VERILOG

COMPILER DIRECTIVES AND MODELLING DELAYS

ROHIT KHANNA

Compiler Directives

- They are special commands used to control compilation of verilog codes.
- All compiler directives are defined by using `keyword construct.
- They are effective from the point they are declared to the point they are overridden by other directive.
- Compiler directive can be declared anywhere in the verilog code but it is recommended to used them outside the module.

`define

- `define is used to define Macros for text substitution in Verilog.
- It is similar to #define in C.
- This directive can be used both inside and outside module definitions.
- Verilog compiler will substitute text of macro wherever it encounters `macro_name
- A macro text could be any arbitrary text specified on the same line.

`define

- If more than one line is required to define macro text , \ (backslash) should be placed at the start of new line.

Declaration Examples:

```
`define size 4
`define sign signed
`define exit $stop
```

Usage Examples:

```
reg  `sign [`size -1 : 0 ] temp;
initial   #5000 `exit;
```

`include

- `include is used to insert entire content of source file in a file during compilation.
- The compiler assumes that content of included source file appears in place of `include compiler directive.
- `include can be used to include globally used function, tasks, and Macros without enclosing them inside module and endmodule.

Syntax:

```
`include "file_name.v"                    // relative path
`include "c:/folder/src/file_name"        // absolute path
```

`undefine

- `undefine is used to remove definition of previously defined Macro.

- An attempt to undefine a text macro which was not previously defined will issue a warning.

Example:

```
`define size 10
 reg [`size -1 : 0 ] temp;
 `undefine size
```

`include- Example

max.v

```
function integer max;
input integer a, b, c;
    if (a> b && a> c )
      max=a;
  else
    if (b>c)
      max=b;
    else
      max=c;
endfunction
```

`include- Example

test.v

```
`include "max.v"
//Verilog 2009 (SV)
module test (a, b, c, op);
input signed [31:0] a, b, c;
output signed [31:0] op;

assign op=max(a, b, c);

endmodule
```

```
module test (a, b, c, op);
input signed [31:0] a, b, c;
output signed [31:0] op;
//Verilog all versions
`include "max.v"

assign op=max(a, b, c);

endmodule
```

`timescale

- The `timescale compiler directive specifies the time unit and precision of the modules that follow the directive.

  Syntax:

  `timescale time_unit/time_precision

- time_unit is the units of measurement for time values, such as simulation time and delay values.

- time_precision is the minimum value(least count) for time. All time values are rounded off to the time_precision values.

`timescale

- time precision value should be less than time unit.
- integers are used to provide magnitude of time unit and time precision. 1, 10, and 100 are only valid integer for timescale specification.
- s (second), ms (millisecond), us (microsecond), ns (nanosecond), ps (picosecond), and fs (femtosecond) are possible units for time unit and time precision.
- If no `timescale is specified, the default values of time unit and precision are used which are tool-specific.

`timescale

```
`timescale  10 ns / 1 ns                    // Time scale declaration

// According to definition
Time Units= 10 ns
Time Precision= 1 ns

//Time and Delay Calculation
#5 is equivalent to 5 * 10 ns = 50 ns
```

If result comes in decimal, it is rounded off to the nearest value with respect to Time Precision.

Example

```
`timescale  10 ns / 1 ns                    // Time scale declaration

module time_test;
integer a=0, b=0, c=0;
initial
begin
#1.53 a=6;
#2.56 b=9;
#1.547 c=4;
end
endmodule
```

a= 6 occurs at 15 ns
b= 9 occurs at 41 ns
c= 4 occurs at 56 ns

$printtimescale

- $printtimescale system task is used to display the time unit and precision for a module.

Syntax:

$printtimescale;

Prints Time Scale of the current module

$printtimescale (module_label);

Prints Time Scale of the module label passed as a parameter.

Hierarchal referencing can also be used.

Example

```
`timescale  10 ns / 1 ps                    // Time scale declaration

module time_test;
parameter d=1.543;
integer a=0;
initial
begin
#d a=6;
$printtimescale;
end
endmodule
```

```
a= 6 occurs at 15430 ps
Time scale of (time_test) is  10ns /  1ps
```

Example

```
`timescale  10 ns / 1 ps                    // Time scale declaration
 module abc;
 xyz u0 ();
 initial
 $printtimescale(u0);
 endmodule
```

Time scale of (abc.u0) is 1ms / 10us

```
`timescale  1ms / 10 us                     // Time scale declaration
 module xyz;
 endmodule
```

Conditional Compilation

- Conditional Compilation is used by a designed to specify what part of code will be compiled based on declaration of certain macros.
- It is accomplished by using compiler directives `ifdef, `ifndef, `else, `elsif and `endif.
- These directives may appear anywhere in the source description.
- Nesting of these compiler directives is allowed.
- `else directive is optional, `ifdef or `ifndef is always closed by a corresponding `endif.

