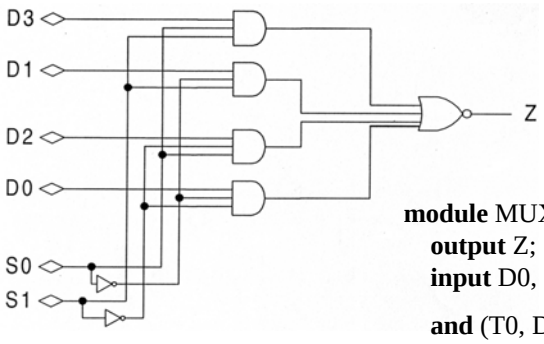
- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

Primitive Gates

- The following gates are built-in types in the simulator
- **and, nand, nor, or, xor, xnor**
  - First terminal is output, followed by inputs
    and a1 (out1, in1, in2);
    nand a2 (out2, in21, in22, in23, in24);
- **buf, not**
  - One or more outputs first, followed by one input
    not N1 (OUT1, OUT2, OUT3, OUT4, INA);
    buf B1 (BO1, BIN);
- **bufif0, bufif1, notif0, notif1**: three-state drivers
  - Output terminal first, then input, then control
    bufif1 BF1 (OUTA, INA, CTRLA);
- **pullup, pulldown**
  - Put 1 or 0 on all terminals
    pullup PUP (PWRA, PWRB, PWRC);
- Instance names are optional
    ex: **not** (QBAR, Q)

2-27

Example



4 X 1 multiplexer circuit

```
module MUX4x1 (Z, D0, D1, D2, D3, S0, S1);
   output Z;
   input D0, D1, D2, D3, S0, S1;

   and (T0, D0, S0BAR, S1BAR),
       (T1, D1, S0BAR, S1),
       (T2, D2, S0, S1BAR),
       (T3, D3, S0, S1);

   not (S0BAR, S0),
      (S1BAR, S1);

   nor (Z, T0, T1, T2, T3);
endmodule
```

2-28

Array of Instances

- An array of instances can be specified using the range specification

```
wire [3:0] OUT, INA, INB;
……
nand GANG[3:0] (OUT, INA, INB);

// This is the same as:
nand GANG3 (OUT[3], INA[3], INB[3]),
     GANG2 (OUT[2], INA[2], INB[2]),
     GANG1 (OUT[1], INA[1], INB[1]),
     GANG0 (OUT[0], INA[0], INB[0]);
```

2-29

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

2-30

Data-Flow Description Style

- Models behavior of combinational logic
- Assign a value to a net using ***continuous assignment***
- Examples:

```
wire [3:0] Z, PRESET, CLEAR;
assign Z = PRESET & CLEAR;

wire COUT, CIN;
wire [3:0] SUM, A, B;
assign {COUT, SUM} = A + B + CIN;
```

- Left-hand side (target) expression can be a:
  - Single net (ex: Z)
  - Part-select (ex: SUM[2:0])
  - Bit-select (ex: Z[1])
  - Concatenation of both (ex: {COUT, SUM[3:0]})
- Expression on right-hand side is evaluated whenever any operand value changes

2-31

Delays

- Delay between assignment of right-hand side to left-hand side

```
assign #6 ASK = QUIET || LATE;   //Continuous delay
```

- Net delay

```
wire #5 ARB;
// Any change to ARB is delayed 5 time units before it
   takes effect
```

- If value changes before it has a chance to propagate, latest value change will be applied
  - Inertial delay

2-32

Operators

- The 9 functional groups of operators are:
  1. Arithmetic
  2. Relational
  3. Equality
  4. Logical
  5. Bit-wise
  6. Reduction
  7. Shift
  8. Conditional
  9. Concatenation

Arithmetic Operators

+	(plus)	*	(multiply)
-	(minus)	/	(divide)
%	(modulus)		

- Integer division will truncate
- **%** gives the remainder with the sign of the first operand
- If any bit of operand is ***x*** or ***z***, result is ***x***
- *reg* data type holds an unsigned value, while *integer* data type holds a signed value

```
reg [7:0] BAR;
integer TAB;
BAR = -4`d6;    // reg BAR has value unsigned 10
TAB = -4`d6;    // integer TAB has value signed -6

BAR-2           // result is 8
TAB-2           // result is -8
```

