
Introduction to Verilog

Course Objectives

- Learn the basic constructs of Verilog
- Learn the modeling structure of Verilog
- Learn the concept of delays and their effects in simulation

Course Outline

- Verilog Overview
- Basic Structure of a Verilog Model
- Components of a Verilog Module
 - Ports
 - Data Types
 - Assigning Values and Numbers
 - Operators
 - Behavioral Modeling
 - Continuous Assignments
 - Procedural Blocks
 - Structural Modeling
- Summary: Verilog Environment

Verilog Overview

What is Verilog?

- IEEE industry standard Hardware Description Language (HDL) - used to describe a digital system
- For both Simulation & Synthesis

Verilog History

- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence purchased Gateway (Verilog-XL simulator)
- 1990 Cadence released Verilog to the public
- **Open Verilog International (OVI)** was formed to control the language specifications.
- 1993 OVI released version 2.0
- 1993 IEEE accepted OVI Verilog as a standard, Verilog 1364

Verilog Structure

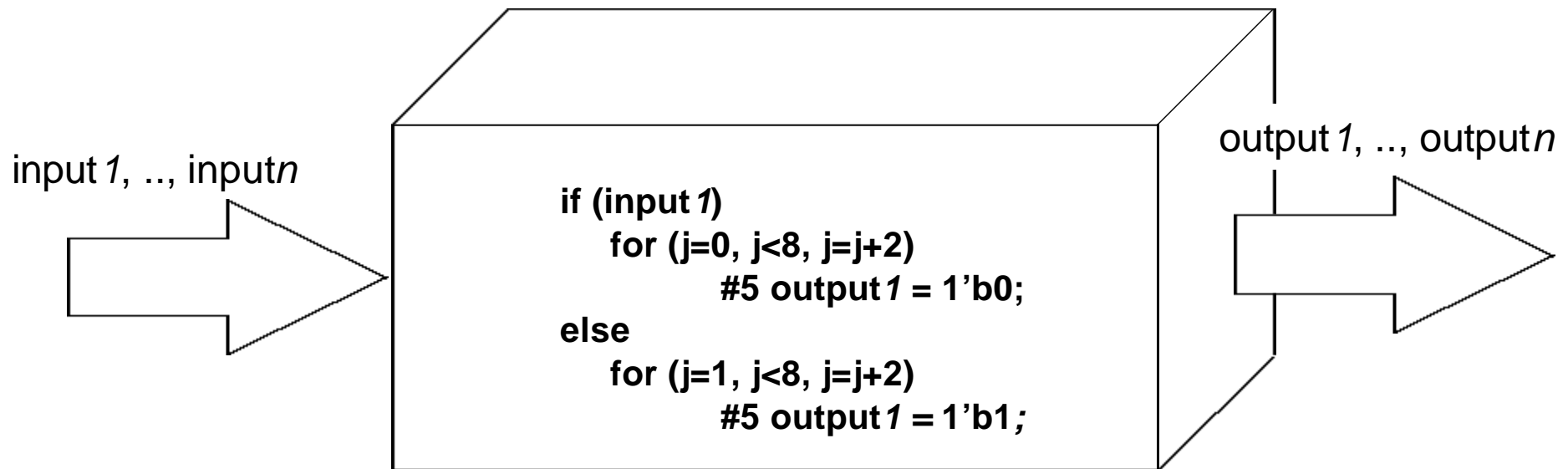
- Verilog HDL : Consists of Keywords, syntax and semantics used to describe hardware functionality and timing.
- PLI : **P**rogramming **L**anguage **I**nterface provides C language routines used to interact between Verilog and EDA tools. (Simulators, Waveform displays)
- SDF : **S**tandard **D**elay **F**ormat - a file used to back-annotate accurate timing information to simulators and other tools.

Terminology

- HDL - Hardware Description Language is a software programming language that is used to model a piece of hardware
- Behavior Modeling - A component is described by its input/output response
- Structural Modeling - A component is described by interconnecting lower-level components/primitives

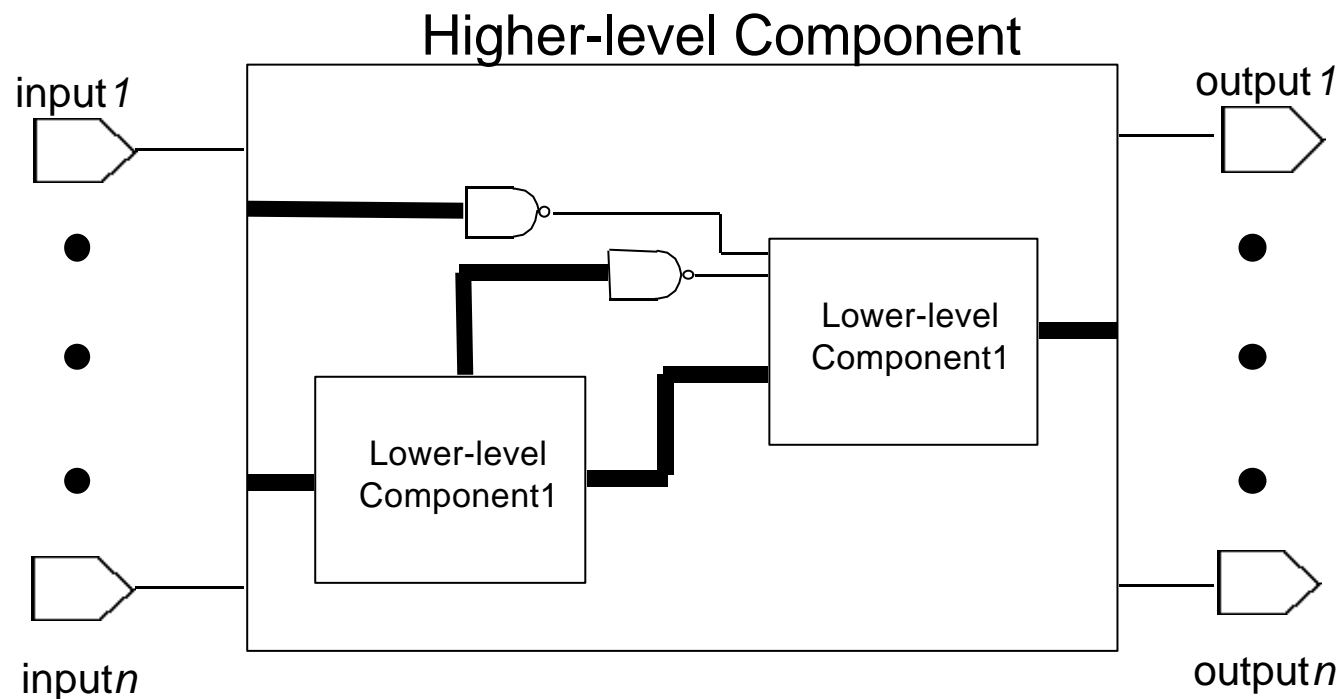
Behavior Modeling

- Only the functionality of the circuit, no structure
- No specific hardware intent
- For the purpose of synthesis, as well as simulation



Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware
- For the purpose of synthesis



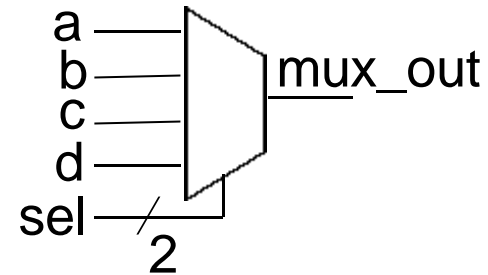
More Terminology

- Register Transfer Level (RTL) - A type of behavioral modeling, for the purpose of synthesis.
 - Hardware is implied or inferred
 - Synthesizable
- Synthesis - Translating HDL to a circuit and then optimizing the represented circuit
- RTL Synthesis - The process of translating a RTL model of hardware into an optimized technology specific gate level implementation

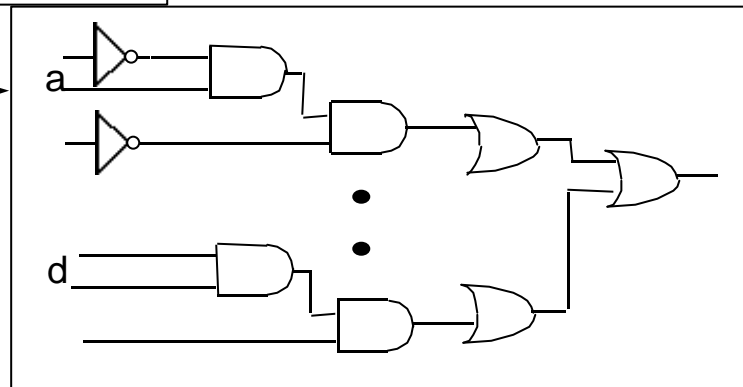
RTL Synthesis

```
always @(a or b or c or d or sel)
begin
  case (sel)
    2'b00: mux_out = a;
    2'b01: mux_out = b;
    2'b10: mux_out = c;
    2'b11: mux_out = d;
  endcase
end
```

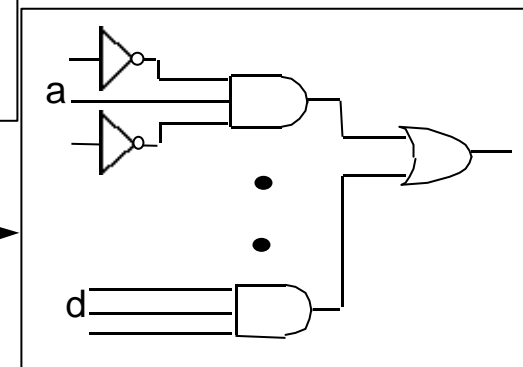
inferred



Translation



Optimization



Verilog vs. Other HDL Standards

- Verilog

- “Tell me how your circuit should behave and I will give you the hardware that does the job.”

- VHDL

- Similar to Verilog

- ABEL, PALASM, AHDL

- “Tell me what hardware you want and I will give it to you”

Verilog vs. Other HDL Standards

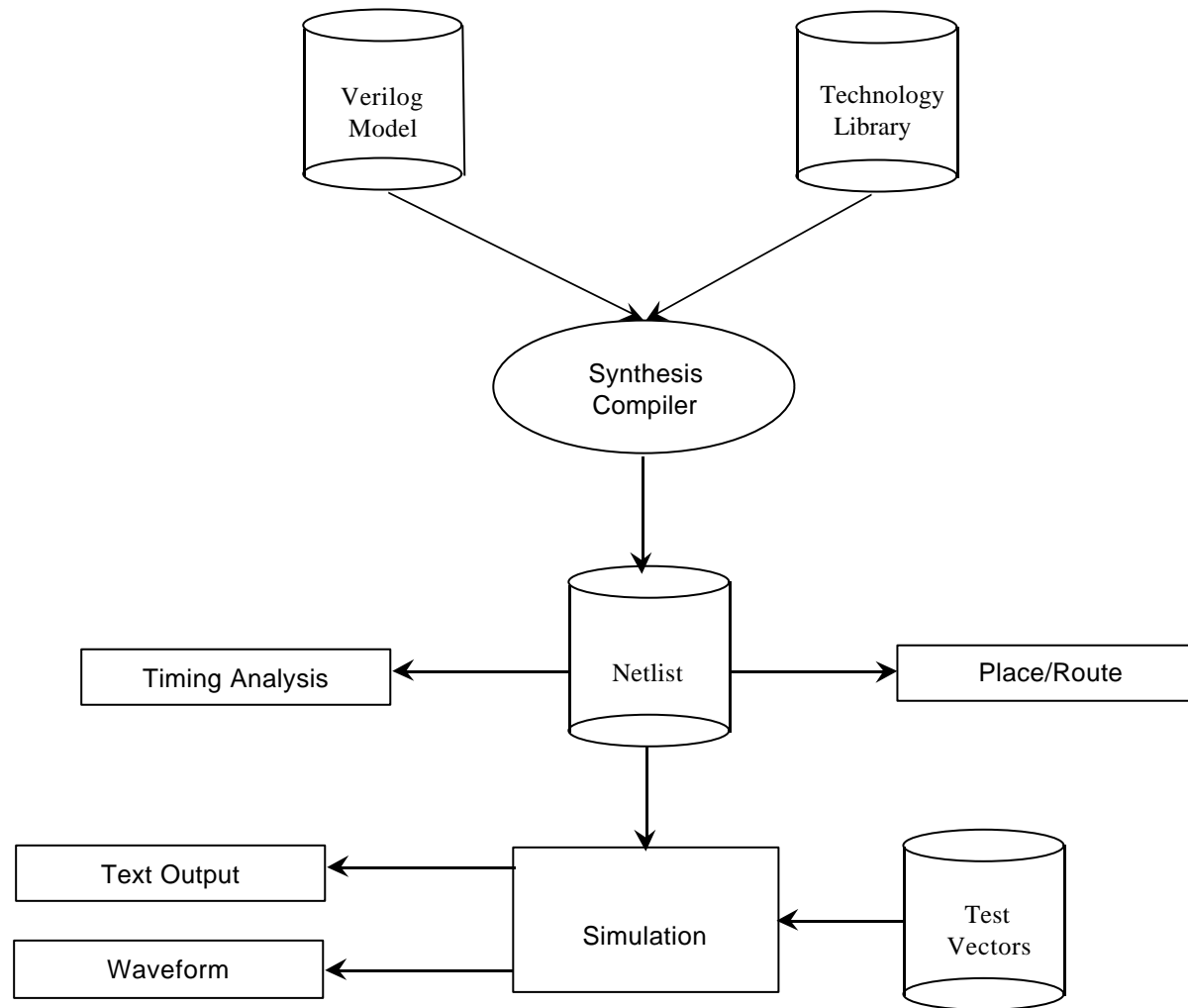
■ Verilog

- “Give me a circuit whose output only changes when there is a low-to-high transition on a particular input. When the transition happens, make the output equal to the input until the next transition.”
- Result: Verilog Synthesis provides a positive edge-triggered flipflop

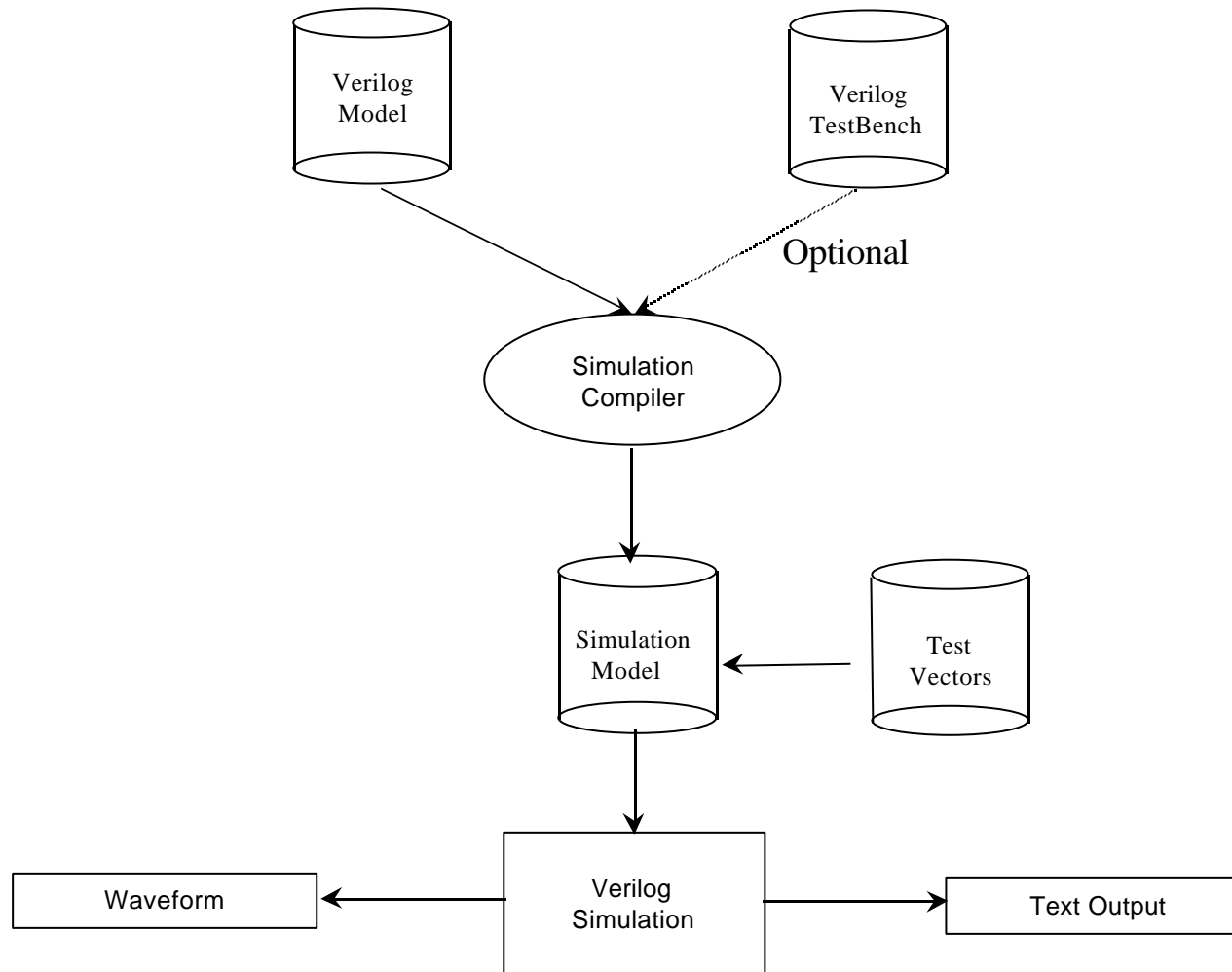
■ ABEL, PALASM, AHDL

- “Give me a D-type flipflop.”
- Result: ABEL, PALASM, AHDL synthesis provides a D-type flipflop. The sense of the clock depends on the synthesis tool.