Example1

```
`define AND
module logic_gate (input a, b, output c);
 `ifdef AND
   assign c = a & b;
 `elsif OR
   or (c, a, b);
  `else
   assign c = a ^ b;
  `endif
endmodule
```

Result:

and gate

Example2

```
`define and
module logic_gate (input a, b, output c);
 `ifdef AND
   assign c = a & b;
 `elsif OR
   or (c, a, b);
  `else
   assign c = a ^ b;
  `endif
endmodule
```

Result:

xor gate

Example3

```
`define upcount

module counter (input clk, output integer count=0);

  always @ (posedge clk)
  `ifndef upcount
   count<= count + 1;
   `else
   count<= count – 1;
  `endif
 endmodule
```

Result:

down counter

Example4

```
`define upcount

`ifdef size
module logic_gate (input a, b, output c);
assign c = a | b;
endmodule
`else
module counter (input clk, output integer count=0);
always @ (posedge clk) count<= count + 1;
endmodule
`endif
```

Result:

up counter

Conditional Execution

- Conditional Execution is used by a designed to control the execution of code at run time.
- All statements are compiled by are executed conditionally.
- $test$plusargs and $value$plusargs system function is used for conditional execution.

Syntax:

$test$plugargs("string")

$value$plugargs("user_string", value)

$test$plusargs

- This system function search the plusargs for the provided string.
- The plusargs arguments present in the command line are compared with the string.
- If prefix of one of the plusagrs matches all characters in provided string, a non-zero integer is returned.
- If no plusargs present in command line match with provided string, integer value zero is returned.

Example1

```
module sample;
initial
begin
if ($test$plusargs("HELLO")) $display("Hello executed");
if ($test$plusargs("HI")) $display("Hi executed");
if ($test$plusargs("HELP")) $display("Help executed");
end
endmodule
```

Transcript

```
vsim –novopt sample +HELLO
> Hello Executed
```

Example2

```
module sample;
initial
begin
if ($test$plusargs("HELLO")) $display("Hello executed");
if ($test$plusargs("HE")) $display("He executed");
if ($test$plusargs("HEL")) $display("Hel executed");
end
endmodule
```

Transcript

```
vsim –novopt sample +HELL
>He executed
>Hel executed
```

$value$plusargs

Syntax :

$value$plusargs("<string>=%<format>", program_variable)

- Each plusargs is compared with the specified strings.
- If plusargs matches with the string, then the value given to that plusargs is assigned to the program variable
- If string is found, a non-zero integer is returned.
- If string is not found, integer value zero is returned and program variable is not modified.

Example

```
module sample;
integer dec ; reg [8*6:1] str;
initial
begin
if ($value$plusargs("HELLO=%d", dec))
  $display("Hello executed dec=%d", dec);
if ($value$plusargs("HE=%s", str))
   $display("He executed str=%s", str);
end
endmodule
```

Result

Transcript

vsim –novopt sample +HELLO=9

> Hello executed dec = 9

vsim –novopt sample +HE=abc

> He executed str= abc

vsim –novopt sample +HELLO=3 +HE=xyz

> Hello executed dec = 3

> He executed str= xyz

Procedural Continuous Assignments

- These are procedural statements which are used to drive register or net continuously by an expression for a specified time.
- They are used to override existing assignment to a register or net.
- They are extension of procedural assignment statements
- There are two types of procedural continuous assignment
  - ❖ assign and deassign
  - ❖ force and release

assign and deassign

- assign and deassign keywords are used for this type of procedural continuous assignment
- Left hand side of this procedural continuous assignment can only be a register.
- These statements override effect of regular procedural assignment.
- Usage of assign and deassign construct are considered as bad coding style, may not be supported by synthesis tools.