Relational Operators

>	(greater than)
<	(less than)
>=	(greater than or equal to)
<=	(less than or equal to)

- If ***x*** or ***z*** in operand, result is ***x***
- If unequal bit lengths, smaller operand is zero-filled on most significant side (i.e. on left)

2-35

Equality Operators

==	(logical equality, result may be unknown)
!=	(logical inequality, result may be unknown)
===	(case equality, x and z also compared)
!==	(case inequality, x and z also compared)

```
A = `b11x0;
B = `b11x0;
```

(A == B) is unknown
(A === B) is true

- Unknown (comparison failed) is same as false in synthesis
- Compare bit by bit, zero-filling on most significant side

2-36

Logical Operators

&&	(logical and)
||	(logical or)
!	(unary logical negation)

```
A = `b0110;    // non zero value
B = `b0100;    // non zero value
```

(A || B) is 1
(A && B) is also 1

- Non-zero value is treated as 1
- If result is ambiguous, set to x

2-37

Bit-Wise Operators

~	(unary negation)
&	(binary and)
|	(binary or)
^	(binary exclusive-or)
~^, ^~	(binary exclusive-nor)

```
A = `b0110;
B = `b0100;
```


operate on the corresponding bits

A | B = 0110
A & B = 0100

- If operand sizes are unequal, smaller one is zero-filled on the most significant bit side

2-38

Reduction Operators

&	(reduction and)
~&	(reduction nand)
|	(reduction or)
~|	(reduction nor)
^	(reduction xor)
~^	(reduction xnor)

```
A = `b0110;
B = `b0100;

| A = 1
& B = 0
```

- Bit-wise operation on a single operand to produce 1-bit result

Shift Operators

<<	(left shift)
>>	(right shift)

```
reg [7:0] QPEG;
QPEG = 4`b0111;

QPEG >> 2  has the value 0001
```

- Logical shift
- Fill vacated bits by 0
- If right operand is ***x*** or ***z***, result is ***x***
- Right operand is always an unsigned number

Other Operators

- Conditional operator

  expr1 ? expr2 : expr3

```
wire [2:0] STUDENT = MARKS > 18 ? GRADE_A : GRADE_C;
```

- Concatenation

```
wire [7:0] DBUS, BUS_A, BUS_B;
assign BUS_A [7:4] = {DBUS[3], DBUS[2], DBUS[1], DBUS[0]};
assign BUS_B = {DBUS[3:0], DBUS[7:4]};
```

- Replication

```
wire [7:0] DBUS;
wire [11:0] ABUS;
assign ABUS = { 3 {4`b1011}};              // 1011 1011 1011
assign ABUS = {{4 {DBUS[7]}}, DBUS};       // sign extension
```

2-41

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

2-42

Behavioral Modeling

- Procedural blocks:
  - **initial block**: executes only once
  - **always block**: executes in a loop
- Block execution is triggered based on user-specified conditions
  - **always** @ (**posedge** clk) …….
- All procedural blocks are automatically activated at time 0
- All procedural blocks are executed **concurrently**
- **reg** is the main data type that is manipulated within a procedural block
  - It holds its value until assigned a new value

2-43

Initial Statement

- Executes only once at the beginning of simulation

```
initial
   statements
```

- Used for initialization and waveform generation

```
//Initialization:
reg [7:0] RAM[0:1023];
reg RIB_REG;

initial
begin
   integer INX;

   RIB_REG =0;
   for (INX = 0; INX < 1024; INX = INX + 1)
         RAM[INX] = 0;
end
```

group multiple statements

2-44

Always Statement

- Executes continuously; must be used with some form of timing control

always *(timing_control)*
  statements

```
always
  CLK = ~CLK
  // Will loop indefinitely
```

- Four forms of event expressions are often used
  - An **OR** of several identifiers (comb/seq logic)
  - The rising edge of a identifier (for clock signal of a register)
  - The falling edge of a identifier (for clock signal of a register)
  - Delay control (for waveform generator)
- Any number of *initial* and *always* statements may appear within a module
- Initial and always statements are all executed in parallel