Typical Synthesis Design Flow



Typical Simulation Design Flow



Verilog Modeling

Verilog - Basic Modeling Structure

```
module module_name (port_list);
```

```
    port declarations
```

```
    data type declarations
```

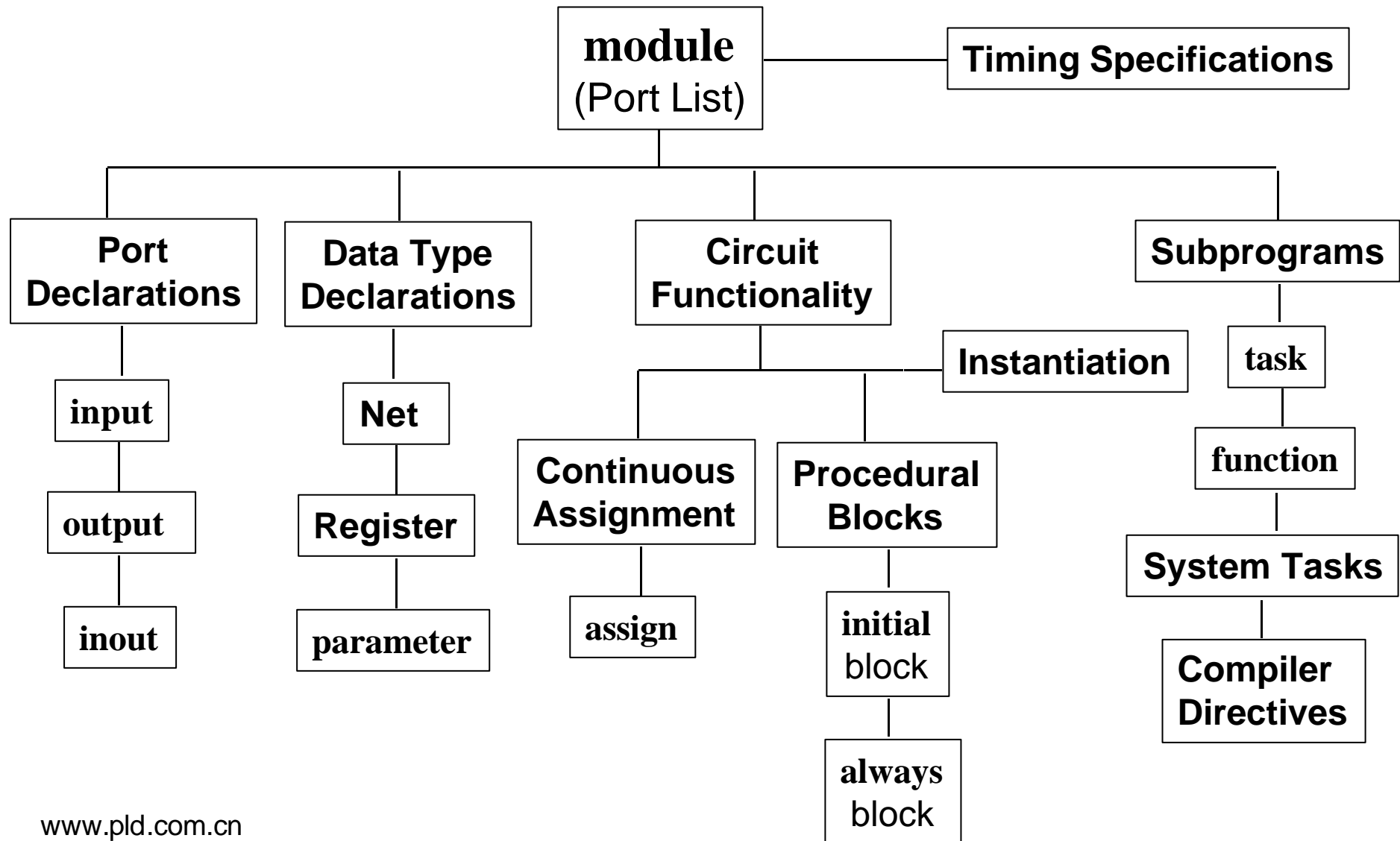
```
    circuit functionality
```

```
    timing specifications
```

```
endmodule
```

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

Components of a Verilog Module



Components of a Verilog Module

module *Module_name* (*Port_list*)

Port declarations (if ports are present)

Parameters (optional)

Data type declarations

Continuous Assignments (assign)

*Procedural Blocks (**initial** and **always**)*
- behavioral statements

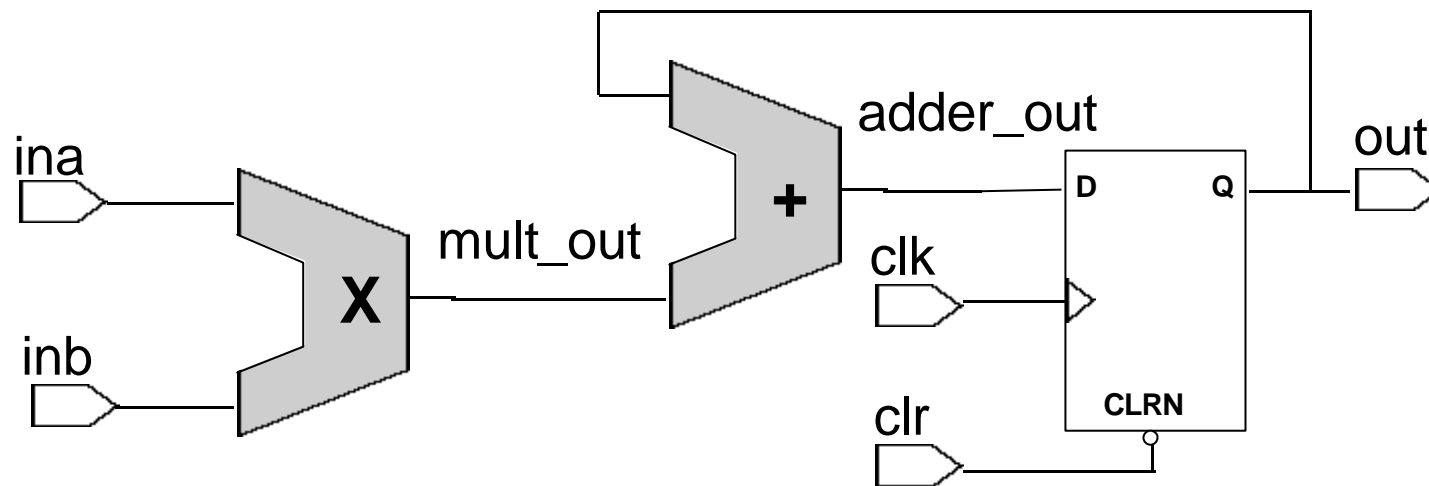
Instantiation of lower-level modules

Tasks and Functions

Timing Specifications

endmodule

Schematic Representation - MAC



Verilog Model: Multiplier-Accumulator (MAC)

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (out, ina, inb, clk, clr);
```

```
input [7:0] ina, inb;
```

```
input clk, clr;
```

```
output [15:0] out;
```

```
wire [15:0] mult_out, adder_out;
```

```
reg [15:0] out;
```

```
parameter set = 10;
```

```
parameter hld = 20;
```

```
assign adder_out = mult_out + out;
```

```
always @ (posedge clk or posedge clr)
```

```
begin
```

```
    if (clr)
```

```
        out = 16'h0000;
```

```
    else
```

```
        out = adder_out;
```

```
    end
```

```
    multa u1(.in_a(ina), .in_b(inb), .m_out(mult_out));
```

```
specify
```

```
    $setup (ina, posedge clk, set);
```

```
    $hold (posedge clk, ina, hld);
```

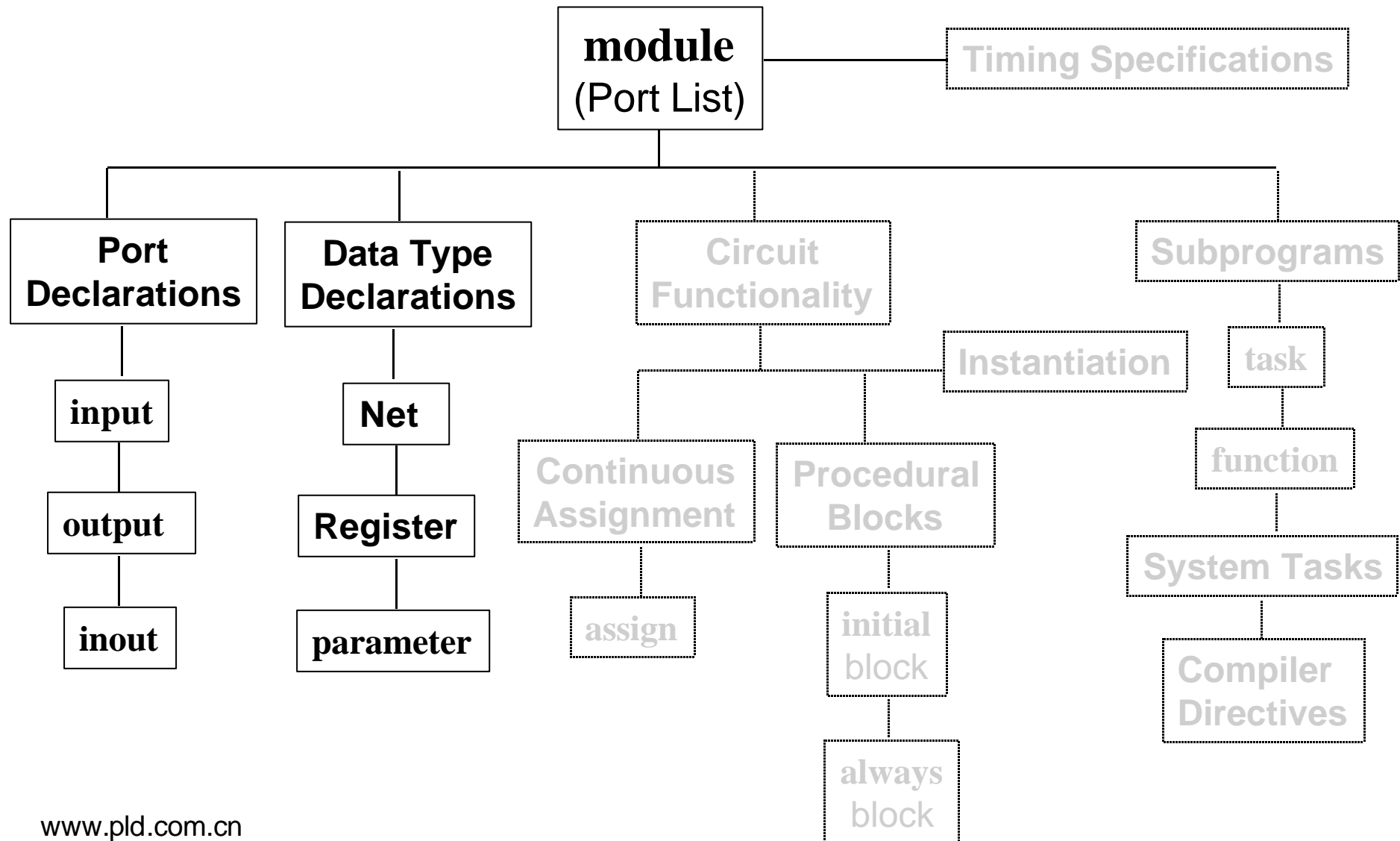
```
    $setup (inb, posedge clk, set);
```

```
    $hold (posedge clk, inb, hld);
```

```
endspecify
```

```
endmodule
```

Let's take a look at:



Ports

■ Port List:

- A listing of the port names
- Example:

```
module mult_acc (out, ina, inb, clk, clr);
```

■ Port Types:

- **input** --> input port
- **output** --> output port
- **inout** --> bidirectional port

■ Port Declarations:

- **<port_type>** *<port_name>*;
- Example:

```
input [7:0] ina, inb;  
input clk, clr;  
output [15:0] out;
```


Ports: List and Declaration

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (out, ina, inb, clk, clr);
```

```
input [7:0] ina, inb;
```

```
input clk, clr;
```

```
output [15:0] out;
```

```
wire [15:0] mult_out, adder_out;
```

```
reg [15:0] out;
```

```
parameter set = 10;
```

```
parameter hld = 20;
```

```
assign adder_out = mult_out + out;
```

```
always @ (posedge clk or posedge clr)
```

```
begin
```

```
    if (clr)
```

```
        out = 16'h0000;
```

```
    else
```

```
        out = adder_out;
```

```
    end
```

```
    multa u1(.in_a(ina), .in_b(inb), .m_out(mult_out));
```

```
specify
```

```
    $setup (ina, posedge clk, set);
```

```
    $hold (posedge clk, ina, hld);
```

```
    $setup (inb, posedge clk, set);
```

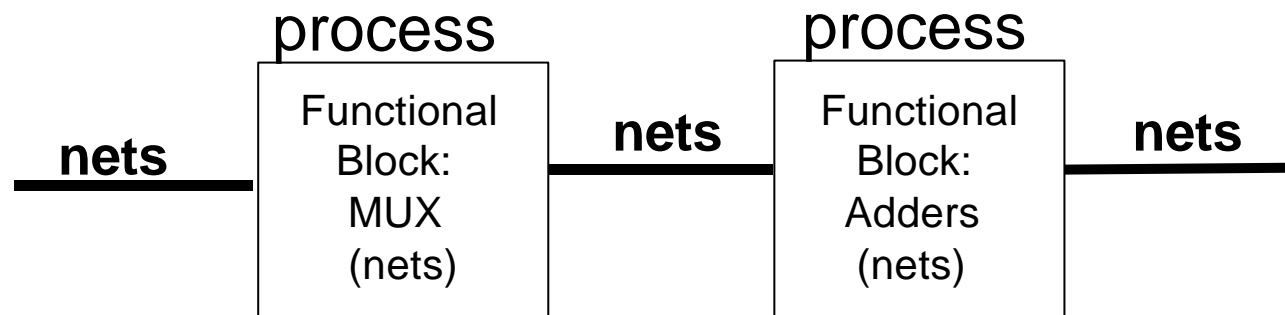
```
    $hold (posedge clk, inb, hld);
```

```
endspecify
```

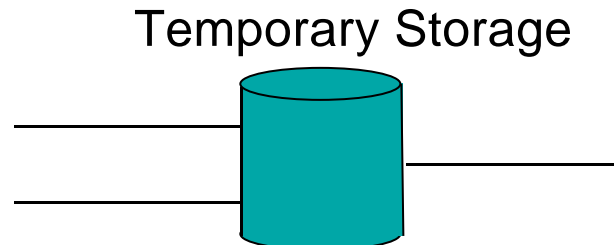
```
endmodule
```

Data Types

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



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



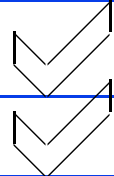
Net Data Type

- **wire** --> represents a node
- **tri** --> represents a tri-state node

- Bus Declarations:
 - **<data_type>** [*MSB* : *LSB*] *<signal name>* ;
 - **<data_type>** [*LSB* : *MSB*] *<signal name>* ;

- Examples:
 - **wire** *<signal name>* ;
 - **wire** [15:0] mult_out, adder_out;

Net Data Types

Net Data Type	Functionality	Supported by Synthesis
wire tri	Used for interconnect	
supply0 supply1	Constant logic value	
wand triand	Used to model ECL	
wor trior	Used to model ECL	
tri0 tri1	Pull-down, Pull-up	
triereg	Stores last value when not driven	

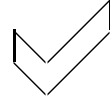
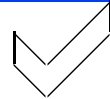
Register Data Types

- **reg** - unsigned variable of any bit size
- **integer** - signed variable (usually 32 bits)

- Bus Declarations:
 - **<data_type>** [*MSB* : *LSB*] *<signal name>* ;
 - **<data_type>** [*LSB* : *MSB*] *<signal name>* ;

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

Register Data Types

Register Data Type	Functionality	Supported by Synthesis
reg	Unsigned variable of any bit size	
integer	Signed variable - usually 32 bits	
time	Unsigned integer - usually 64 bits	
real	Double precision floating point variable	

Memory

- Two dimensional register array
- Can not be a net type
- Examples:

```
reg [31:0] mem[0:1023]; // 1Kx32  
reg [31:0] instr;  
  
instr = mem[2];
```

- Double-indexing is not permitted

```
instr = mem[2][7:0] // Illegal
```

Parameter

- Parameter - assigning a value to a symbolic name

```
parameter size = 8;  
reg [size-1:0] a, b;
```


Data Type

- Every signal (which includes ports) must have a data type
 - Signals must be *explicitly* declared in the data type declarations of your module
 - Ports are, by default, wire net data types if they are *not explicitly* declared

Data Types: Declaration

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (out, ina, inb, clk, clr);
```