Example

D Flip-Flop with asynchronous reset

```
module dff (input din, rst, clk, output reg q);

  always @ (posedge clk)
  q<=din;

  always @ (reset)
     if (reset)
        assign q=1'b0;
     else
        deassign q;                    //Next assignment at posedge clk
endmodule
```

force and release

- force and release keywords are used for this type of procedural continuous assignment
- Left hand side of this procedural continuous assignment can be register or net .
- These are typically used for interactive debugging. Register or net are forced to a value for certain time.
- It is recommended not to used force and release for design codes. They should only appear in test benches.

Example1

Design Code

```
module dff (din, clk, q);
input din, clk;
output reg q;

  always @ (posedge clk)
  q<=din;

endmodule
```

Test Bench

```
module dff_test;
reg din, clk=1; wire q;

dff u0 (din, clk, q);

always #5 clk=~clk;

initial begin
force u0.q=0;
#9 release u0.q;
forever #10 din=$random; end
endmodule
```

Example2

Design Code

```
module expr (a, b, c, d);
input a, b, c;
output d;
assign d= a & b ^ c;

endmodule
```

Test Bench

```
`define r $random
module expr_test;
reg a, b, c; wire d;

expr u0 (a, b, c, d);

initial begin
force u0.d=0;        //initialize net
#0 release u0.q;
forever begin #5 a=`r, b=`r c=`r; end
endmodule
```

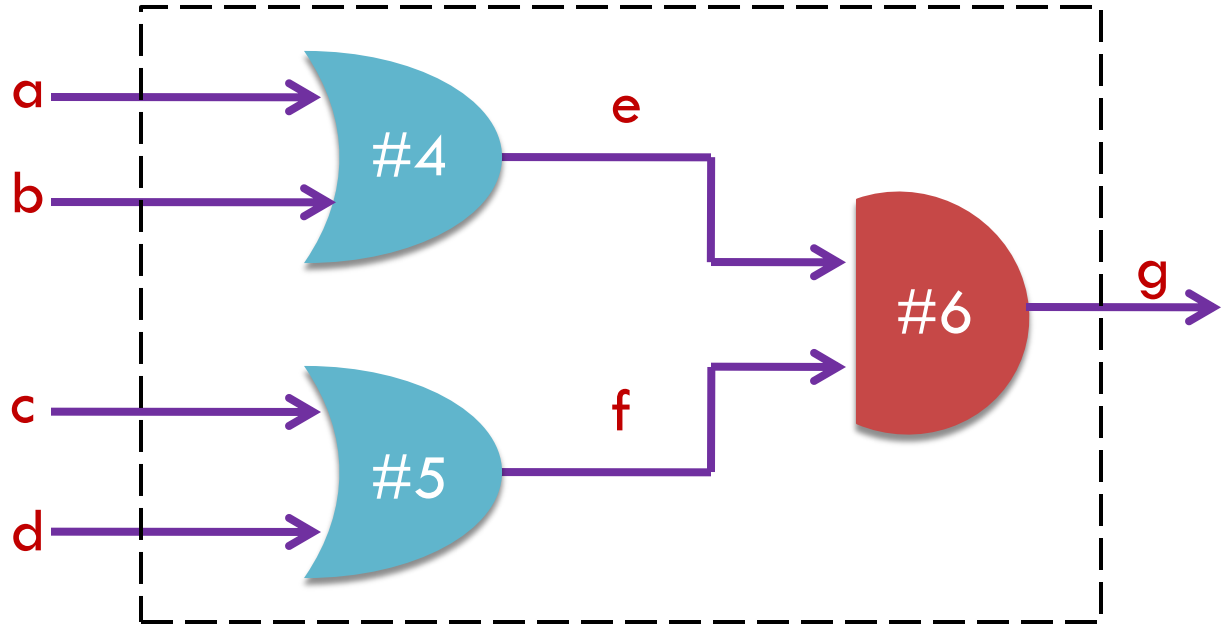
Modeling Delays

Verilog provides three ways for modeling delays

- distributed delays
- lumped delays
- pin to pin delays

Distributed Delay

- Delays are assigned individually to each and every element in the circuit.