Truth Table to Verilog

```
module COMB(A, B, Y1, Y2);
   input  A, B;
   output  Y1, Y2;
   reg  Y1, Y2;

  always @(A or B)
   begin
    case ({A, B})
     2'b 00 : begin Y1=1; Y2=0; end
     2'b 01 : begin Y1=1; Y2=0; end
     2'b 10 : begin Y1=1; Y2=0; end
     2'b 11 : begin Y1=0; Y2=1; end
    endcase
  end

endmodule
```

Any value changes of A or B will trigger this block

A	B	Y1	Y2
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Other Examples

```
module example (D, CURRENT_STATE, Q, NEXT_STATE);
   input D, CURRENT_STATE;
   output Q, NEXT_STATE;
   reg CLK, Q, NEXT_STATE;

   always #5 CLK = ~CLK;

   always @(posedge CLK)
   begin
      Q = D;
   end

   always @(negedge CLK)
   begin
      NEXT_STATE = CURRENT_STATE;
   end

endmodule
```

delay-controlled always block
clock period = 10

activated when CLK has
a 0 -> 1 transition

activated when CLK has
a 1 -> 0 transition

2-47

Procedural Assignments

- The assignment statements that can be used inside an *always* or *initial* block
- The target must be a register or integer type
- The following forms are allowed as a target
  - Register variables
  - Bit-select of register variables (ex: A[3])
  - Part-select of register variables (ex: A[4:2])
  - Concatenations of above (ex: {A, B[3:0]})
  - Integers
- Two kinds of procedural assignments
  - Blocking assignment
  - Non-blocking assignment

2-48

Blocking v.s. Non-Blocking

- Blocking assignment (=)
  - Assignments are blocked when executing
  - The statements will be executed in sequence, one after one

```
always @(posedge CLK) begin
        B = A;
        C = B;
end
```

(1 flip-flop, data is B, data out is C)

- Non-blocking assignment (<=)
  - Assignments are not blocked
  - The statements will be executed concurrently

```
always @(posedge CLK) begin
        B <= A;
        C <= B;
end
```

(2 pipelined flip-flops, A to B to C)

2-49

More Examples

```
// Blocking assignment:
initial
begin
   CLR = #5 1;
   CLR = #4 0;
   CLR = #10 1;
end

/* CLR is assigned 1 at time 5,
0 at time 9, and 1 at time 19 */
```

```
// Non-blocking assignment:
initial
begin
   CLR <= #5 1;
   CLR <= #4 0;
   CLR <= #10 1;
end

/* CLR is assigned 0 at time 4,
1 at time 5, and 1 at time 10 */
```

* Value is indeterminated if multiple values are assigned at the same time.

2-50

Procedural Continuous Assignments

- Allows expression to be driven continuously into registers or nets
- There are two kinds of procedural continuous assignments:

- *assign* and *deassign*:
  - for registers
- An *assign* statement overrides all procedural assignments to a register
- The *deassign* statement ends the continuous assignment to a register
- Value remains until assigned again

- *force* and *release*:
  - for net
- Similar to *assign-deassign*, except that it can be applied to nets as well as registers
- *force* statement on a net overrides all drivers of the net, until a *release* is executed on the net

2-51

Two Examples

```
input D, CLR, CLK, PRE;
reg Q;
……
always @(posedge CLK)
    Q = D;

always @(CLR or PRE)
    if (!CLR)
        assign Q = 0;
        // D has no effect on Q
    else if (!PRE)
        assign Q = 1;
        // D has no effect on Q
    else
        deassign Q;
        // Q can have D's value
```

```
wire PRT, STD, DXZ;
……
or #1 (PRT,  STD, DXZ);

initial begin
    force PRT = DXZ & STD;
    #5
    release PRT;
    // wait for 5 time units
    $finish;
end

/* PRT = DXZ & STD from time 0
    to time 5. Then PRT = DXZ |
    STD after time 5. (released) */
```

2-52

Conditional Statements

- *if* and *else if* statements

```
if (expression)
   statements
{ else if (expression)
   statements        }
[ else
   statements        ]
```

```
if (total < 60) begin
   grade = C;
   total_C = total_C + 1;
end
else if (sum < 75) begin
   grade = B;
   total_B = total_B + 1;
end
else grade = A;
```