```
input [7:0] ina, inb;
```

```
input clk, clr;
```

```
output [15:0] out;
```

```
wire [15:0] mult_out, adder_out;
```

```
reg [15:0] out;
```

```
parameter set = 10;
```

```
parameter hld = 20;
```

```
assign adder_out = mult_out + out;
```

```
always @ (posedge clk or posedge clr)
```

```
begin
```

```
if (clr)
```

```
    out = 16'h0000;
```

```
else
```

```
    out = adder_out;
```

```
end
```

```
multa u1(.in_a(ina), .in_b(inb), .m_out(mult_out));
```

```
specify
```

```
    $setup (ina, posedge clk, set);
```

```
    $hold (posedge clk, ina, hld);
```

```
    $setup (inb, posedge clk, set);
```

```
    $hold (posedge clk, inb, hld);
```

```
endspecify
```

```
endmodule
```

Assigning Values - Numbers and Operators

Assigning Values - Numbers

■ Are **sized** or **unsized**: <size>'<base format><number>

- **Sized** example: `3'b010` = 3-bit wide binary number
 - The prefix (3) indicates the size of number
- **Unsized** example: `123` = 32-bit wide decimal number by default
 - **Defaults**
 - No specified <base format> defaults to **decimal**
 - No specified <size> defaults to **32-bit** wide number

■ Base Format:

- Decimal ('d or 'D) `16'd255` = 16-bit wide decimal number
- Hexadecimal ('h or 'H) `8'h9a` = 8-bit wide hexadecimal number
- Binary ('b or 'B) `'b1010` = 32-bit wide binary number
- Octal ('o or 'O) `'o21` = 32-bit wide octal number

Numbers

- Negative numbers - specified by putting a minus sign before the <size>
 - **Legal:** `-8'd3` = 8-bit negative number stored as 2's complement of 3
 - **Illegal:** `4'd-2` = **ERROR!!**
- Special Number Characters:
 - `'_'` (underscore): used for readability
 - Example: `32'h21_65_bc_fe` = 32-bit hexadecimal number
 - `'x'` or `'X'` (unknown value)
 - Example: `12'h12x` = 12-bit hexadecimal number; LSBs unknown
 - `'z'` or `'Z'` (high impedance value)
 - Example: `1'bz` = 1-bit high impedance number

Numbers

■ Extended

- If MSB is **0**, **x**, or **z**, number is extended to fill MSBs with **0**, **x**, or **z**, respectively
 - Examples: $3'b01 = 3'b001$, $3'bx1 = 3'bxx1$, $3'bz = 3'bzzz$
- If MSB is **1**, number is extended to fill MSBs with **0**
 - Example: $3'b1 = 3'b001$

Short Quiz

- Short Quiz:

- Q: What is the actual value for 4'd017 in binary?

Answers

■ Short Quiz:

- Q: What is the actual value for 4'd017 in binary?
- A: 4'b0001, MSB is truncated (10001)

Arithmetic Operators

Operator Symbol	Operation Performed	Examples $ain = 5, bin = 10, cin = 2'b01, din = 2'b0Z$
+	Add	$bin + cin = 11$
-	Subtract, Negate	$bin - cin = 9, -bin = -10$
*	Multiply	$ain * bin = 50$
/	Divide	$bin / ain = 2$
%	Modulus	$bin \% ain = 0$

- Treats vectors as a whole value
- If any operand is Z or X, then the results are unknown
 - Example: $ain + din = \text{unknown}$
- If results and operands are same size, then carry is lost

Bitwise Operators

Operator Symbol	Operation Performed	Examples $a_{in} = 3'b101$, $b_{in} = 3'b110$, $c_{in} = 3'b01X$
\sim	Invert each bit	$\sim a_{in}$ is $3'b010$
$\&$	And each bit	$a_{in} \& b_{in}$ is $3'b100$, $b_{in} \& c_{in}$ is $3'b010$
$ $	Or each bit	$a_{in} b_{in}$ is $3'b111$
\wedge	Xor each bit	$a_{in} \wedge b_{in}$ is $3'b011$
$\wedge \sim$ or $\sim \wedge$	Xnor each bit	$a_{in} \wedge \sim b_{in} = 3'b100$

- Operates on each bit of the operand
- Result is the size of the largest operand
- Left-extended if sizes are different

Reduction Operators

Operator Symbol	Operation Performed	Examples ain = 5'b10101, bin = 4'b0011 cin = 3'bZ00, din = 3'bX011
&	And all bits	&ain = 1'b0, &din = 1'b0
~&	Nand all bits	~&ain = 1'b1
 	Or all bits	 ain = 1'b1, cin = 1'bX
~ 	Nor all bits	~ ain = 1'b0
^	Xor all bits	^ain = 1'b1
~^ or ^~	Xnor all bits	~^ain = 1'b0

- Reduces a vector to a single bit
- X or Z are considered unknown, but result maybe a known value
 - Example: **&din** results in 1'b0

Relational Operators

Operator Symbol	Operation Performed	Examples
		ain 3'b010, bin = 3'b100, cin=3'b111 din = 3'b01z, ein = 3'b01x
>	Greater than	ain > bin results false (1'b0)
<	Less than	ain < bin results true (1'b1)
>=	Greater than or equal	ain >= din results unknown (1'bX)
<=	Less than or equal	ain <= ein results unknown (1'bX)

- Used to compare values
- Returns a 1 bit scalar value of boolean true (1) / false (0)
- If any operand is Z or X, then the results are unknown

Equality Operators

Operator Symbol	Operation Performed	Examples
		ain 3'b010, bin = 3'b100, cin=3'b111 din = 3'b01z, ein = 3'b01x
==	Equality	ain == cin results false (1'b0)
!=	Inequality	ein != ein results unknown (1'bX)
===	Case equality	ein === ein results true (1'b1)
!==	Case inequality	ein !== din results true (1'b1)

- Used to compare values
- Returns a 1 bit scalar value of boolean true (1) / false (0)
- If any operand is Z or X, then the results are unknown
- Case equality and inequality includes x and z

Logical Operators

Operator Symbol	Operation Performed	Examples ain = 3'b101, bin = 3'b000
!	Not true	!ain is false (1'b0)
&&	Both expressions true	ain && bin results false (1'b0)
 	One or both expressions true	ain bin results true (1'b1)

- Returns a 1 bit scalar value of boolean true (1) / false (0)
- If any operand is Z or X, then the results are unknown

Shift Operators

Operator Symbol	Operation Performed	Examples ain = 4'b1010, bin = 4'b10X0
>>	Shift right	bin >> 1 results 4'b010X
<<	Shift left	ain << 2 results 4'b1000

- Shifts a vector left or right some number of bits
- Zero fills
- Shifted bits are lost

Miscellaneous Operators

Operator Symbol	Operation Performed	Examples
?:	Conditional	(condition) ? true_val : false_val; sig_out = (sel==2'b01) ? A : B ;
{ }	Concatenate	ain = 3'b010, bin = 4'b1100 {ain,bin} results 7'b0101100
{ { } }	Replicate	{3{2'b10}} results 6'b101010

Operator Precedence

■ Operators default precedence

+ , - , ! , ~ (unary)

+ , - (Binary)

<< , >>

< , > , <= , >=

== , !=

&

^ , ^~ or ~^

|

&&

||

?: (ternary)

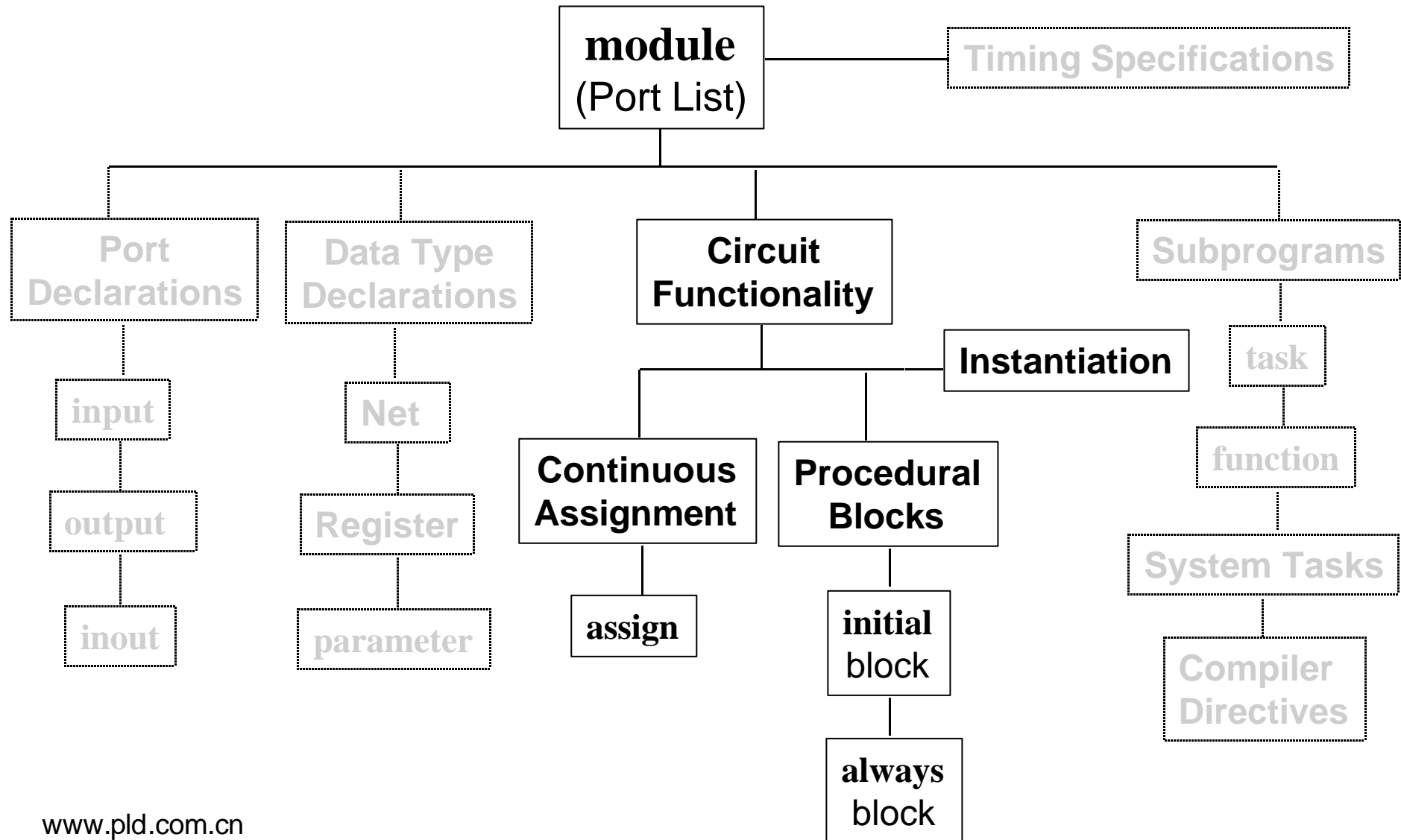
Highest Priority



Lowest Priority

■ () can be used to override default

Components of a Verilog Module

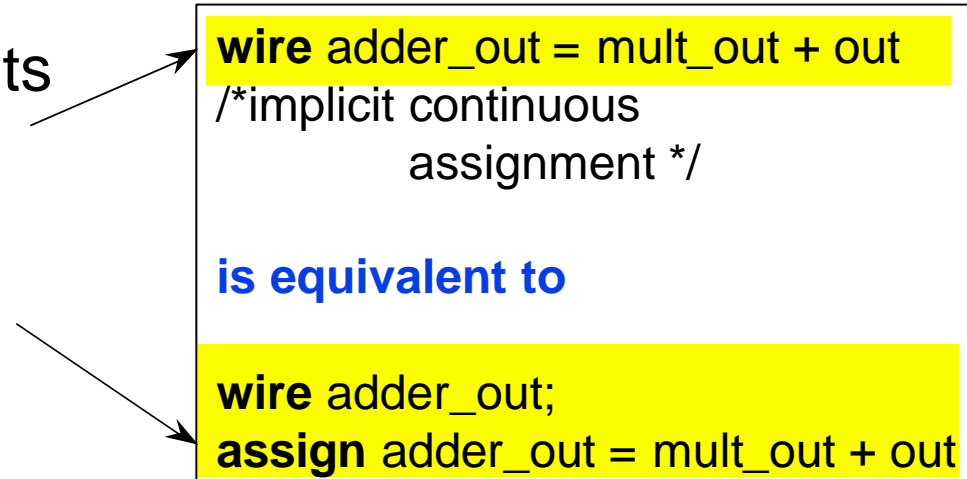


Behavioral Modeling

Continuous Assignment

Continuous Assignments

- Model the behavior of Combinatorial Logic by using operators
- Continuous assignments can be made when the net is declared
- OR by using the **assign** statement



```
wire adder_out = mult_out + out  
/*implicit continuous  
assignment */
```

is equivalent to

```
wire adder_out;  
assign adder_out = mult_out + out
```

Continuous Assignments: Characteristics

- 1) Left-hand side (LHS) must be a net data type
- 2) Always active: When one of the right-hand side (RHS) operands changes, expression is evaluated, and LHS net is updated immediately
- 3) RHS can be net, register, or function calls
- 4) Delay values can be assigned to model gate delays

```
wire adder_out = mult_out + out  
/*implicit continuous  
assignment */
```

is equivalent to

```
wire adder_out;  
assign adder_out = mult_out + out
```

Continuous Assignments

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (out, ina, inb, clk, clr);
```

```
input [7:0] ina, inb;
```

```
input clk, clr;
```

```
output [15:0] out;
```

```
wire [15:0] mult_out, adder_out;
```

```
reg [15:0] out;
```

```
parameter set = 10;
```

```
parameter hld = 20;
```

```
// Continuous Assignment  
assign adder_out = mult_out + out;
```

```
always @ (posedge clk or posedge clr)
```

```
begin
```

```
if (clr)
```

```
    out = 16'h0000;
```

```
else
```

```
    out = adder_out;
```

```
end
```

```
multa u1(.in_a(ina), .in_b(inb), .m_out(mult_out));
```

```
specify
```

```
    $setup (ina, posedge clk, set);
```

```
    $hold (posedge clk, ina, hld);
```

```
    $setup (inb, posedge clk, set);
```

```
    $hold (posedge clk, inb, hld);
```

```
endspecify
```

```
endmodule
```

Continuous Assignment - Example

```
module ander (out, ina, inb);  
  
  input [7:0] ina, inb;  
  output [7:0] out;  
  
  wire [7:0] out;  
  
  assign out = ina & inb;  
  
endmodule
```

Simulation Time

- Simulation Time is the same for all modules during a simulation run
 - Simulation starts at time 0
 - Simulation time advances when all processes at current time are simulated

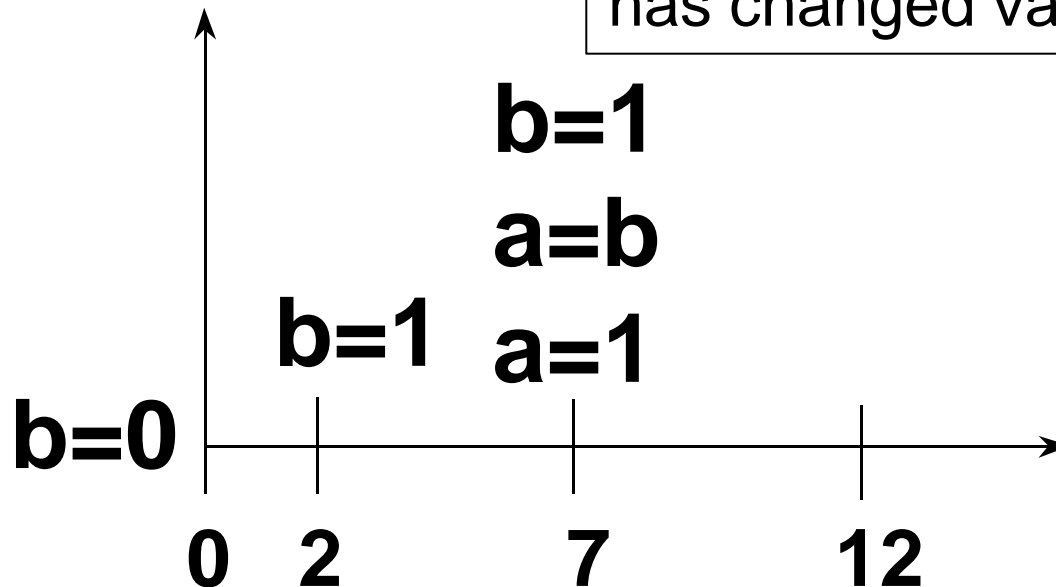
One Type of Continuous Assignment Delay

■ Regular Assignment Delay

assign #5 a = b;

Statement Executed

Input 'b' is sampled and is assigned to 'a' 5 time units after 'b' has changed value.



Behavioral Modeling

Procedural Blocks

Two Structured Procedures (Blocks)

- **initial** Block - Used to initialize behavioral statements for simulation
 - **always** Block - Used to describe the circuit functionality using behavioral statements
-
- ⇒ Each **always** and **initial** block represents a separate process
 - ⇒ Processes run in parallel and start at simulation time 0
 - ⇒ However, statements inside a process execute sequentially
 - ⇒ **always** and **initial** blocks cannot be nested

Two Procedural Blocks

always and **initial** blocks

Behavioral Statements

Assignments:
Blocking
Nonblocking

Timing Specifications

Initial Block

- Consists of behavioral statements
- If there are more than one behavioral statement inside an **initial** block, the statements need to be grouped using the keywords **begin** and **end**.
- If there are multiple **initial** blocks, each block executes concurrently at time 0.

