Introduction to Verilog

intel

1

Intel® FPGA Products

FPGA Mid-range FPGAs SoC & Transceivers	FPGA Optimized for High Bandwidth	FPGA Optimized for High Bandwidth	FPGA Cost/Power Balance SoC & Transceivers	FPGA/CPLD Lowest Cost, Lowest Power

Resources

Embedded Soft and Hard Processors	Design Software	Acceleration Boards & Development Kits	Intellectual Property (IP)
Nios® II ARM*	Intel® Quartus® Prime Design Software Intel® FPGA SDK for OpenCL™		▪ Industrial ▪ Computing ▪ Enterprise

intel 2

2

Course Objectives

- Implement Basic Constructs of Verilog
- Implement Modeling Structures of Verilog

3

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
  - Compiler Directives and System Tasks/Functions

4

5

What is Verilog?

- IEEE industry standard Hardware Description Language (HDL) - used to describe system.
- Use in both hardware simulation & synthesis

6

Verilog History

- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence Design Systems purchased Gateway (Verilog-XL simulator)
- 1990 Cadence Design Systems released Verilog to the public
- Open Verilog International (OVI) was formed to control the language specifications
- 1993 OVI released version 2.0
- 1995 IEEE accepted OVI Verilog as a standard, Verilog 1364
- 2001 IEEE revised standard
- 2005 IEEE accepted new revision for the standard

7

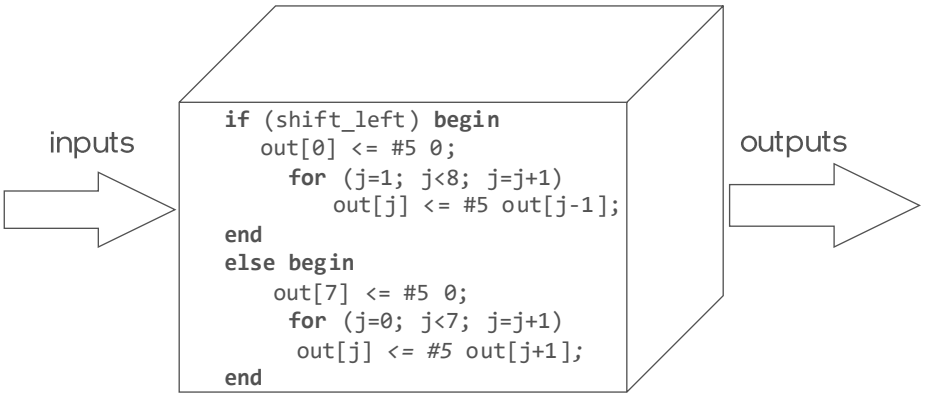
Verilog HDL Terminology

- HDL: 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

8

Behavior Modeling

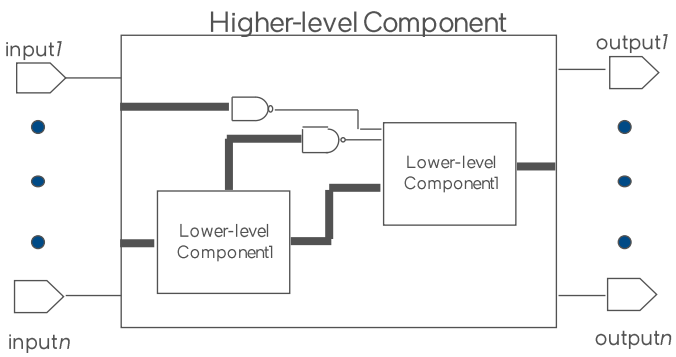
- Only the functionality of the circuit, no structure
- Synthesis tool creates correct logic



9

Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware



10

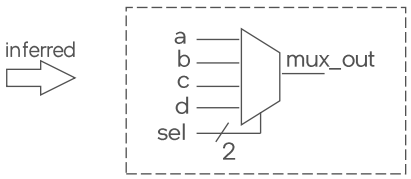
More Terminology

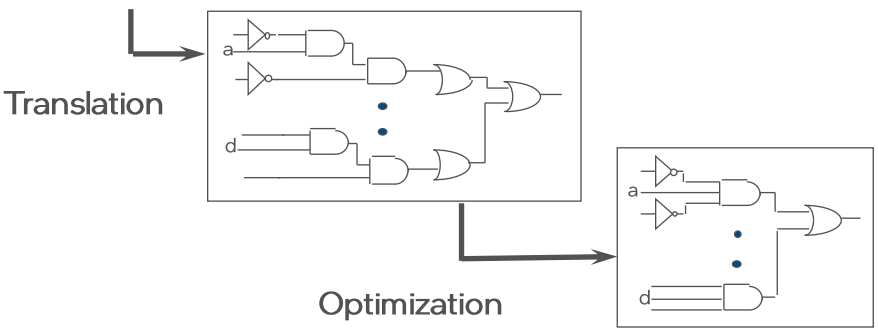
- Register Transfer Level (RTL): A type of behavioral modeling, for the purpose of synthesis
  - Hardware is implied or inferred
  - Synthesizable
- Synthesis: Translating HDL to a circuit and then optimizing the represented circuit
- RTL Synthesis: Translating an RTL model of hardware into an optimized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

11

RTL Synthesis

```
always @(a, b, c, d, sel)
   case (sel)
      2'b00: mux_out = a;
      2b'01: mux_out = b;
      2b'10: mux_out = c;
      2'b11: mux_out = d;
   endcase
```





12

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"

13

Verilog vs. Other HDL Standards (cont.)