- *case* statement

```
case (case_expression)
   case_item_expression
   {, case_item_expression } :
   statements
   ……
   [ default: statements ]
endcase
```

```
case (OP_CODE)
   2`b10:     Z = A + B;
   2`b11:     Z = A – B;
   2`b01:     Z = A * B;
   2`b00:     Z = A / B;
   default:   Z = 2`bx;
endcase
```

2-53

Supplements for Case Statement

- *case_expression* is evaluated first
- A case item can have more than one *case_item_expressions* separated with a comma(**,**)
  - Any match between those case_item_expressions will execute the following statements
- All *case_item_expressions* are evaluated and compared in the order given
- *case_expression* or *case_item_expressions* need not be constant expressions
- ***x*** and ***z*** values are compared as well

2-54

Don't-Cares in Case

- *casez* considers ***z*** values (in *case-expression* and *case_item_expression*) as don't-cares
- *casex* considers ***x*** and ***z*** values as don't-cares
- A don't-care implies this bit position is not considered

```
casez (MASK)
   4`b1???:      DBUS(4) = 0;
   4`b01??:      DBUS(3) = 0;
   4`b001?:      DBUS(2) = 0;
   4`b0001:      DBUS(1) = 0;
endcase
```

- A z value is same as a ? in literals to represent a don't-care
- In the non-completely specified cases, *casex* and *casez* often result in a more efficient hardware

2-55

Loop Statements

- Four loop statements are supported
  - The *for* loop
  - The *while* loop
  - The *repeat* loop
  - The *forever* loop
- Most of the loop statements are not synthesizable in current commercial synthesizers

2-56

The *for* Loop

for (*initial_assignment*; *condition*; *step_assignment*)
statements

```
integer i;  // i must be register or integer
for (i=0; i<8; i=i+1)
begin
   $display (" i = %d", i);
end
```

- The syntax is similar to that in C language
- Nested *for* loops supported

2-57

The *while* Loop

while (*condition*)
statements

```
while (BY > 0)
begin
   ACC = ACC << 1;
   BY = BY + 1;
end
```

- The syntax is similar to that in C language
- Nested *while* loops supported

2-58

The *repeat* Loop

repeat (*loop-count*)
statements

```
repeat (5)
  $display ("test repeat loop");

repeat (SHIFT_BY)
  P_REG = P_REG << 1;
```

- Nested *repeat* loops supported

2-59

The *forever* Loop

forever
statements

```
forever
  #10 CLOCK = ~CLOCK;
```

- Should only be used with timing controls or with the **disable** statement

2-60

Timing Controls

- The execution sequence can be rescheduled by using these timing controls
- The following statements can be executed only when current timing controls are finished
- Delay control:
  - # *delay*
  - Time duration: from the time initially encountered the statement to the time it is executed
- Event control:
  - Statement execution is delayed until the occurrence of some simulation event
  - Edge-triggered control (@ …)
  - Level-sensitive control (wait …)

2-61

Delay Control

- The procedural statement execution is delayed by the specified delay
- If delay expression is ***x*** or ***z***, it will be treated as zero delay
- If delay expression is negative, its two's complement unsigned number will be used

```
#2 TX = RX -5;

#STROBE   COMPARE = TX ^ MASK;

#(PERIOD/2)   CLOCK = ~CLOCK;
```

2-62

Event Controls

- Edge-triggered control
  - Negative edge:
    (1 -> x, z, or 0)
    (x or z -> 0)
  - Positive edge:
    (0 -> x, z, or 1)
    (x or z -> 1)
  - Events can be or'ed as well to indicate "if any one of the events occur"

```
@ (posedge CLK) CS = NS;
@ (A or B) COUNT = 0;
@ CLA   ZOO = FOO;
// wait for any change of CLA
@ (posedge CLR or negedge
   RST) Q = 0;
```

- Level-sensitive control
  - Execution of a procedural statement is delayed until a condition becomes true
  - *wait* assignment:
    wait (*condition*)
    statements
  - If condition is already true, the next statement is evaluated immediately

```
wait (SUM > 22) SUM = 0;
wait (posedge CLK) Q = D;
```