⇒ Not supported by synthesis

Initial Block

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

Initial Block Example

```
module system;
```

```
reg a, b, c, d;
```

```
// single statement
```

```
initial
```

```
    a = 1'b0;
```

```
/* multiple statements:  
   needs to be grouped */
```

```
initial
```

```
    begin
```

```
        b = 1'b1;
```

```
        #5 c = 1'b0;
```

```
        #10 d = 1'b0;
```

```
    end
```

```
initial
```

```
    #20 $finish;
```

```
endmodule
```

Time	Statement Executed
0	a = 1'b0; b = 1'b1;
5	c = 1'b0;
15	d = 1'b0;
20	\$finish

Always Block

- Consists of behavioral statements
- If there are more than one behavioral statement inside an **always** block, the statements can be grouped using the keywords **begin** and **end**.
- If there are multiple **always** blocks, each block executes concurrently.

Always Block

- Used to model a process that is repeated continuously in a digital circuit
 - ⇒ An **always** block starts at time 0 and executes the behavioral statements continuously in a looping fashion.
 - ⇒ Behavioral statements inside an **always** block execute sequentially.
 - ⇒ Therefore, order of statements does matter.

Characteristics

- 1) Left-hand side (LHS) must be a register data type: Can be a reg, integer, real, or time variable or a memory element
- 2) LHS can be a bit-select or part-select
- 3) A concatenation of any of the above
- 4) Right-hand side (RHS): All operators can be used in behavioral expressions

```
reg [15:0] out;
```

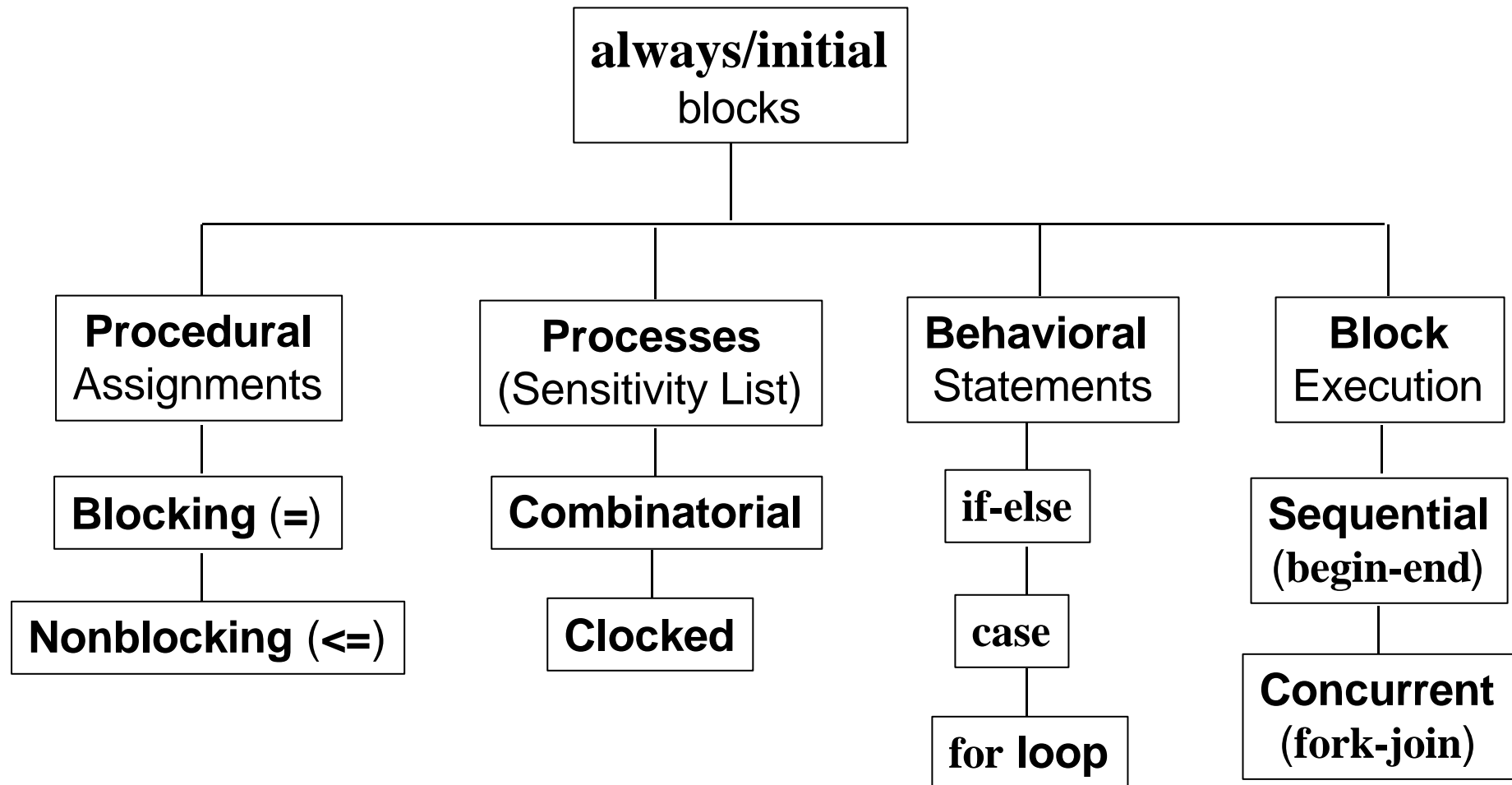
```
always @ (posedge clk or posedge clr)
begin
    if (clr)
        out = 16'h0000;
    else
        out = adder_out;
end
```

Always Block - Example

```
module clock_gen (clk);  
output clk;  
reg clk;  
  
parameter period=50, duty_cycle=50;  
  
initial  
    clk = 1'b0;  
  
always  
    #(duty_cycle*period/100) clk = ~clk;  
  
initial  
    #100 $finish;  
  
endmodule
```

Time	Statement Executed
0	clk = 1'b0
25	clk = 1'b1
50	clk = 1'b0
75	clk = 1'b1
100	\$finish

Always/Initial Blocks



Procedural Assignments

- Let's first look at the two different procedural assignments:
 - Blocking Assignment
 - Nonblocking Assignment

Procedural Assignments

- Procedural Assignments - assignments that are made inside the two structured procedures (**initial** and **always** blocks)
- Update values of **reg**, **integer**, **real**, or **time** variables
- Value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

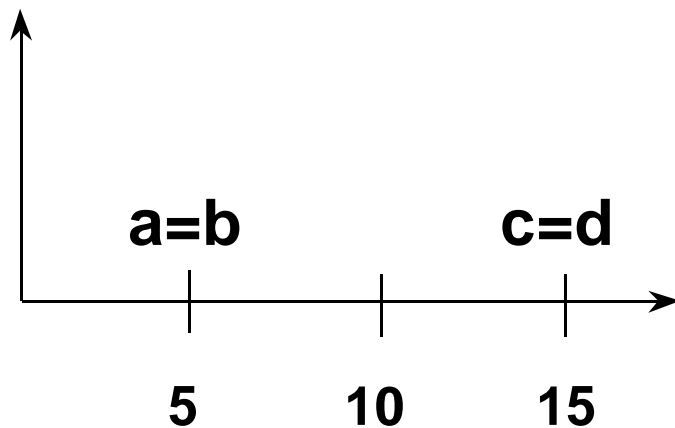
Two types of Procedural Assignments

- Blocking Assignment (=) : executed in the order they are specified in a sequential block
- Nonblocking Assignment (<=) : allow scheduling of assignments without blocking execution of the statements that follow in a sequential block
 - Recommended: Use Nonblocking assignments for clocked processes when writing synthesizable code.

Blocking vs. Nonblocking Assignments

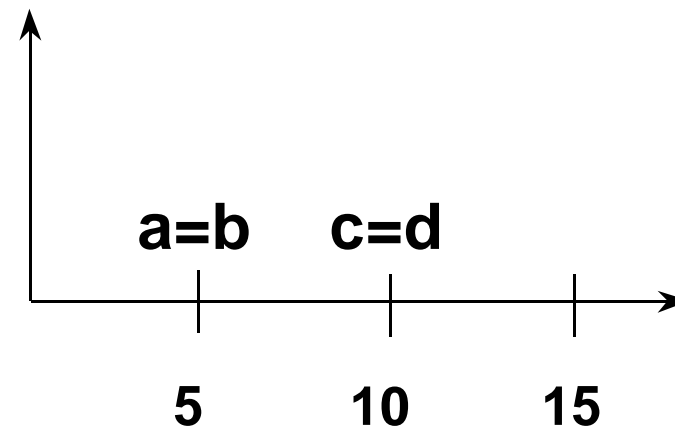
Blocking (=)

```
initial
begin
    #5    a = b;
    #10   c = d;
end
```



Nonblocking (<=)

```
initial
begin
    #5    a <= b;
    #10   c <= d;
end
```



Simulation Time

- Simulation Time is the same for all modules during a simulation run
 - Simulation starts at time 0
 - Simulation time advances when all processes at current time are simulated

3 Delay Controls for Procedural Assignment

- Regular Delay Control
- Intra-assignment Delay Control
- Zero Delay Control

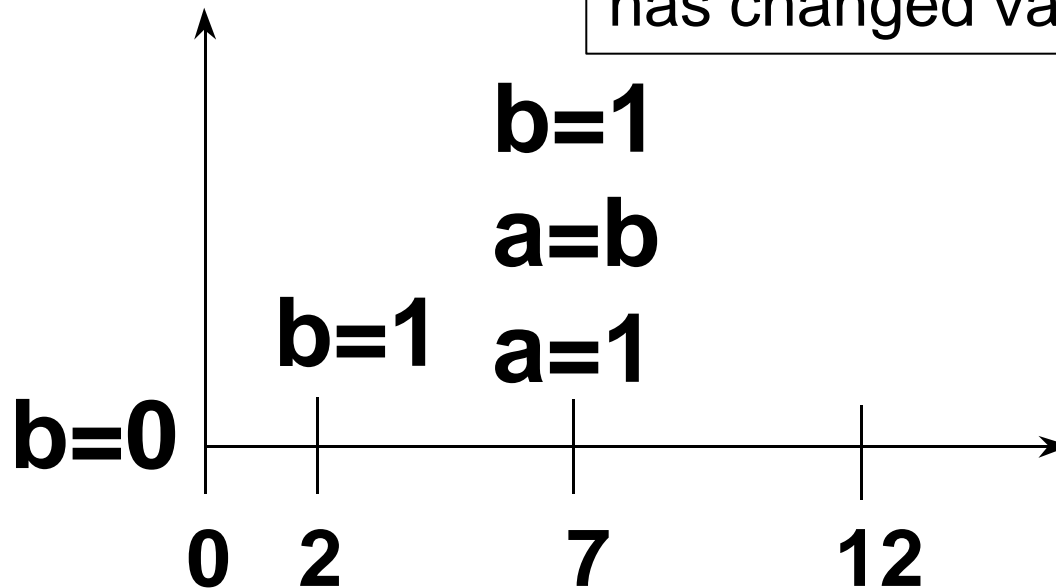
Regular Delay Control

■ Regular Assignment Delay

assign #5 a = b;

Statement Executed

Input 'b' is sampled and is assigned to 'a' 5 time units after 'b' has changed value.



Intra-assignment Delay Control

■ Intra-assignment Delay Control

initial

a = #5 b;

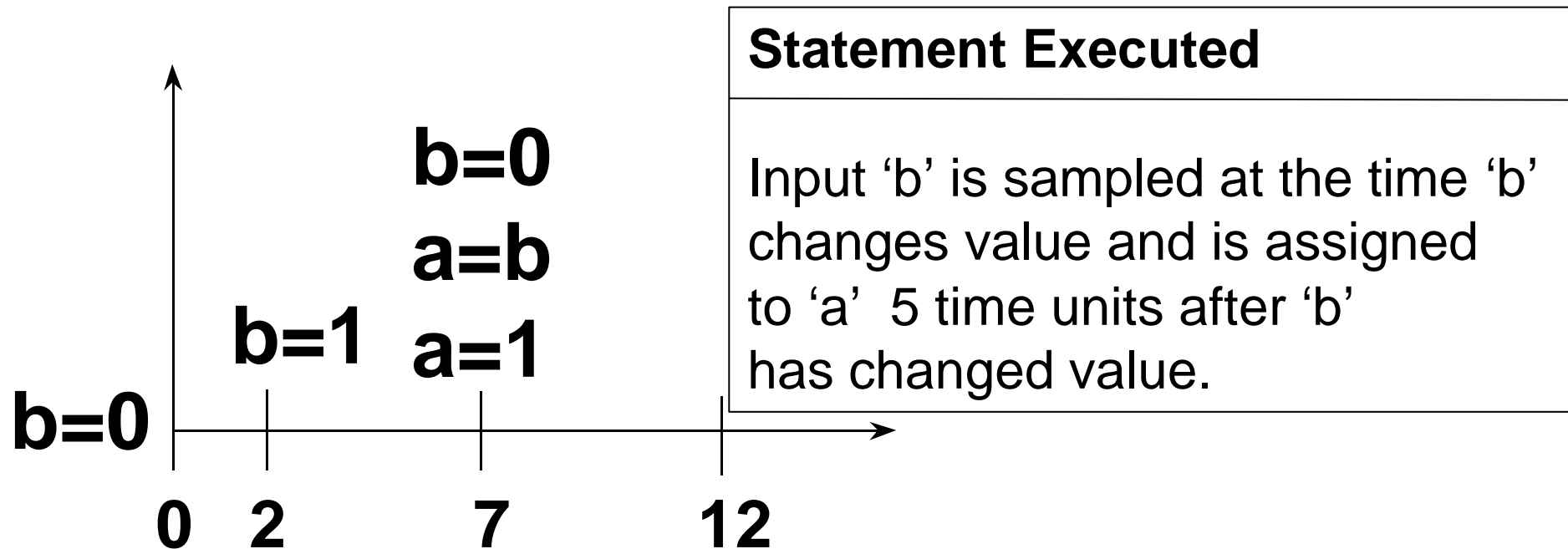
is equivalent to

<=>

initial

temp = b;

#5 a = temp;



Zero Delay Control

■ Zero Delay Control

initial

begin

a = 0;

b = 0;

end

initial

begin

#0 a = 1;

#0 b = 1;

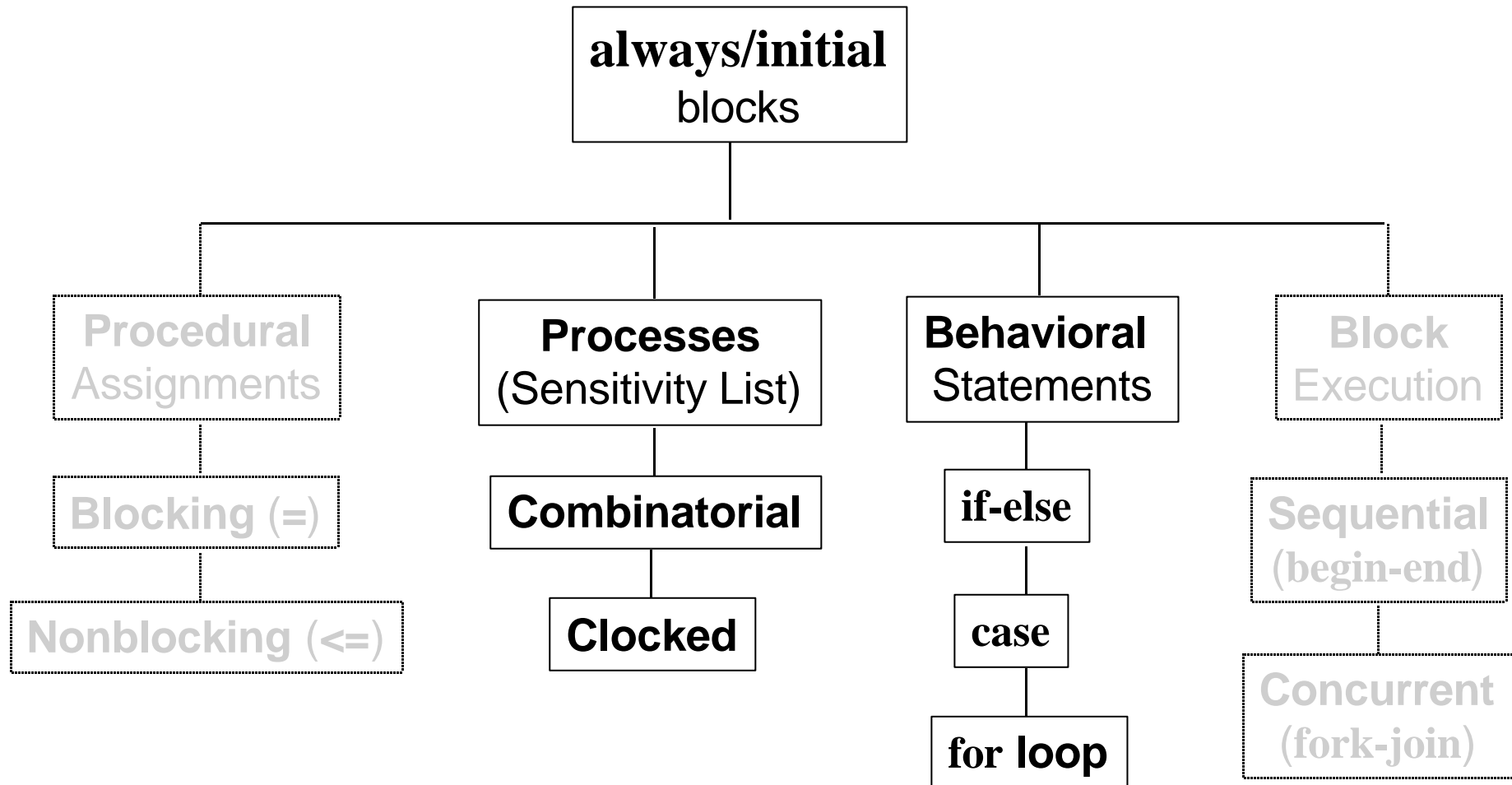
end

Statement Executed

All four statements will be executed at simulation time 0. However, since a = 1 and b = 1 have #0, they will be executed last.