Example1

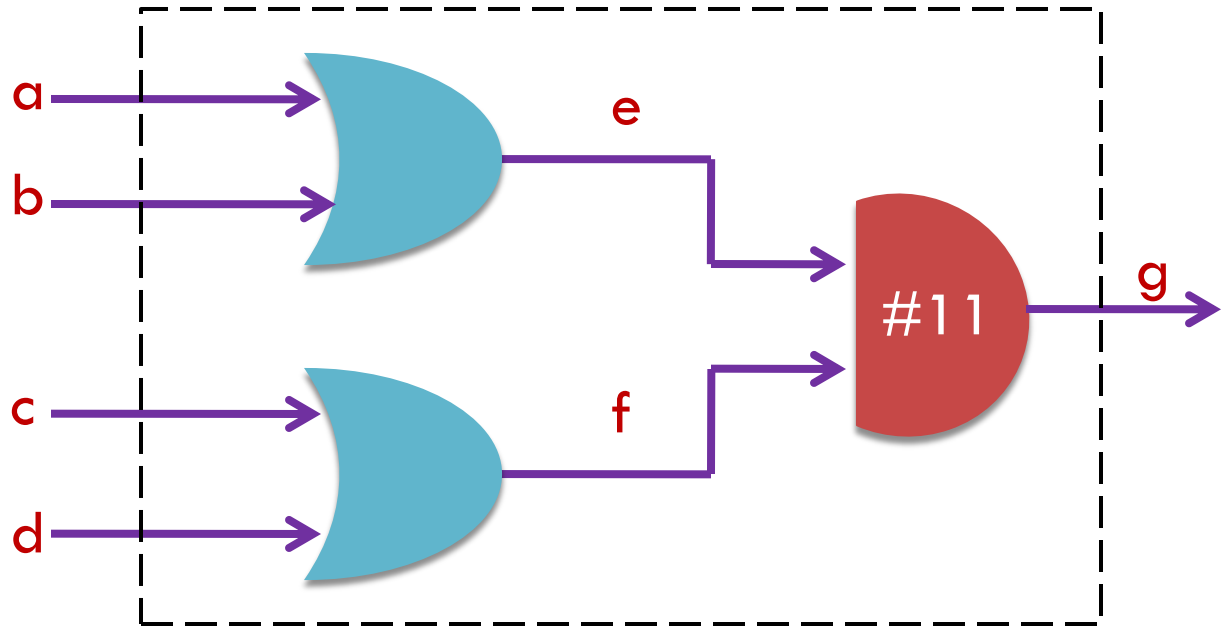
```
module design (a, b, c, d, g);
input a, b, c, d;
output g;

wire e, f;

or #4 (e, a, b);
or #5 (f, c, d);
and #6 (g, e, f);

endmodule
```

Example2

```
module design (a, b, c, d, g);
input a, b, c, d;
output g;

wire e, f;

assign #4 e=a | b;
assign #5 f =c | d;
assign #6 g =e & f;

endmodule
```

Lumped Delay

- Lumped Delays are used to specify Delay for the entire module.
- The cumulative delay of all path is specified as single delay on the output gate of the module.



Example1

```
module design (a, b, c, d, g);
input a, b, c, d;
output g;

wire e, f;

or  (e, a, b);
or  (f, c, d);
and #11 (g, e, f);

endmodule
```

Example2

```
module design (a, b, c, d, g);
input a, b, c, d;
output g;

wire e, f;

assign  e=a | b;
assign  f =c | d;
assign #11 g =e & f;

endmodule
```

Features

- Distributed Delays
  - Difficult to implement
  - Delays are accurate
- Lumped Delays
  - Most easy to implement
  - Delays are less accurate
  - All path have same amount of delay