2-63

Block Statements

- Grouping multiple statements as one statement
- Two types:
  - Sequential block (*begin-end* block):
    Statements are executed sequentially in the given order
  - Parallel block (*fork-join* block):
    Statements in this block are executed concurrently
- Blocks can be named optionally
  - Registers can be declared locally
  - Blocks can be referenced (for *disable* statement)
  - Can uniquely identify registers

2-64

Two Block Statements

begin
[: *block_id* {*declarations*}]
statements
end

```
// waveform generation
begin : seq_waveform
   #2 STREAM = 1;  // time 2
   #5 STREAM = 0;  // time 7
   #3 STREAM = 1;  // time 10
   #4 STREAM = 0;  // time 14
   #2 STREAM = 1;  // time 16
   #5 STREAM = 1;  // time 21
end
```

fork
[: *block_id* {*declarations*}]
statements
join

```
// waveform generation
fork
   #2 STREAM = 1;
   #7 STREAM = 0;
   #10 STREAM = 1;
   #14 STREAM = 0;
   #16 STREAM = 1;
   #21 STREAM = 1;
join
```


Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
  - Tasks and functions
  - Disable statement
  - User-defined primitives
- Simulation and test benches

Tasks and Functions

- Provide the ability to call common procedures from different places within a description
- Enable large procedures to be broken into smaller ones
  - Make reading and debugging easier

- Tasks
  - Concurrent procedures
  - Can have timing controls
  - Can call other functions and tasks
  - Can have zero or more arguments
  - No value returned
- Functions
  - Executed in one simulation time unit
  - Can't have timing controls
  - Can't call other tasks
  - Must have one or more input arguments
  - Return a single value

2-67

Tasks

- Task definition

  task task_id;
  [*declarations*]
  statements
  endtask

- Task calling

  task_id [(expr1, expr2, …, exprN)];

  - List of arguments must match the order of arguments in task definition
  - Local variables are static; if concurrently called, same local variables are shared

```
// task example
    parameter MAXBIT = 8;
    task REVERSE_BIT;
      input [MAXBIT-1 : 0] DIN;
      output [MAXBIT-1 : 0] DOUT;
      integer K;
      begin
        for (K=0; K<MAXBIT; K=K+1)
            DOUT [MAXBIT-K] = DIN[K];
      end
    endtask

// calling the task
    reg [MAXBIT-1 : 0] REG_X, NEW_X;
    REVERSE_BIT (REG_X, NEW_X);
```

2-68

Functions

- Function definition

  function [range] func_id;
  input declarations
  other declarations
  statements
  endfunction

  – If no range is specified, 1-bit is assumed
  – Output is **reg** type with same name of func_id

- Function calling

  func_id (expr1, expr2, …, exprN);

  – Can be used in any expressions

```
// function example
    parameter MAXBIT = 8;
    function [MAXBIT-1] REVERSE_BIT;
      input [MAXBIT-1 : 0] DIN;
      integer K;
      begin
        for (K=0; K<MAXBIT; K=K+1)
            REVERSE_BIT [MAXBIT-K]
            = DIN[K];
      end
    endfunction

// calling the function
    reg [MAXBIT-1 : 0] REG_X, NEW_X;
    NEW_X = REVERSE_BIT(REG_X);
```

2-69

Disable Statement

- Can be used to terminate a task or a block before it finishes executing all its statements
  disable *task_id*;
  disable *block_id*;
- Used to model hardware interrupts and global resets
  – Similar to the “break” statement in C language
- Execution continues with the next statement

```
begin: BLK_A
    // stmt 1
    // stmt 2
    disable BLK_A;
    // stmt 3      } never executed
    // stmt 4      }
end
// stmt 5
        ↑
        └── continue from here
```

```
task BIT_TASK;
    ……
    disable BIT_TASK;
    ……
endtask

/* when disable statement is
    executed, the task is aborted */
```

2-70

User-Defined Primitives (UDP)

- Usage is exactly the same as gate primitives
- May be a combinational UDP or a sequential UDP
- A sequential UDP may model both level-sensitive and edge-sensitive behavior
- Behavior is described as a table
- Each UDP has one output: 0, 1, x (z is not allowed)
- If input is z, output becomes an x
- In sequential UDP, output has the same value as the internal state