- Not recommended to assign 2 different values to a variable
 - May cause a race condition
- Zero Delay Control - provides a useful way of controlling the order of execution

Always/Initial Blocks



Processes and Behavioral Statements

- Now, let's look at the two different processes:
 - Combinatorial Process
 - Clocked Process
- Let's also look at some behavioral statements

Sensitivity List

- Sensitivity List:

```
always @(sensitivity_list)
    begin
        -- Statement #1
        -- .....
        -- Statement #N
    end
```

- This procedural block (process) executes after a change in any signal in the Sensitivity List

Two Types of Processes

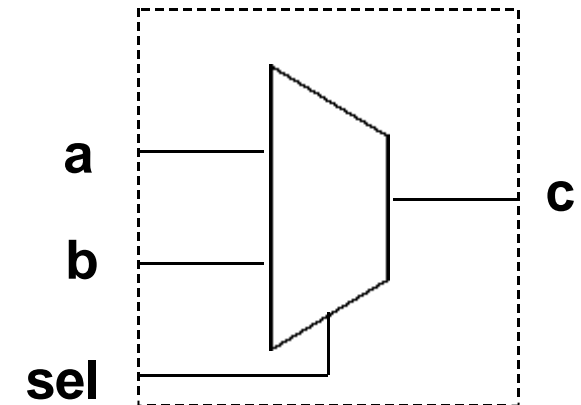
- **Combinatorial Process**

- Sensitive to all inputs used in the combinatorial logic

- **Example**

always @(a or b or sel)

sensitivity list includes all inputs used in the combinatorial logic



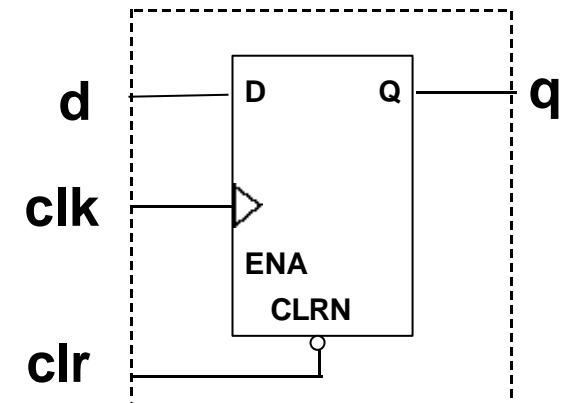
- **Clocked Process**

- Sensitive to a clock or/and control signals

- **Example**

always @(posedge clk or negedge clr)

sensitivity list does not include the d input, only the clock or/and control signals



Combinatorial Process

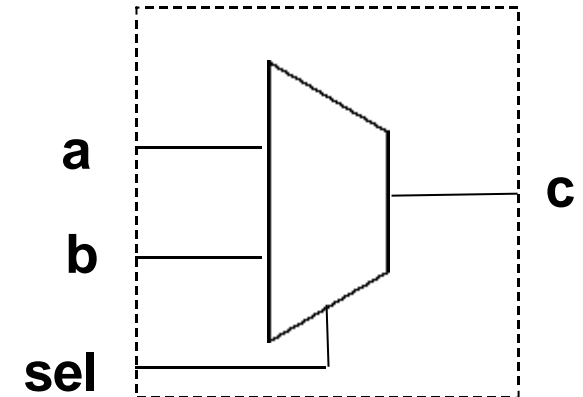
- Let's first look at:
 - Combinatorial Process Examples

- **Combinatorial Process**
 - Sensitive to all inputs used in the combinatorial logic

- **Example**

always @(a or b or sel)

sensitivity list includes all inputs used in the combinatorial logic



Behavioral Statements

- Behavioral Statements
 - IF-ELSE statement
 - CASE statement
 - Loop statements

- These Behavioral Statements can also be used in a Clocked Process

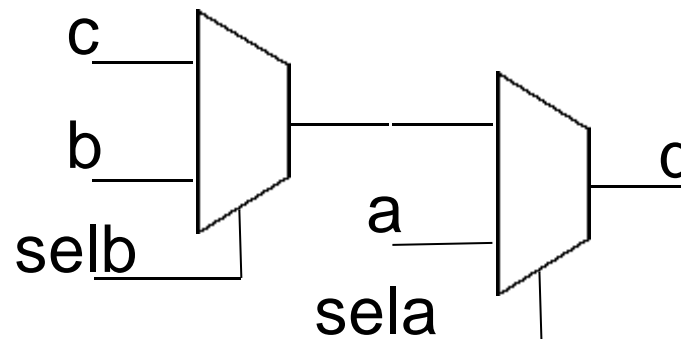
If-Else Statements

■ Format:

```
if (<condition1>)
    sequence of statement(s)
else
    if (<condition2>)
        sequence of statement(s)
        .
        .
    else
        sequence of statement(s)
```

■ Example:

```
always @(sela or selb or a or b or c)
    begin
        if (sela)
            q = a;
        else
            if (selb)
                q = b;
            else
                q = c;
    end
```



If-Else Statements

- Conditions are evaluated in order from top to bottom
 - Prioritization
- The first condition, that is true, causes the corresponding sequence of statements to be executed.
- If all conditions are false, then the sequence of statements associated with the “**else**” clause is evaluated.

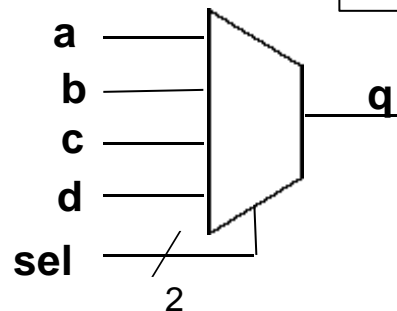
Case Statement

■ Format:

```
case (expression)
  <condition1> :
    sequence of statement(s)
  <condition2> :
    sequence of statement(s)
    .
    .
  default :
    sequence of statement(s)
endcase
```

■ Example:

```
always @(sel or a or b or c or d)
begin
  case (sel)
    2'b00 :
      q = a;
    2'b01 :
      q = b;
    2'b10 :
      q = c;
    default :
      q = d;
  endcase
end
```



Case Statement

- Conditions are evaluated at once
 - No Prioritization
- **All** possible conditions must be considered
- **default** clause evaluates all other possible conditions that are not specifically stated.

Two Other Forms of Case Statements

■ casez

- Treats all 'z' values in the case conditions as don't cares, instead of logic values
- All 'z' values can also be represented by '?'

■ casex

- Treats all 'x' and 'z' values in the case conditions as don't cares, instead of logic values

casez (encoder)

```
4'b1??? : high_lvl = 3;  
4'b01?? : high_lvl = 2;  
4'b001? : high_lvl = 1;  
4'b0001 : high_lvl = 0;  
default : high_lvl = 0;
```

endcase

- if **encoder** = 4'b1zzz, then **high_lvl** = 3

casex (encoder)

```
4'b1xxx : high_lvl = 3;  
4'b01xx : high_lvl = 2;  
4'b001x : high_lvl = 1;  
4'b0001 : high_lvl = 0;  
default : high_lvl = 0;
```

endcase

- if **encoder** = 4'b1xzx, then **high_lvl** = 3

Loop Statements

- **forever** loop - executes continually
- **repeat** loop - executes a fixed number of times
- **while** loop - executes if expression is true
- **for** loop - executes once at the start of the loop and then executes if expression is true

⇒ Loop statements - used for repetitive operations

Forever and Repeat Loops

- **forever** loop - executes continually

```
initial
begin
    clk = 0;
    forever #25 clk = ~clk;
end
```

Clock with period
of 50 time units

- **repeat** loop - executes a fixed number of times

```
if (rotate == 1)
    repeat (8)
        begin
            tmp = data[15];
            data = {data << 1, tmp};
        end
```

Repeats a rotate
operation 8 times

While Loop

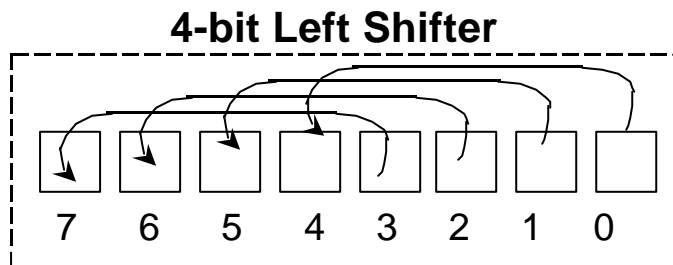
- **while** loop - executes if expression is true

```
initial
begin
    count = 0;
    while (count < 101)
        begin
            $display ("Count = %d", count);
            count = count + 1;
        end
    end
end
```

Counts from 0 to 100
Exits loop at count 101

For Loop

- **for loop** -
executes once
at the start of
the loop and
then executes
if expression is
true



```
integer i; // declare the index for the FOR LOOP
```

```
always @(inp or cnt)
begin
```

```
    result[7:4] = 0;
    result[3:0] = inp;
    if (cnt == 1)
        begin
```

```
            for (i = 4; i <= 7; i = i + 1)
```

```
                begin
```

```
                    result[i] = result[i-4];
```

```
                end
```

```
            result[3:0] = 0;
```

```
        end
```

Clocked Process

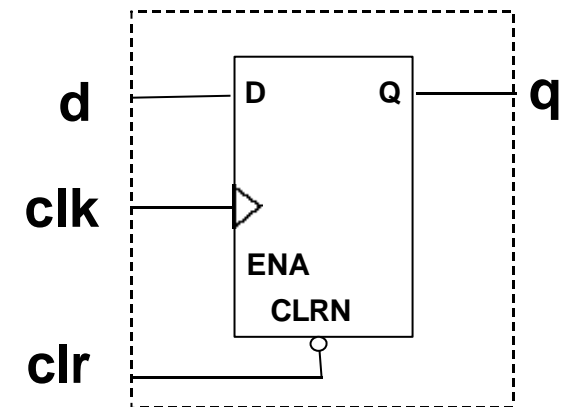
- Let's now look at:
 - Clocked Process Examples
 - Functional for synthesis
- Nonblocking assignments (\leq) are used for clocked processes when writing synthesizable code

- **Clocked Process**
 - Sensitive to a clock or/and control signals

- **Example**

always @(posedge clk or negedge clr)

*sensitivity list does not include the **d** input,
only the clock or/and control signals*



Functional Latch vs. Functional Flipflop

Level-sensitive Latch

```
module latch (d, gate,q);  
  
input d, gate ;  
output q ;  
  
wire d, gate ;  
reg q ;  
  
always @(d or gate)  
    if (gate)  
        q <= d ;  
endmodule
```

Edge-triggered Flipflop

```
module dff ( d, clk, q);  
  
input d, clk ;  
output q;  
  
wire d, clk;  
reg q ;  
  
always @(posedge clk)  
    q <= d ;  
endmodule
```

Synchronous vs. Asynchronous

Synchronous Preset & Clear

```
module sync (d,clk, clr, pre, q);  
  
input d, clk, clr, pre ;  
output q ;  
  
reg q ;  
  
always @(posedge clk)  
begin  
    if (clr)  
        q <= 1'b0 ;  
    else if (pre)  
        q <= 1'b1 ;  
    else  
        q <= d ;  
end  
endmodule
```

Asynchronous Clear

```
module async (d,clk, clr, q);  
  
input d, clk, clr ;  
output q ;  
  
reg q ;  
  
always @(posedge clk or posedge clr)  
begin  
    if (clr)  
        q <= 1'b0 ;  
    else  
        q <= d ;  
end  
endmodule
```

Clock Enable

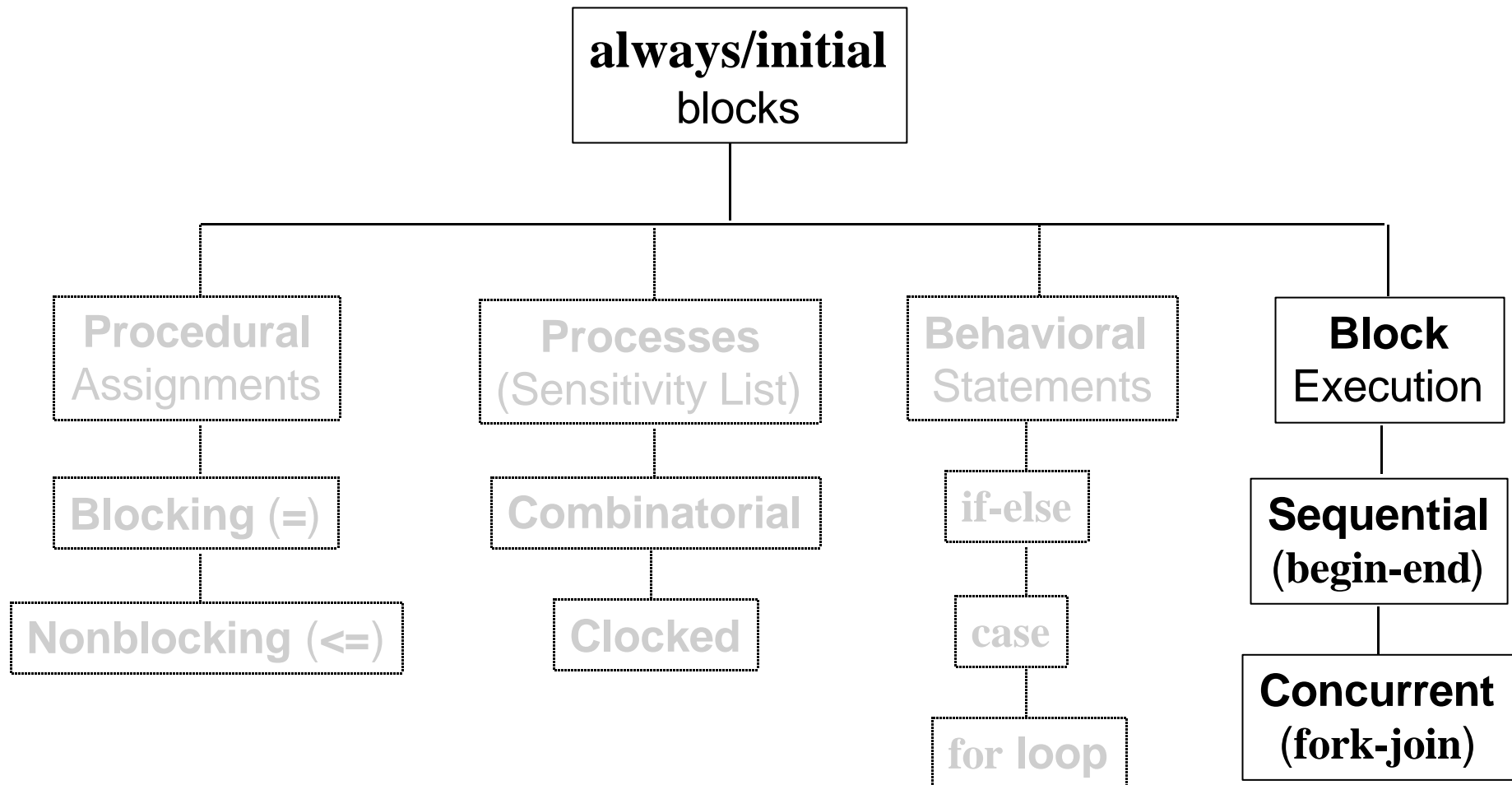
Clock Enable

```
module clk_enb (d, ena, clk, q) ;  
  
input d, ena, clk ;  
output q ;  
  
reg q ;  
  
/* If clock enable port does not exist in target technology,  
a mux is generated */  
  
always @( posedge clk )  
    if (ena)  
        q <= d ;  
  
endmodule
```


Functional Counter

```
module cntr(q, aclr, clk, func, d);
input aclr, clk;
input [7:0] d;
input [1:0] func;           // Controls the functionality
output [7:0] q;
reg [7:0]q;
always @(posedge clk or posedge aclr) begin
    if (aclr)
        q <= 8'h00;
    else
        case (func)
            2'b00: q <= d;      // Loads the counter
            2'b01: q <= q + 1; // Counts up
            2'b10: q <= q - 1; // Counts down
            2'b11: q <= q;
        endcase
    end
endmodule
```

Always/Initial Blocks



Block Execution

- Finally, let's look at the two different block execution inside an always block:
 - Sequential Blocks
 - Parallel Blocks

Two Types of Block Executions

- Sequential Blocks - statements between begin and end execute sequentially
 - If there are multiple behavioral statements inside an **initial** and **always** block and you want the statements to execute sequentially, the statements must be grouped using the keywords **begin** and **end**.
- Parallel Blocks - statements between fork and join execute in parallel
 - If there are multiple behavioral statements inside an **initial** and **always** block and you want the statements to execute in parallel, the statements must be grouped using the keywords **fork** and **join**.