Pin to Pin Delay

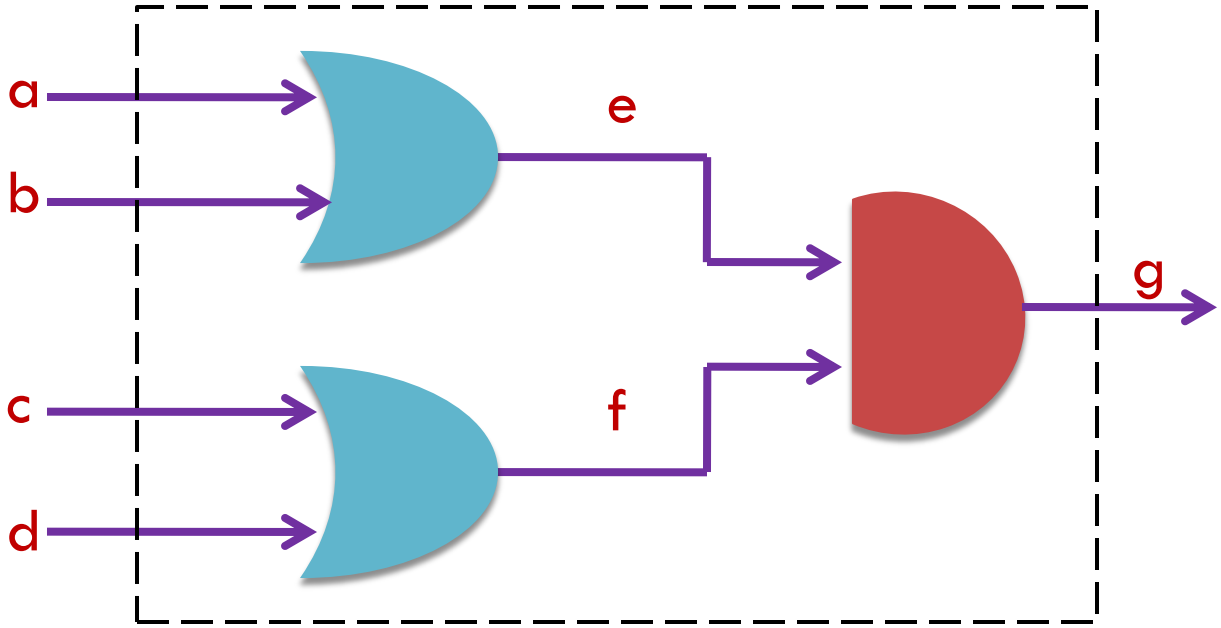
- Used to define delay from an input pin to an output pin.
- It is possible to specify same delay or individual delay for all or each input pin to output pin.
- They are also called as path delays.
- For bigger circuits this delay model is easier to implement as compared to distributed delay model.
- Specify blocks are used to provide pin to pin delays.

Specify Block

- Bounded by specify and endspecify keyword.
- It should be declared inside a module outside procedural blocks (always or initial).
- Specify block can be used for the following:
  - Describing various paths in a module and assigning delay to those paths.
  - Performing Timing Checks (example: Setup, Hold)
  - Defining parameters local to specify block using specparam keyword

Pin to Pin Delay

Example:



Delays:

a to g =10

b to g =10

c to g =11

d to g =11

Module Path Connection

- There are two ways to describe timing paths
  - Parallel Connection
  - Full Connection

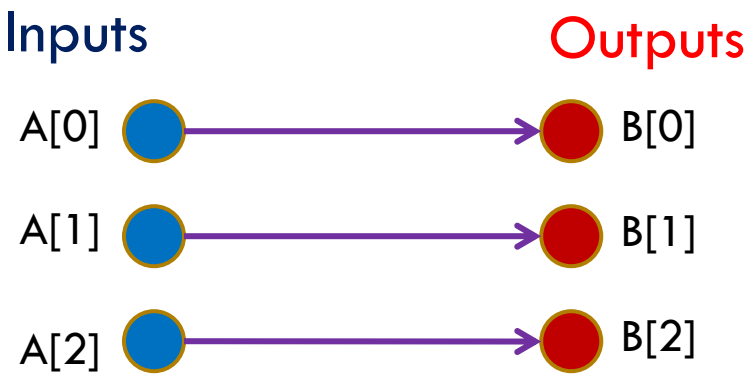
Parallel Connection

Syntax:

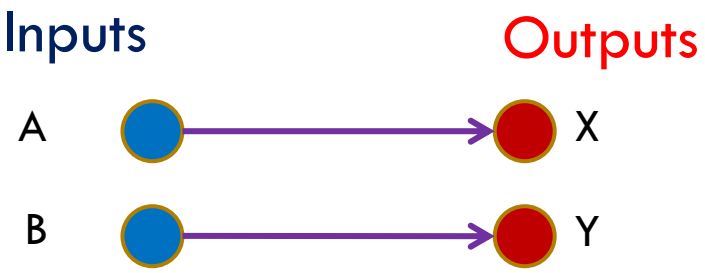
```
(source_pin => destination_pin) = delay_value;
```

- Each bit of source connects to corresponding bit of destination.
- Parallel Connection can be created between source and destination only if they are of same size.
- They only connect one source to one destination.