2-71

Combinational UDP

```
primitive MUX1BIT (Z, A, B, SEL);
   output Z;
   input A, B, SEL;
   table
     // A     B     SEL   :   Z
        0     ?     1     :   0     ;
        1     ?     1     :   1     ;
        ?     0     0     :   0     ;
        ?     1     0     :   1     ;
        0     0     x     :   0     ;
        1     1     x     :   1     ;
   endtable
endprimitive
```

Don't care

- Any combination that is not specified is an x
- Output port must be the first port
- "?" represents any one of "0", "1", "x" logic values

2-72

Level-Sensitive Sequential UDP

```
primitive LATCH (Q, CLK, D);
   output Q;
   reg Q;
   input CLK, D;
   table
   // clock   data    :       state   :       output (next_state)
       0       1       :       ?       :       1       ;
       0       0       :       ?       :       0       ;
       1       ?       :       ?       :       -       ;
   endtable
endprimitive
```

- "?" means don't care
- "-" means no change in the output

2-73

Edge-Sensitive Sequential UDP

```
primitive D_EDGE_FF (Q, CLK, DATA);
   output Q;
   reg Q;
   input CLK, DATA;
   table
   // CLK     data    :       state   :       next (Q)
     (01)      0       :       ?       :       0       ;
     (01)      1       :       ?       :       1       ;
     (0x)      1       :       1       :       1       ;
     (0x)      0       :       0       :       0       ;
      // ignore negative edge of clock
     (?0)      ?       :       ?       :       -       ;
      // ignore data change on steady clock
      ?        (??)    :       ?       :       -       ;
   endtable
endprimitive
```

2-74

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Other topics
- Simulation and test bench

2-75

Simulation

- Design, stimulus, control, saving responses, and verification can be completed in a single language
  - Stimulus and control
    - Use initial procedural block
  - Saving responses
    - Save on change
    - Display data
  - Verification
    - Automatic compares with expected responses
- The behavior of a design can be simulated by HDL simulators
  - Test benches are given by users as the inputs of the design
  - Some popular simulators
    - Verilog-XL (Cadence™, direct-translate simulator)
    - NC-Verilog (Cadence™, compiled-code simulator)
    - VCS (ViewLogic™, compiled-code simulator)

2-76

Time Unit and Precision

- Compiler directive: **`timescale**
  `timescale *time_unit* / *time_precision*
- Directive must appear outside a module definition
- 1, 10, 100 / s, ms, us, ns, ps, fs

```
`timescale 1ns / 100ps
module AND_FUNC (Z, A, B);
  output Z;
  input A, B;
  and #(5.22, 6.17) A1(Z, A, B);
endmodule

/* Delays are in ns. Delays are rounded to 100ps (0.1ns).
   Therefore, 5.22 becomes 5.2ns, 6.17 becomes 6.2ns */

`timescale 10ns / 1ns
/* then 5.22 becomes 52ns, 6.17 becomes 62ns */
```

2-77

Supports for Verification

- Getting simulation time
  - **$time** (64-bit integer), **$stime** (32-bit integer), **$realtime** (real number)
  - The returned value is scaled to the time unit of the module that invoke it

```
`timescale 10ns / 1ns
module TB;
  ……
  initial
        $monitor("A=%d B=%d", A, B, "at time %t",$time);
endmodule

A=0 B=0 at time 0
A=0 B=1 at time 5
A=0 B=0 at time 16
……
/* $time value is scaled to the time unit and then rounded */
```

2-78

Supports for Verification

- Text output (show results at standard output)
  - **$display**: print out the current values of selected signals
    - Similar to the printf() function in C language
  - **$write**: similar to **$display** but it does not print a "\n"
  - **$monitor**: display the values of the signals in the argument list whenever any signal changes its value
  - Examples:

```
$display ("A=%d at time %t", A, $time);
    A=5 at time 10

$monitor ("A=%d CLK=%b at time %t", A, CLK, $time);
    A=2 CLK=0 at time 0
    A=3 CLK=1 at time 5
    ……
```


Supports for Verification

- File output (results are stored in a specified file)