Sequential vs. Parallel Blocks

- Sequential and Parallel Blocks can be nested

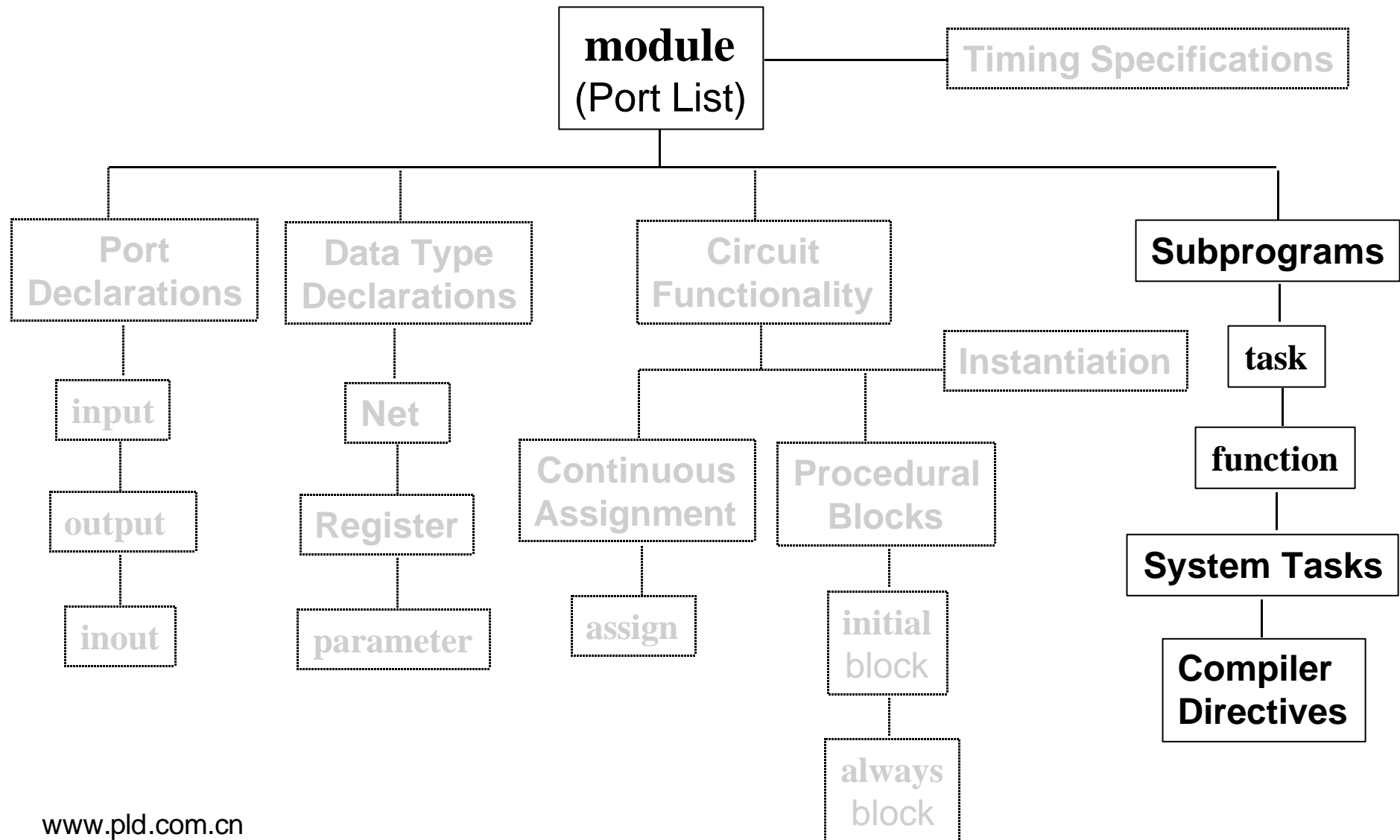
```
initial
    fork
        #10 a = 1;
        #15 b = 1;
    begin
        #20 c = 1;
        #10 d = 1;
    end
    #25 e = 1;
join
```

Time	Statement Executed
10	a = 1
15	b = 1
20	c = 1
25	e = 1
30	d = 1

Behavioral Modeling

Tasks and Functions

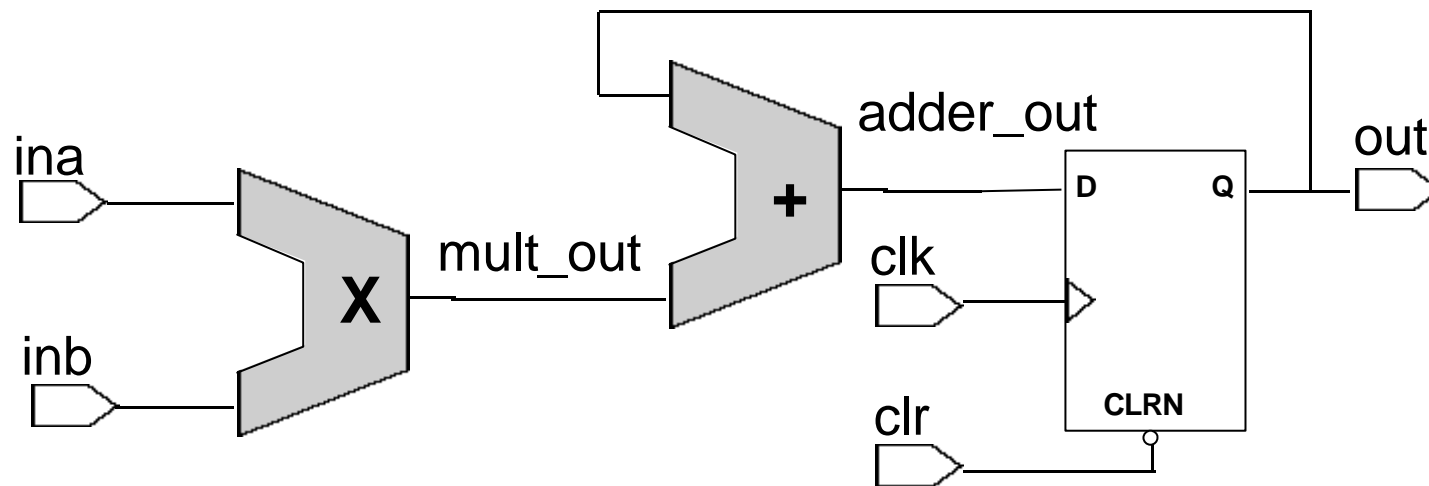
Components of a Verilog Module



Verilog Functions and Tasks

- Function and Tasks are subprograms.
- Useful for code that is repetitive in module
- Add to module readability
- Function
 - Return a value based on its inputs
 - Produces combinatorial logic
 - Used in expressions: `assign mult_out = mult (ina, inb);`
- Tasks
 - Like procedures in other languages
 - Can be combinatorial or registered.
 - Task are invoked as statement: `stm_out (nxt, first, sel, filter);`

Create a Function for the multiplier



Function Definition - Multiplier

Function Definition:

```
function [15:0] mult;  
    input [7:0] a, b ;  
    reg [15:0] r;  
    integer i ;  
begin  
    if (a[0] == 1)  
        r = b;  
    else  
        r = 0 ;  
    for (i =1; i <=7; i = i + 1)  
        begin  
            if (a[i] == 1)  
                r = r +b <<i ;  
            end  
    mult = r;  
end  
endfunction
```

Function Invocation - MAC

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (out, ina, inb, clk, clr);
```

```
input [7:0] ina, inb;
```

```
input clk, clr;
```

```
output [15:0] out;
```

```
wire [15:0] mult_out, adder_out;
```

```
reg [15:0] out;
```

```
parameter set = 10;
```

```
parameter hld = 20;
```

```
assign adder_out = mult_out + out;
```

```
always @ (posedge clk or posedge clr)  
begin
```

```
    if (clr)
```

```
        out = 16'h0000;
```

```
    else
```

```
        out = adder_out;
```

```
    end
```

```
// Function Invocation
```

```
assign mult_out = mult (ina, inb);
```

```
specify
```

```
    $setup (ina, posedge clk, set);
```

```
    $hold (posedge clk, ina, hld);
```

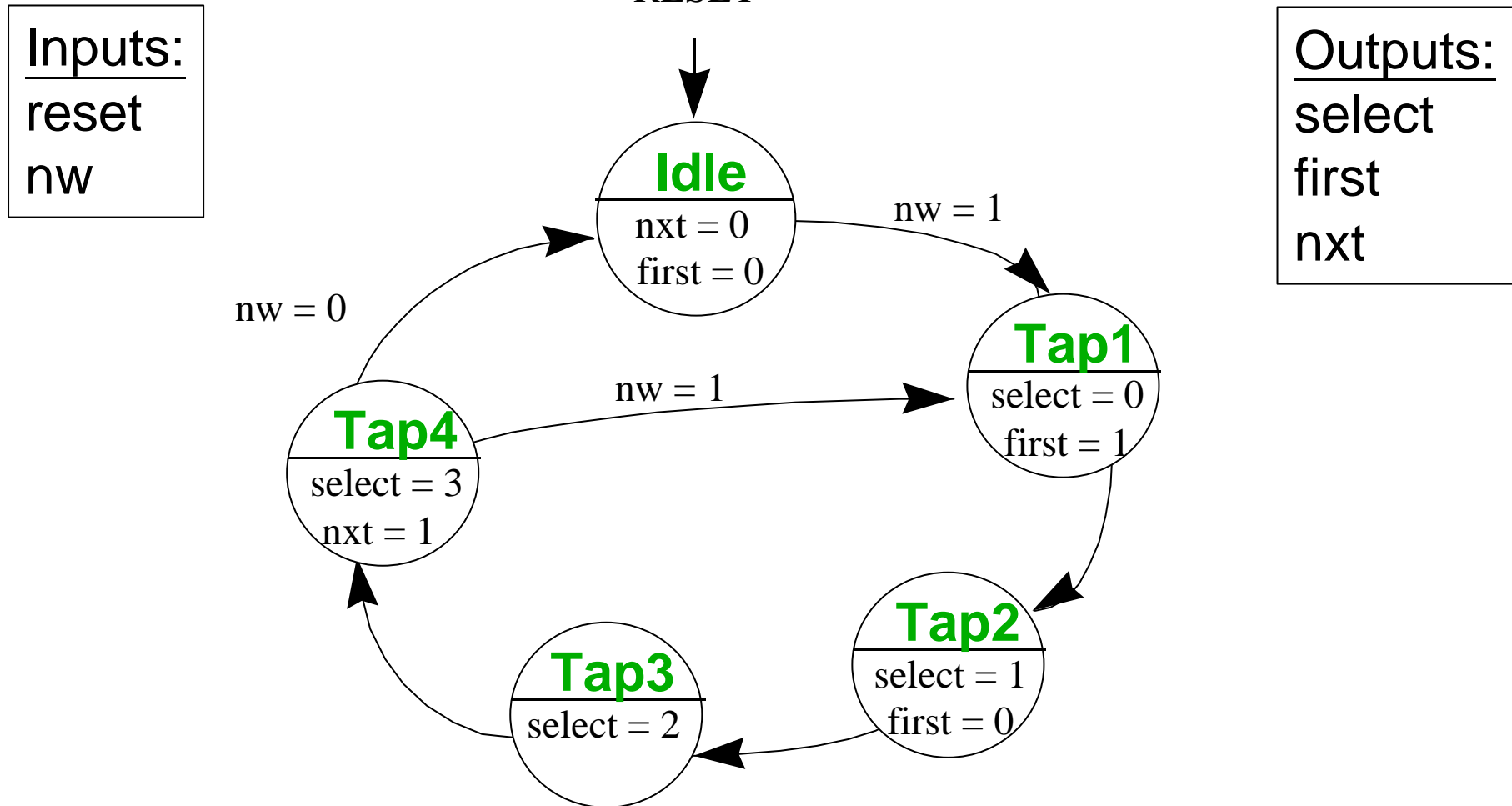
```
    $setup (inb, posedge clk, set);
```

```
    $hold (posedge clk, inb, hld);
```

```
endspecify
```

```
endmodule
```

Create a Task for the Statemachine Output



Task Definition - Statemachine Output

```
task stm_out (nxt, first, sel, filter);  
input [2:0] filter;  
output nxt, first;  
output [1:0] sel;  
reg nxt, first;  
reg [1:0] sel;  
parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;  
begin  
    nxt = 0; first = 0;  
    case (filter)  
        tap1: begin sel = 0; first = 1; end  
        tap2: sel = 1;  
        tap3: sel = 2;  
        tap4: begin sel = 3; nxt = 1; end  
        default: begin nxt = 0; first = 0; sel = 0; end  
    endcase  
  
    end  
endtask
```

Task Invocation - Statemachine

```
module stm_fir (nxt, first, sel, clk, reset, nw);
input clk, reset, nw;
output nxt, first;
output [1:0] sel;
reg nxt, first;
reg [1:0] sel;
reg [2:0] filter;
parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;
always @(posedge clk or posedge reset)
begin    if (reset) filter = idle;
        else case (filter)
            idle: if (nw==1) filter = tap1;
            tap1: filter = tap2;
            tap2: filter = tap3;
            tap3: filter = tap4;
            tap4: if (nw==1) filter = tap1;
                else filter = idle; endcase end

always @(filter)
    // Task Invocation
    stm_out (nxt, first, sel, filter);
```

//Indicate the transition among states

//Output based on corresponding state

Differences

Functions

- Can enable another function but not another task
- Always executes in zero simulation time
- Can not contain any delay, event, or timing control statements
- Must have at least one input argument
- Always return a single value
- Can not have output or inout arguments

Tasks

- Can enable other tasks and functions
- May execute in non-zero simulation time
- May contain delay, event, or timing control statements
- May have zero or more input, output, or inout arguments
- Returns zero or more values

Review - Behavioral Modeling

Continuous Assignment

```
module full_adder4(fco, fsum, cin, a, b);  
  
output [3:0] fsum;  
output fco;  
input [3:0] a, b;  
input cin;  
  
wire [3:0] fco, fsum;  
wire cin;  
  
assign {fco, fsum} = cin + a + b;  
  
endmodule
```

Procedural Block

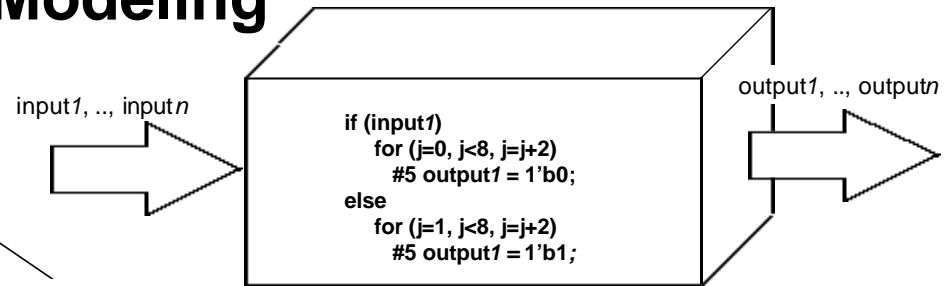
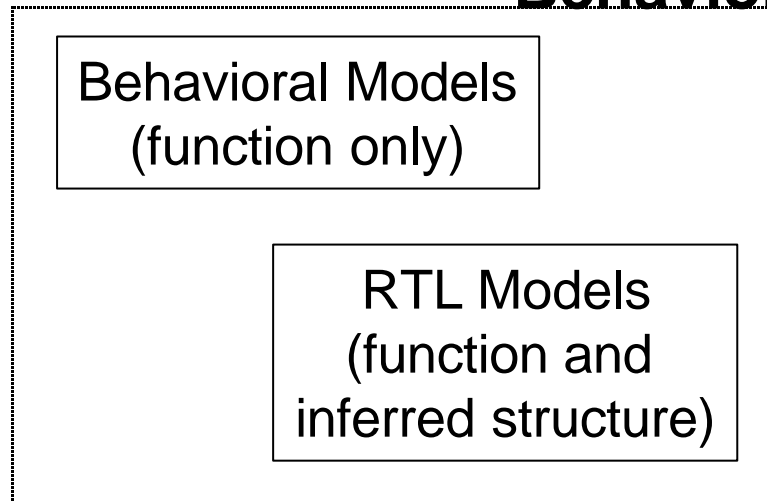
```
module fl_add4(fco, fsum, cin, a, b);  
  
output [3:0] fsum;  
output fco;  
input [3:0] a, b;  
input cin;  
  
reg [3:0] fco, fsum;  
  
always @(cin or a or b)  
    {fco, fsum} = cin + a + b;  
  
endmodule
```

■ Will produce the same logical model and functionality

Structural Modeling

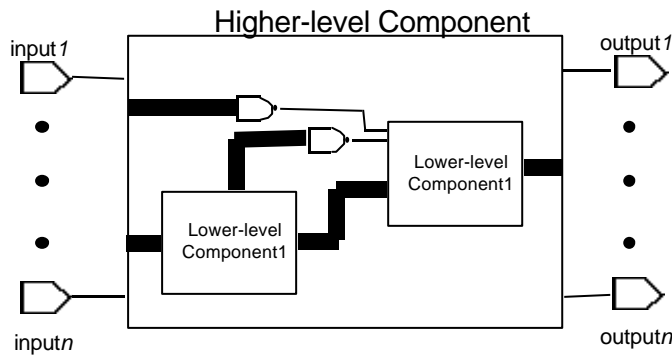
Levels of Abstraction

Behavioral Modeling



Abstract

Detailed **Structural Modeling**



Gate Level Models
(function and
structure)

Switch Level Models
(function and
structure)

Structural Modeling













- Defines function and structure of a digital circuit
- Adds to Hierarchy

Verilog Structural Modeling

- Component Level Modeling - instantiating user-created lower-level designs (components)
- Gate Level Modeling - instantiating Verilog built-in gate primitives
 - and, nand, or, nor, xor, xnor
 - buf, bufif0, bufif1, not, notif0, notif1
- Switch Level Modeling - instantiating Verilog built-in switch primitives
 - nmos, rnmos, pmos, rpmos, cmos, rcmos
 - tran, rtran, tranif0, rtranif0, tranif1, rtrainif1, pullup, pulldown
 - ⇒ Switch level modeling will not be discussed

Verilog Structural Modeling

- Verilog has predefined gate primitives

Primitive	Name	Function	Primitive	Name	Function
	and	n-input AND gate		buf	n-output buffer
	nand	n-input NAND gate		not	n-output buffer
	or	n-input OR gate		bufif0	tristate buffer lo enable
	nor	n-input NOR gate		bufif1	tristate buffer hi enable
	xor	n-input XOR gate		notif0	tristate inverter lo enable
	xnor	n-input XNOR gate		notif1	tristate inverter hi enable

Instantiation of Gate Primitives

■ Instantiation Format:

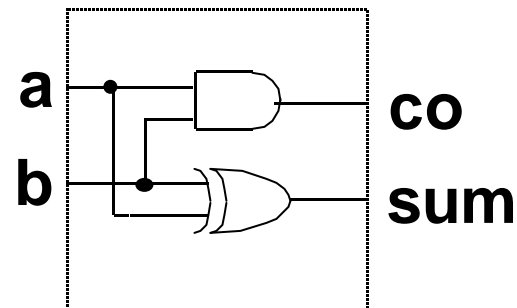
<gate_name> #<delay> <instance_name> (port_list);

- ***<gate_name>*** - The name of gate
- ***#delay*** - The gate delay
 - Not required, used for simulation
- ***<instance_name>*** - Any name that you want
 - Not required for Verilog gate primitives
- ***(port_list)*** - The port connection

Connecting ports by ordered list

- For Verilog gate primitives, the first port on the port list is the output, followed by the inputs.
 - **<gate_name>**
 - and
 - xor
 - **#delay** - OPTIONAL
 - 2 time unit for the and gate
 - 4 time unit for the xor gate
 - **<instance_name>** - OPTIONAL
 - u1 for the and gate
 - u2 for the xor gate
 - **(port_list)**
 - (co, a, b) - (output, input, input)
 - (sum, a, b) - (output, input, input)