Parallel Connection



Number of connections are 3



Number of connections are 2

Parallel Connection

Examples:

```
input X; input [1:0] A; input [2:0] C;
output Y; output [1:0] B; output [1:0] D;

(X=>Y) = 7;                    //X and Y are scalar

(A[0]=>B[0])=3;                //A and B are 2-bit each
(A[1]=>B[1])=3;                //Delay values can be different

(A=>B)=3;                      //Alternative
(C=>D)=7;                      //Incorrect
```

Example

```
module design ( input a, b, c, d, output g);

wire e, f;

or  (e, a, b);
or  (f, c, d);
and (g, e, f);

// Specify block
endmodule
```

```
specify
(a => g) = 10;
(b => g) = 10;
(c => g) = 11;
(d => g) = 11;
endspecify
```

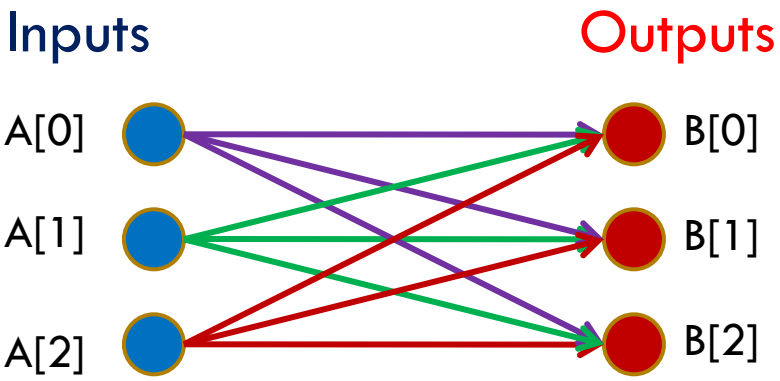
Full Connection

Syntax:

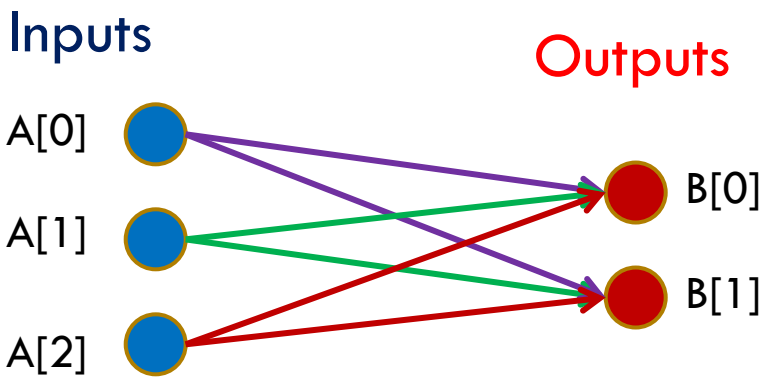
(source_pins *> destination_pins) = delay_value;

- Each bit of source connects to every bit of destination.
- Source and destination need not be of same size.
- They can be used to describe path between:
  - a vector and a scalar
  - vectors of different sizes
  - multiple sources or multiple destinations

Full Connection



Number of connections are
3*3 =9



Number of connections are
3*2 =6

Full Connection

Examples:

```
input X, Y; input [1:0] A;
output Z, B;

(X, Y*>Z)= 7;          //X and Y are scalar
(X*>Z)=7;              //Alternative
(Y*>Z)=7;

(A*>B)=3;              //A is 2-bit and B is scalar

(A[0]*>B)=3;           //Alternative
(A[1]*>B)=3;
```

Full Connection

Examples:

```
input X, Y; input [1:0] A;
output Z, B;

(X, Y*>Z, B) = 7;          //X and Y are scalar

(X*>Z)=7;                  //Alternative 2 * 2=4 connections
(Y*>Z)=7;
(X*>B)=7;
(Y*>B)=7;
```

Full Connection

Examples:

```
input [2:0] A;  output [1:0] B;

(A*>B) = 7;              //A is 3 bits and B is 2 bits

(A[0]*>B[0])=7;          //Alternative 3*2=6 connections
(A[1]*>B[0])=7;
(A[2]*>B[0])=7;
(A[0]*>B[1])=7;
(A[1]*>B[1])=7;
(A[2]*>B[1])=7;
```

Example

```
module design ( input a, b, c, d, output g);

wire e, f;

or  (e, a, b);
or  (f, c, d);
and (g, e, f);

// Specify block
endmodule
```

```
specify
(a, b*> g) = 10;
(c, d*> g) = 11;
endspecify
```

specparam

- specparam keyword is used to declare special parameters inside specify blocks.
- They are used to specify pin to pin delay values and then these parameters are used instead of hardcoded delay values.
- These parameters remain local to a specify block.