  <*multi_channel_descriptor* > = **$fopen** ("<*file_name*>");
  $fclose (<*multi_channel_descriptor* >);
  $fdisplay (<*multi_channel_descriptor* >, P1, P2, …, Pn);
  $fwrite (<*multi_channel_descriptor* >, P1, P2, …, Pn);
  $fmonitor (<*multi_channel_descriptor* >, P1, P2, …, Pn);
- Value change dump (VCD) file
  - Store every value changes of selected signals in a file with a special format
    - Changing at what time and to which value
  - Widely used for post-processing (ex: waveform viewer)

    $dumpfile ("<*file_name*>");
    /* default name is verilog.dump */
    $dumpvars (*level_num*, *signal_list*);
    $dumpvars; // dump all variables

```
# time1
  <new_value><variable1>
  <new_value><variable2>
        ⋮
# time2
  <new_value><variable1>
  <new_value><variable2>
        ⋮
```

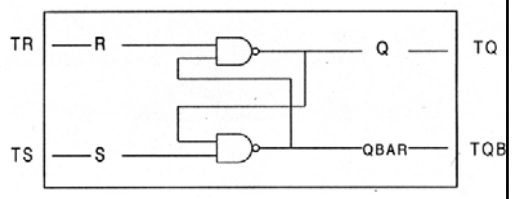
Test Bench

module test_bench;
data type declaration
module instantiation
applying stimulus
display results
endmodule

- A test bench is a top level module without inputs and outputs
- Data type declaration
  - Declare storage elements to store the test patterns
- Module instantiation
  - Instantiate pre-defined modules in current scope
  - Connect their I/O ports to other devices
- Applying stimulus
  - Describe stimulus by behavior modeling
- Display results
  - By text output, graphic output, or waveform display tools

An Example

```
module RS_FF (Q, QBAR, R, S);
  output  Q, QBAR;
  input  R, S;
          1 time-unit(ns) delay
  nand  #1 (Q, R, QBAR);
  nand  #1 (QBAR, S, Q);
endmodule
```



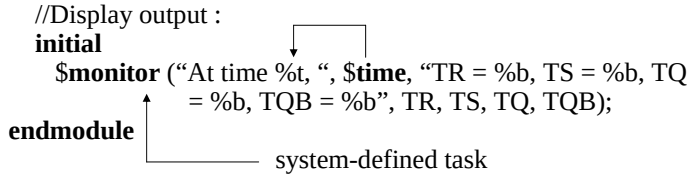

```
module test;
  reg  TS, TR;

  //Instantiate module under test :
  RS_FF INST_A (.Q(TQ), .QBAR(TQB), .S(TS), .R(TR));
    //Using named association.
                          connect TQ to Q
```

An Example (cont.)

```
//Apply stimulus;
initial begin
   TR = 0; TS = 0;
   #5 TS = 1;
   #5 TS = 0; TR = 1;
   #5 TS = 1; TR = 0;
   #5 TS = 0;
   #5 TR = 1;
   #5 $stop;
 end

//Display output :
initial
  $monitor ("At time %t, ", $time, "TR = %b, TS = %b, TQ
           = %b, TQB = %b", TR, TS, TQ, TQB);
endmodule
```



2-83

Another Example

```
module Add4 (S, Co, A, B, Ci);
  output [3:0]  S;
  output  Co;
  input [3:0]  A, B;
  input  Ci;

  assign {Co, S} = A + B + Ci;
endmodule

module test_FA;
  integer  A, B;
  reg  Ci;
  wire [3:0]  Sum;
  wire  Co;

  //Instantiate module under test :
  Add4 U1(Sum, Co, A[3:0],
          B[3:0], Ci);

  //Apply exhaustive patterns
  initial begin
    for (A=0; A<=15; A=A+1)
      for (B=0; B<=15; B=B+1)
      begin
        Ci = 0;
        #1 $display ("A=%d
             B=%d Ci=%b
             SUM=%d Co=%b",
             A, B, Ci, SUM, Co);
        Ci = 1;
        #1 $display ("A=%d
             B=%d Ci=%b
             SUM=%d Co=%b",
             A, B, Ci, SUM, Co);
      end
  end
endmodule
```

2-84