```
module half_adder (co, sum, a, b);  
  
    output co, sum;  
    input a, b;  
  
    parameter and_delay = 2;  
    parameter xor_delay = 4;  
  
    and #and_delay u1(co, a, b);  
    xor #xor_delay u2(sum, a, b);  
  
endmodule
```



User-Defined Primitives (UDP)

- Allows users to define their own primitives
- Defined as truth tables
- Both combinatorial and clocked logic may be represented in the truth table
- Once defined, a UDP is used exactly the same as a built-in primitive
- Characteristics:
 - Only 1 output
 - Must have at least 1 input but no more than 10

UDP - Latch

```
primitive latch (q, clock,data); // Level sensitive, active low
output q;
reg q;
input clock, data;
initial q = 1'b0; // Output is initialized to 1'b0.
                    // Change 1'b0 to 1'b1 for power up Preset

table
    // clock data current state next state
        0      1      :?:      1;
        0      0      :?:      0;
        1      ?      :?:      -; // '-' = no change
endtable
endprimitive
```

UDP - Register

```
primitive d_edge_ff (q, clock,data); //edge triggered, active high
output q;
reg q;
input clock, data;
initial q = 1'b0;    //Output is initialized to 1'b0.
                    //Change 1'b0 to 1'b1 for power up Preset

table
// clk data state next
(01)  0  :?:   0;
(01)  1  :?:   1;
(0x)  1  :1:   1;
(0x)  0  :0:   0;
(?0)  ?   :?:  -; // ignore negative edge of the clock
?     (??) :?:  -; // ignore data changes on clock levels
endtable
endprimitive
```

Instantiation of lower-level Components

■ Instantiation Format:

<component_name> #<delay> <instance_name> (port_list);

- ***<component_name>*** - The name of your lower-level component
- ***#delay*** - The component delay
 - Not required, used for simulation
- ***<instance_name>*** - Any name that you want
 - Required, unlike Verilog gate primitives
- ***(port_list)*** - The port connection

Connecting ports by ordered list or by name

- For user-created lower-level components, the port connection is defined by the module declaration's port list order.

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

- Therefore for the first half_adder :
- The ports are connected by ordered list
- The order of the port connection does matter
 - **co -> c1, sum -> s1, a -> a, b -> b**

- For the second half_adder:

- The ports are connected by name
- The order of the port connection does not matter
 - **a -> s1, b -> cin, sum -> fsum, co -> c2**

```
module full_adder(fco, fsum, cin, a, b);
```

```
output fco, fsum;
```

```
input cin, a, b;
```

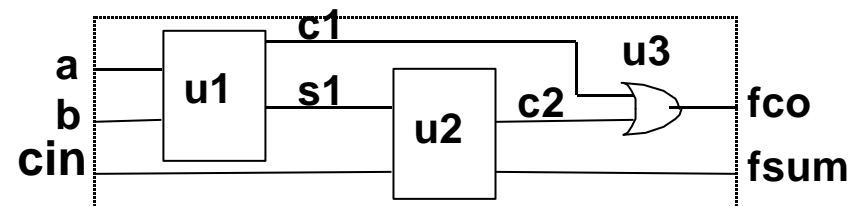
```
wire c1, s1, c2;
```

```
half_adder u1(c1, s1, a, b);
```

```
half_adder u2(.a(s1), .b(cin),  
               .sum(fsum), .co(c2));
```

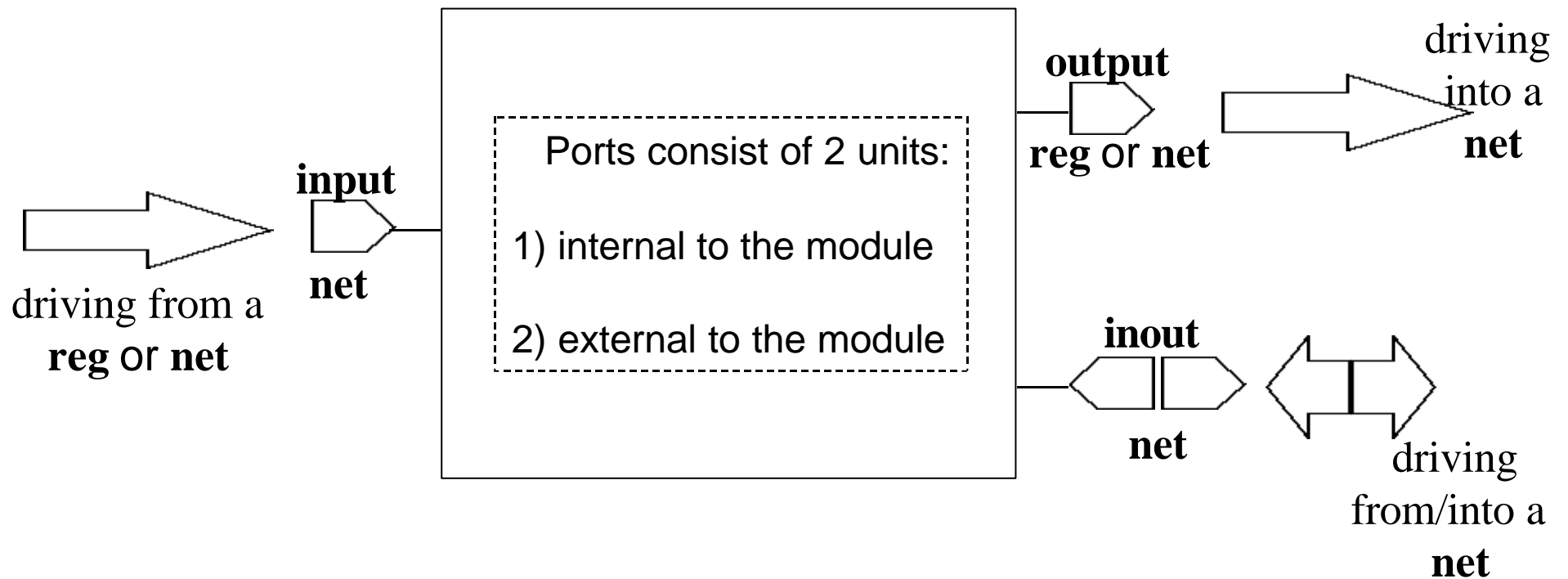
```
or u3(fco, c1, c2);
```

```
endmodule
```



Port Connection Rules

Port connections when modules
are instantiated within other
modules



Defparam

- Used to change the value of a lower-level component parameter(s)

```
module full_adder(fco, fsum, cin, a, b);
```

```
output fco, fsum;
```

```
input cin, a, b;
```

```
wire c1, s1, c2;
```

```
defparam u1.and_delay = 4, u1.xor_delay = 6;
```

```
defparam u2.and_delay = 3, u2.xor_delay = 5;
```

```
half_adder u1(c1, s1, a, b);
```

```
half_adder u2(.a(s1), .b(cin),  
               .sum(fsum), .co(fco));
```

```
or u3(fco, c1, c2);
```

```
endmodule
```

Simulation Time

- Simulation Time is the same for all modules during a simulation run
 - Simulation starts at time 0
 - Simulation time advances when all processes at current time are simulated

Gate Delays

- Rise Delay - transition from 0, x, or z to a 1
- Fall Delay - transition from 1, x, or z to a 0
- Turn-off Delay - transition from 0, 1 or x to a z

```
<component_name> #(Rise, Fall, Turnoff) <instance_name> (port_list);
```

```
and #(2)      u1 (co, a, b); // Delay of 2 for all transitions  
and #(1, 3)   u2 (co, a, b); // Rise = 1, Fall = 3  
bufif0 #(1, 2, 3) u3 (out, in, enable) ;  
                // Rise = 1, Fall = 2, Turn-off = 3
```


Min/Typ/Max Values

- Min Value - the minimum delay that you expect the gate to have
- Typ Value - the typical delay that you expect the gate to have
- Max Value - the maximum delay that you expect the gate to have

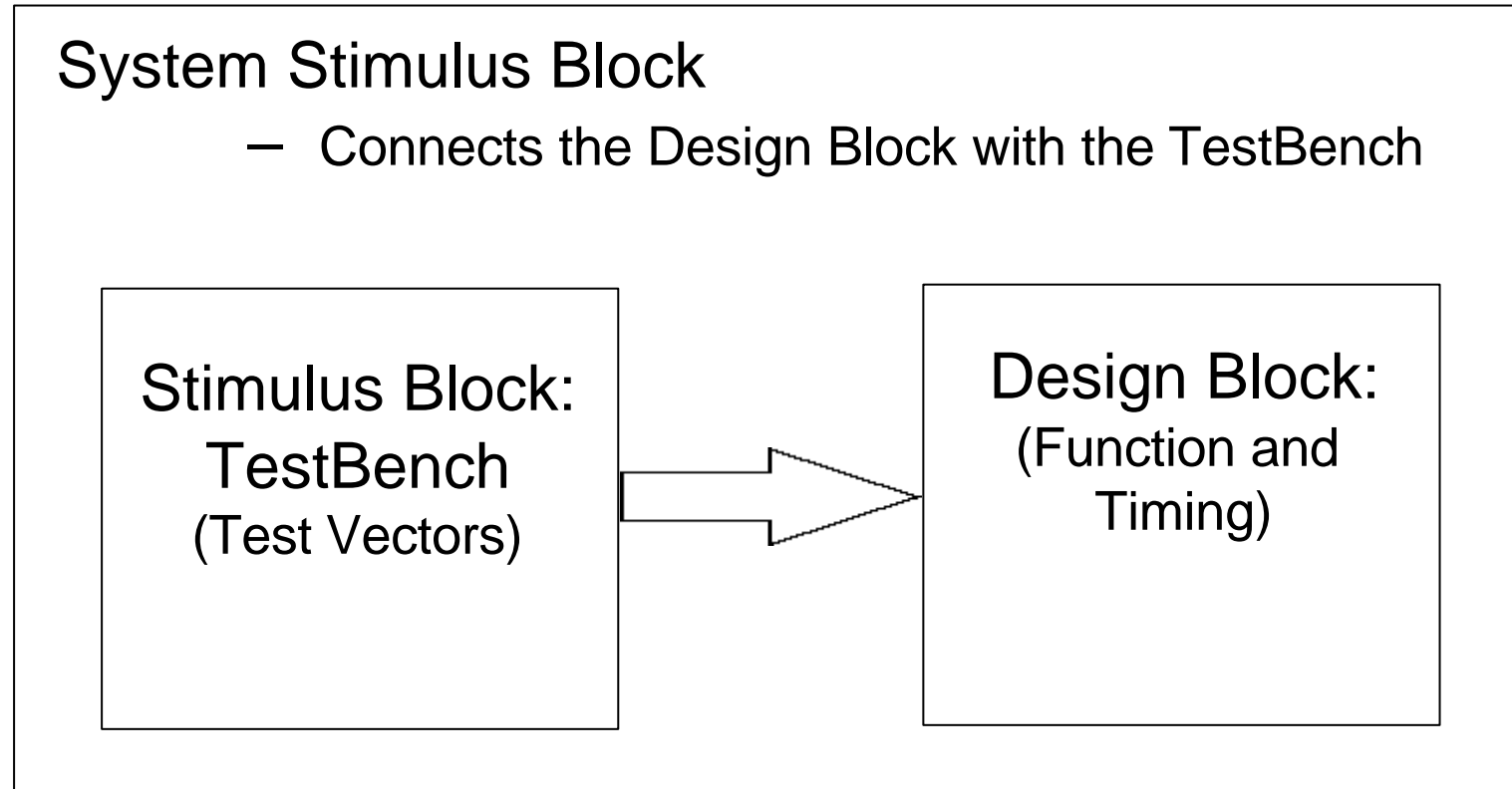
#(Min:Typ:Max, Min:Typ:Max, Min:Typ:Max)

```
and #(1:2:3)          u1 (co, a, b);  
and #(1:2:3, 1:2:3)   u2 (co, a, b);  
bufif0 #(2:3:4, 2:3:4, 3:4:5) u3 (out, in, enable) ;
```

Verilog Summary

Verilog Environment

- Contains a Design Block and a Stimulus Block



Verilog - Design Block

```
module counter(q, clk, clr, f, in);  
input clk, clr;  
input [1:0] f;  
input [7:0] d;  
output [7:0] q;  
reg [7:0] q;  
parameter set = 4, hold = 1;  
  
clock_gen #(100, 50) clock(clk);  
  
always @(posedge clk or posedge clr)  
  begin  
    if (clr)  
      q = 8'h00;
```

```
  else  
    case (f)  
      2'b00: q = d; // Loads the counter  
      2'b01: q = q + 1; // Counts up  
      2'b10: q = q - 1; // Counts down  
      2'b11: q = q;  
    endcase  
  end  
  
  specify  
    $setup (d, posedge clk, set);  
    $hold (posedge clk, d, hold);  
  endspecify  
  
endmodule
```

Verilog - Stimulus Block

```
module counter_test(clrd, fd, ind, clkd);  
input clkd;  
output clrd;  
output [1:0] fd;  
output [7:0] ind;  
reg clrd;  
reg [1:0] fd;  
reg [7:0] ind;  
clock_gen #(100, 50) clockd(clkd);  
always @(posedge clkd) begin  
    clrd=1; fd=0; ind=0;  
    #100 clrd=1; fd=0; ind=0;  
    #100 clrd=0; fd=0; ind=8'b01010101;  
    #100 clrd=0; fd=3; ind=8'b11111111;  
    #100 clrd=0; fd=1; ind=8'b10101010;  
    #100 clrd=0; fd=2; ind=8'b11001100; end  
endmodule
```

Verilog - System Stimulus Block

```
module counter_system;
wire clear, sys_clock;
wire [1:0] func;
wire [7:0] datain, result;

clock_gen #(100,50) system_clock(sys_clock);
counter #(4, 1) op(.clr(clear), .clk(sys_clock), .f(func), .d(datain), .q(result));
counter_test op_test(.clrd(clear), .clkd(sys_clock), .fd(func), .ind(datain));