Syntax :

```
specparam parameter_name = delay_value;
```

Example

```
module design ( input a, b, c, output out);

assign out= (a & b) | c;

specify
specparam aout=2, bout=4, cout=3;
(a=>out)=aout;
(b=>out)=bout;
(c=>out)=cout;
endspecify

endmodule
```

Condition Path Delays

- These are also called as state dependent path delays.
- They are expressed with if conditional statement, else cannot be used. If multiple statements true then least delay will be used.
- The condition can contain bitwise, logical, concatenation, conditional or reduction operators.
- If condition evaluated to X and Z then it is considered as true.
- If condition evaluates to multiple bits, in that case LSB is the considered as result of the condition

Example1

```
module design ( input a, b, output out);

assign out= a & b;

specify
if (a) (a=>out)=4;
if (~a)(a=>out)=3;
endspecify

endmodule
```

Example2

```
module design ( input a, b, output out);

assign out= a & b;

specify
if (a & b) (b=>out)=4;
if (~(a & b))(b=>out)=3;
endspecify

endmodule
```

Example3

```
module design ( input a, b, c output  out);

assign out= a ^ b;

specify
if ( {a, b}== 2'b00)  (c=>out)=4;
if ( {a, b} != 2'b00)   (c=>out)=1;
endspecify

endmodule
```

Rise, Fall and Turn-off delay

- Pin to pin delays can also be expressed in terms of rise, fall and turn-off delay values.
- User can provide one, two, three, six or twelve delays.
- Other number of delay specifications are invalid.

Syntax: (source_pin=>destination_pin)= (Rise, Fall, Turn-off);

- Transition form X to known state takes maximum possible time out of all unknown (0, 1, Z) to a particular known state.
- Transition from known state to X takes minimum possible time out of all particular known state to unknown(0, 1, Z).

Example

```
specify
(a=>b)= (3);          //3 is use for all transactions
endspecify
```

```
specify
(a=>b)= (3, 4);
endspecify
```

0→1, Z→1	3 (Rise Value)
1→0, Z→0	4 (Fall Value)
0→Z	3 (Rise Value)
1→Z	4 (Fall Value)

Example

```
specify
(a=>b)= (3, 4, 2);
endspecify
```

0→1, Z→1	3 (Rise Value)
1→0, Z→0	4 (Fall Value)
0→Z, 1→Z	2 (Turn-off Value)

```
specify
(a=>b)= (3, 4, 2, 1, 5, 6);
endspecify
```

0→1	3 (Rise Value)
1→0	4 (Fall Value)
0→Z	2
Z→1	1
1→Z	5
Z→0	6

Example

```
specify
(a=>b)= (3, 4, 2, 1, 5, 6, 7, 8, 9, 10, 11, 12);
endspecify
```

0→1	3
1→0	4
0→Z	2
Z→1	1
1→Z	5
Z→0	6
0→X	7
X→1	8
1→X	9
X→0	10
X→Z	11
Z→X	12

Transitions for X

X to known

X→ 0

maximum(1→0, Z→0)

X→ 1

maximum(0→1, Z→1)

X→ Z

maximum(0→Z, 1→Z)

Known to X

0→ X

minimum(0→1, 0→Z)

1→ X

minimum(1→0, 1→Z)

Z→ X

minimum(Z→0, Z→1)

Min, Max and Typical Delays

- Min, Max and Typical delay can also be specified for pin to pin delays.

Syntax:

Min: Typical : Max

Example:

```
specparam rise=1:2:3, fall=2:4:5, turn-off=3:5:7;
(a=>b)=(rise, fall, turn-off);
```

Event Sensitive Paths

- Edge sensitive timing construct is used to model input to output delay which occurs only when a specified edge occurs on signal.

Syntax:

([posedge | negedge] edge_pin=> (output_pin : input_pin))
=(rise_time, fall_time);

Example1

```
module dff (input clk, din, output reg dout);

always @ (posedge clk)
dout<=din;

specify
(posedge clk => (dout : din))=(1, 2);        //(rise_time, fall_time)
endspecify

endmodule
```

Example2

```
module dff (input clk, din, output reg dout);

always @ (negedge clk)
dout<=din;

specify
(nedgede clk => (dout : din))=(1, 2);        //(rise_time, fall_time)
endspecify

endmodule
```