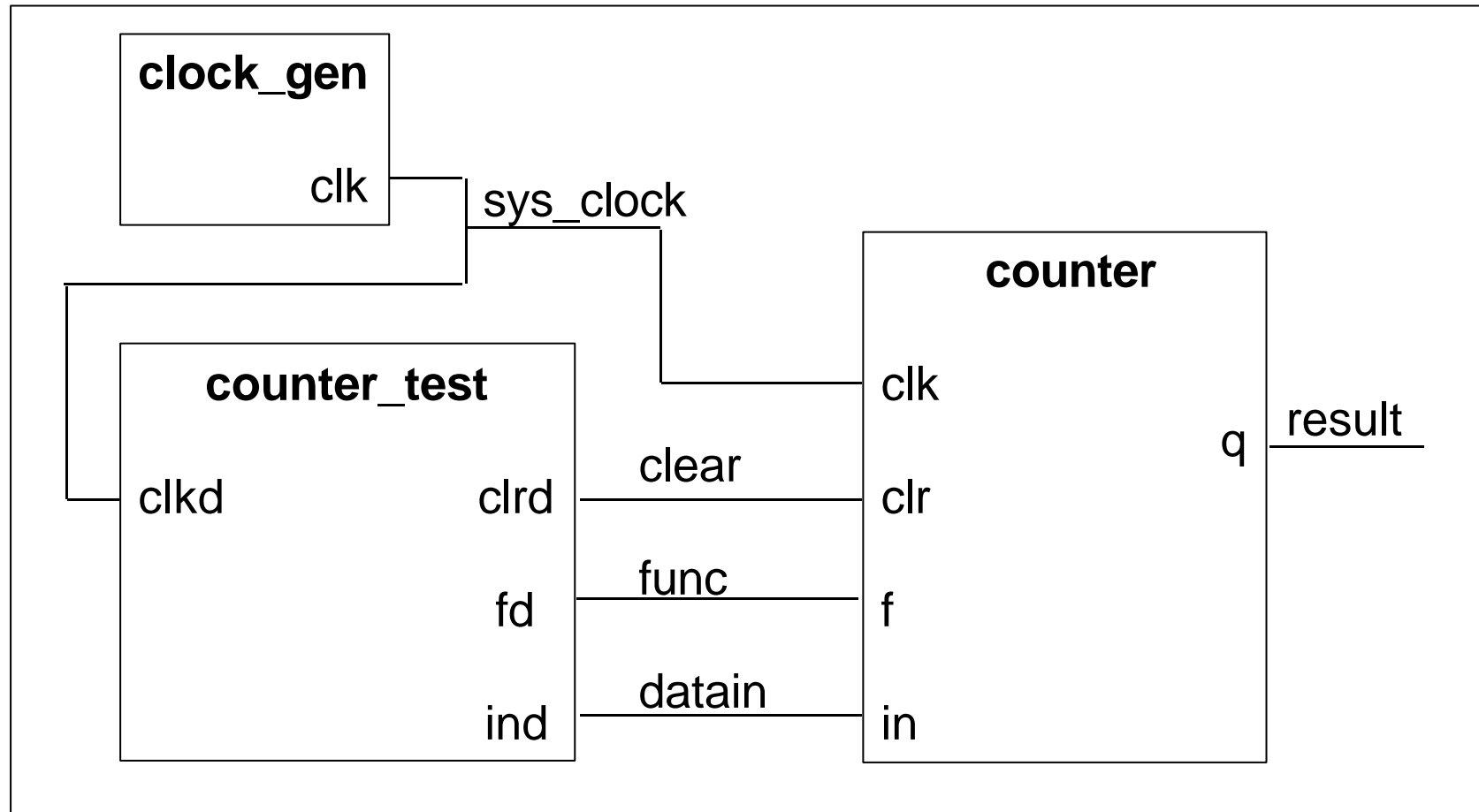
initial begin
    $display("\t\t\t Time\t\t clear\t\t sys_clock\t\t func\t\t datain\t\t result");
    $monitor($time,clear,,,,,,,,,sys_clock,,,,,,,,,func,,,,,,,,,datain,,,,,,,,result);

    #1300 $finish; end

endmodule
```

Verilog Environment

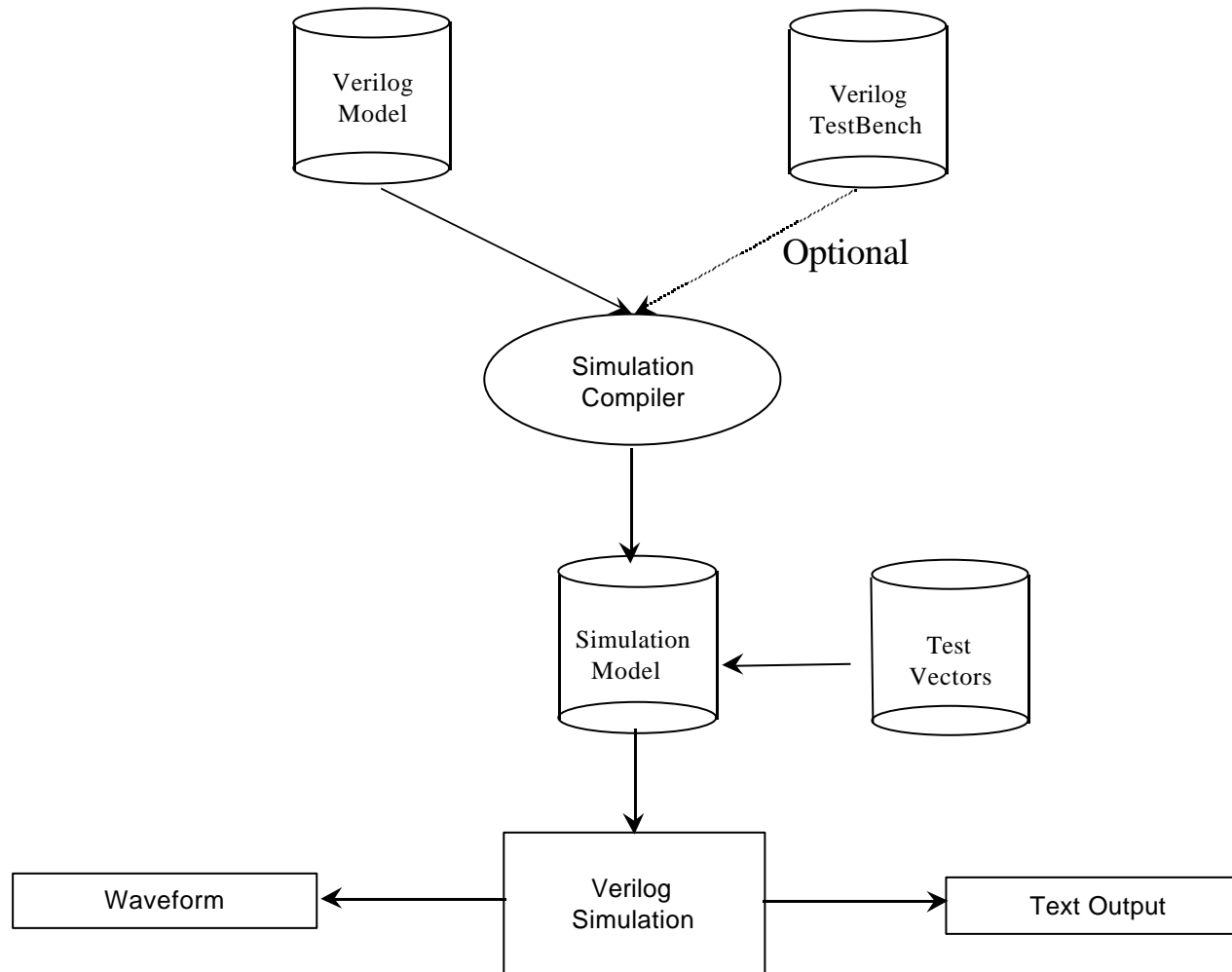
counter_system



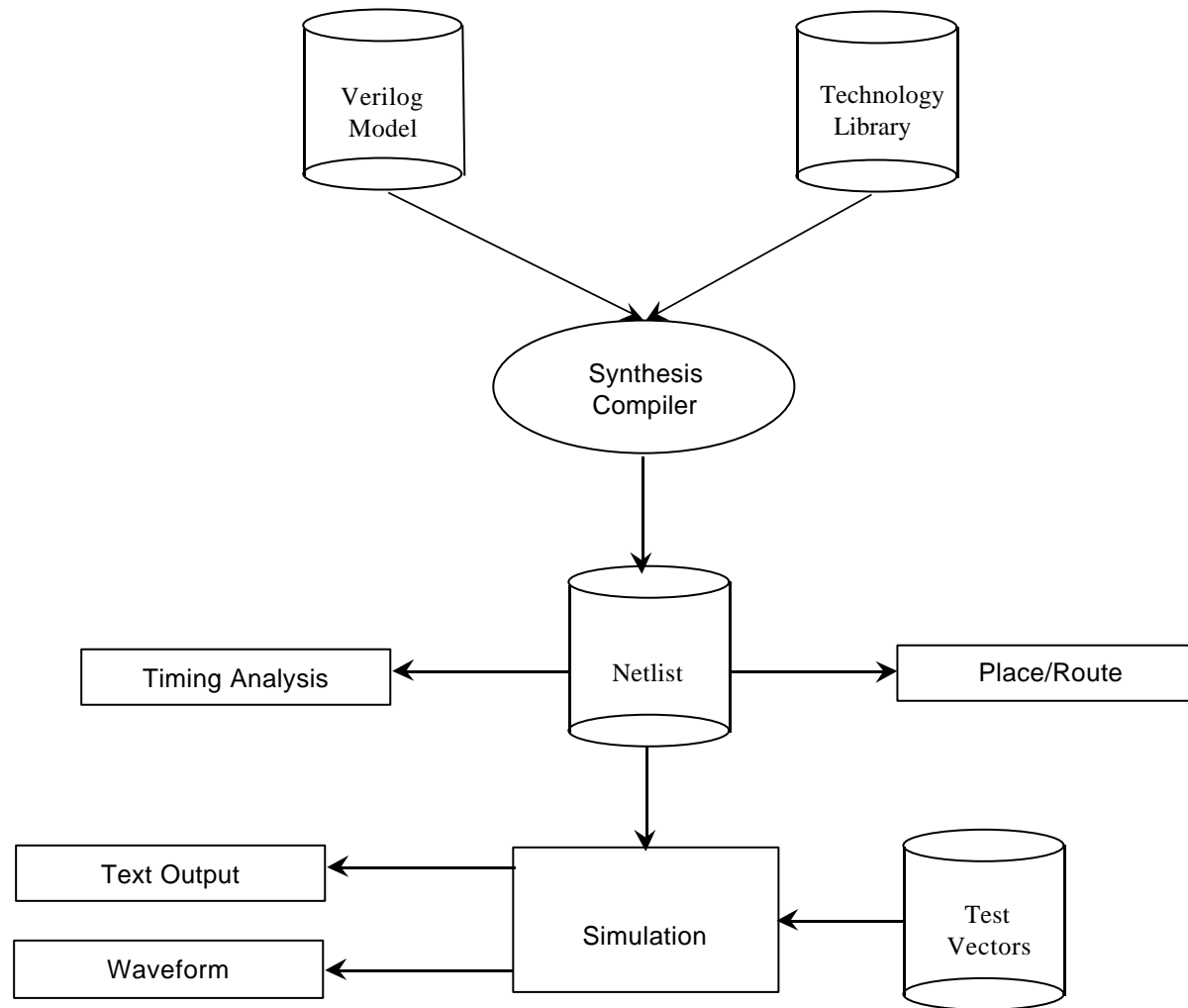
Verilog

- Used for both simulation and synthesis
 - Two sets of constructs:
 - Simulation Constructs
 - Synthesis Constructs
 - Supported by synthesis tools

Typical Simulation Design Flow



Typical Synthesis Design Flow



Verilog

Synthesis

■ Functionality

```
module dff ( d, clk, q);  
  
input d, clk ;  
output q;  
  
wire d, clk;  
reg q ;  
  
always @(posedge clk)  
    q = d ;  
endmodule
```

Simulation

■ Functionality and Timing

```
module dff ( d, clk, q);  
  
input d, clk ;  
output q;  
  
wire d, clk;  
reg q ;  
  
always @(posedge clk)  
    q = d ;  
specify  
    $setup (d, posedge clk, set);  
    $hold (posedge clk, d, hold);  
endspecify  
  
endmodule
```

Appendix

System Tasks and Functions

- Defined by (dollar sign) \$<keyword>
- Located before the **module** declaration

- \$stop; - stop simulation
- \$finish; - quit simulation
- \$display(...); - display value
- \$monitor(...); - monitor value
- \$time; - current simulation time

Compiler Directives

- Defined by (back tick) ``<keyword>`
- Located before the **module** declaration
- ``timescale <reference_time_unit> / <time_precision>`
 - `<reference_time_unit>` specifies the unit of measurement for times and delays
 - `<time_precision>` specifies the precision to which the delays are rounded off during simulation
 - Only 1, 10, and 100 are valid integers for specifying time unit and time precision
 - Example: ``timescale 1 ns / 10 ps`

Compiler Directives

- **`define** - assigns a constant value
 - Example: **`define** SIZE 32
 - Difference between **parameter**:
 - **parameter** has global visibility
 - **`define** is visible locally to that macrofunction
- **`include** - includes entire contents of a Verilog source file
 - Example: **`include** test.v
 - test.v is a separate design file

Conditional Compilation

```
// Conditional Compilation

`ifdef TEST // Compile module counter only if text macro TEST is defined
module counter;
...
..
endmodule
`else // Compile the module counter_test as default
module counter_test;
...
...
endmodule
`endif
```

Timing Specifications

Specify Blocks

- Path Delay - the delay between a source (input or inout) pin and a destination (output or inout) pin
- Path Delays are assigned in Specify Blocks with the keywords `specify` and `endspecify`
- Statements in a Specify Block can do the following:
 - Assign pin-to-pin timing delays
 - Set up timing checks in the circuits
 - Define `specparam` constants
- Alternative to the **#<delay>** construct

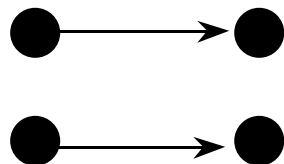
Parallel Connection

- Parallel Connection - specified by the symbol (\Rightarrow)
- Format:

$\langle source \rangle \Rightarrow \langle destination \rangle = \langle delay \rangle$

// Single bit a and b
 $a \Rightarrow b = 5;$

Source Destination



// 2-bit Vector a and b
 $a \Rightarrow b = 5;$

is equivalent to

$a[0] \Rightarrow b[0] = 5;$
 $a[1] \Rightarrow b[1] = 5;$

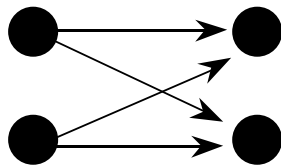
Full Connection

- Full Connection - specified by the symbol ($\ast>$)
 - Each bit in the source is connected to each bit in the destination

■ Format: $\langle source \rangle \ast> \langle destination \rangle = \langle delay \rangle$

Source

Destination



// 2-bit Vector a and b
 $a \ast> b = 5;$

is equivalent to

$a[0] \ast> b[0] = 5;$

$a[1] \ast> b[1] = 5;$

$a[0] \ast> b[0] = 5;$

$a[1] \ast> b[1] = 5;$

Specparam

- Specparam - assigning a value to a symbolic name for a timing specification
 - Similar to parameter but used in specify blocks

specify

specparam a_to_b = 5;

a ==> b = a_to_b;

end specify

Rise, Fall, Turn-off and Min/Typ/Max Values

specify

specparam rise = 4:5:6;

specparam fall = 6:7:8;

specparam turnoff = 5:6:7;

a => b = (rise, fall, turnoff);

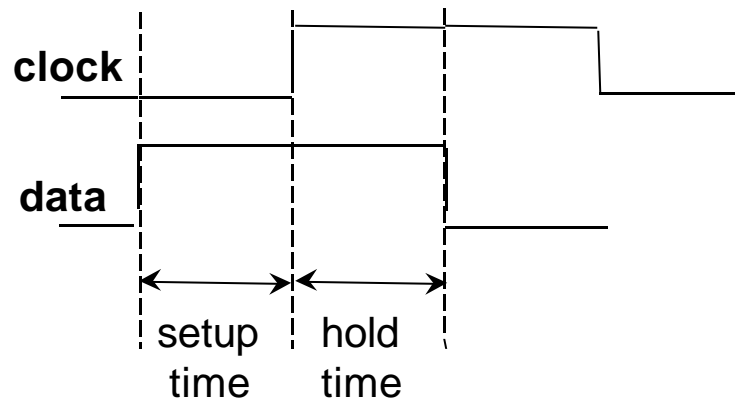
end specify

Timing Checks

- **\$setup** task - system task that checks for the setup time

```
$setup(data_event, reference_event, limit);
```

- data_event - monitored for violations
- reference_event - establishes a reference for monitoring the data_event signal
- limit - minimum time for setup



- **\$hold** task - system task that checks for the hold time

```
$hold(reference_event, data_event, limit);
```

- reference_event - establishes a reference for monitoring the data_event signal
- data_event - monitored for violations
- limit - minimum time for hold

specify

```
$setup (ina, posedge clk, set);  
$hold (posedge clk, ina, hld);  
$setup (inb, posedge clk, set);  
$hold (posedge clk, inb, hld);
```

endspecify

OVI Synthesis Guidelines

- Fully supported constructs for all synthesis tool

Verilog Construct

Module

Module Instantiations

Port Declarations

Net Data Types

wire, tri, supply1, supply0

Register Data Types

reg, integer

Parameter Constants

Integer values

Function and Tasks

begin and end statements

disable of name statement groups

if, if-else, case, casex, casez

Blocking

procedural and continuous

OVI Synthesis Guidelines

- Partially support constructs
- Constructs used with certain restrictions
- Could vary among synthesis tools

Partially Supported

always (always @)

Edge-sensitive

posedge and negedge

for Loop

Bit and part select of vectors

Procedural

non blocking (<=)

Operators (*logical, equalit, relational, reduction*

arithmetic, shift, concatenate, replicate

conditional)

Operator by the power of 2 only

(multiply, divide, modulo)

OVI Synthesis Guidelines

- Optional Ignore constructs are not supported by all synthesis tools.
- Synthesis should ignore construct

Optional-Ignored

Timing and delays (#)

Specify Block

System tasks or functions (\$)

OVI Synthesis Guidelines

- Optional-Aborts are not supported by all synthesis tools.
- Synthesis must abort when encountered

Optional-Abort

Any partially support construct used in a non-supported way.

Net Types

wand, triand, wor, prior, tri0, tri1, trireg

Loops

forever, repeat, while

Identify operators (`===` `!=="`)

wait

initial

fork-join

Procedural deassign

force, release

User Defined Primitives

Some Built-in Primitives