- Verilog
  - `always @ (posedge clk)`
    `q<=d;`
  - Result: Verilog Synthesis provides a positive edge-triggered flip-flop
- ABEL, PALASM, AHDL
  - `DFF(d,q,clk)`
  - Result: ABEL, PALASM, AHDL synthesis provides a D-type flip-flop. The sense of the clock depends on the synthesis tool

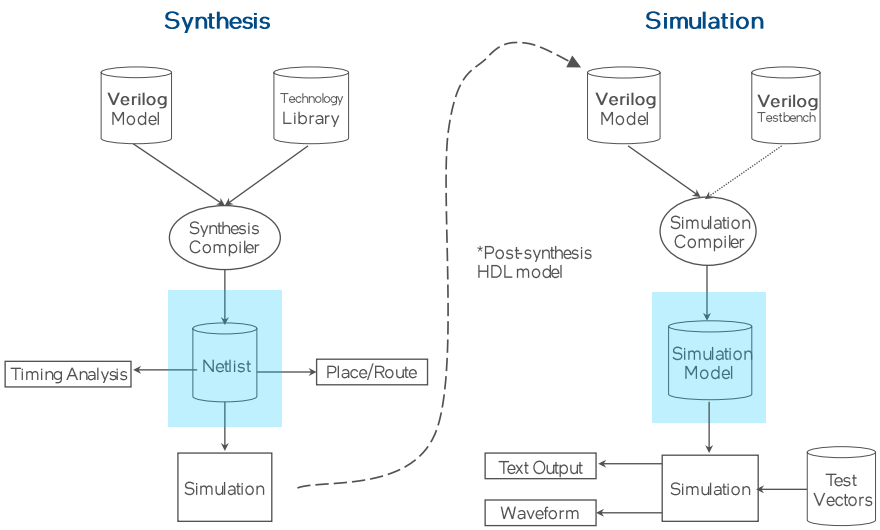
14

Simulation vs. Synthesis

- The Verilog language has two sets of constructs
  - Simulation
  - Synthesis
- Most (but not all) simulation constructs are synthesizable
- Check help or documentation for your synthesis tool to be sure of the supported constructs

15

Typical RTL Synthesis and RTL Simulation Flows



16

Introduction to Verilog

Verilog Modeling

17

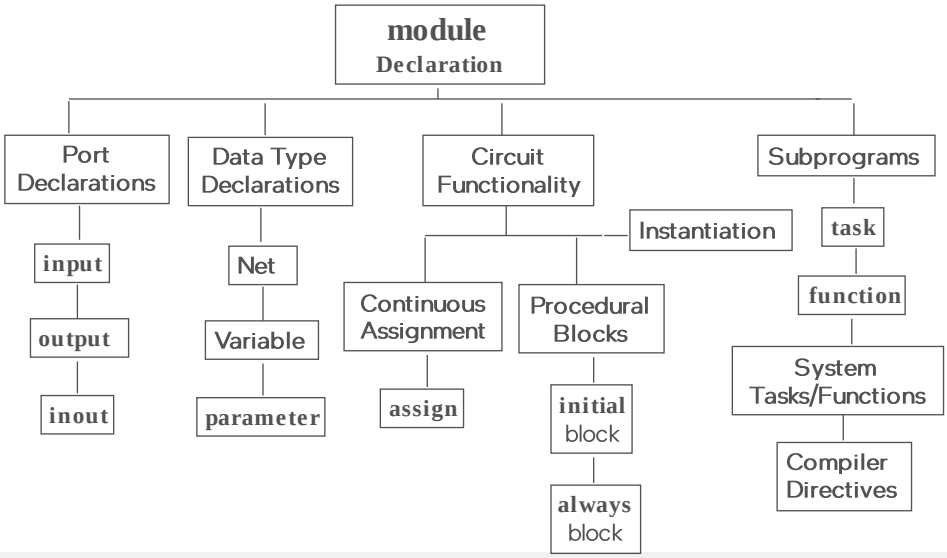
Verilog – Basic Modeling Structure

```
module module_name (port_list);
    port declarations
    data type declarations
    circuit functionality
    timing specifications
endmodule
```

- Begins with keyword **module** & ends with keyword **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 (not discussed)

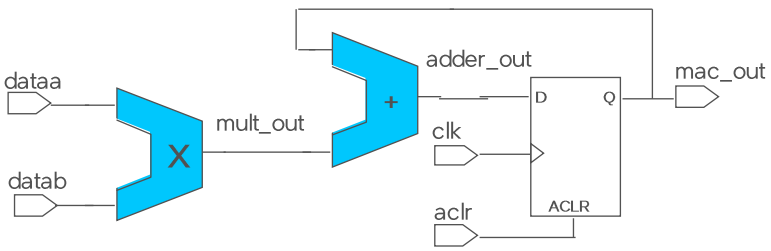
18

Components of a Verilog Module



19

Schematic Representation - MAC



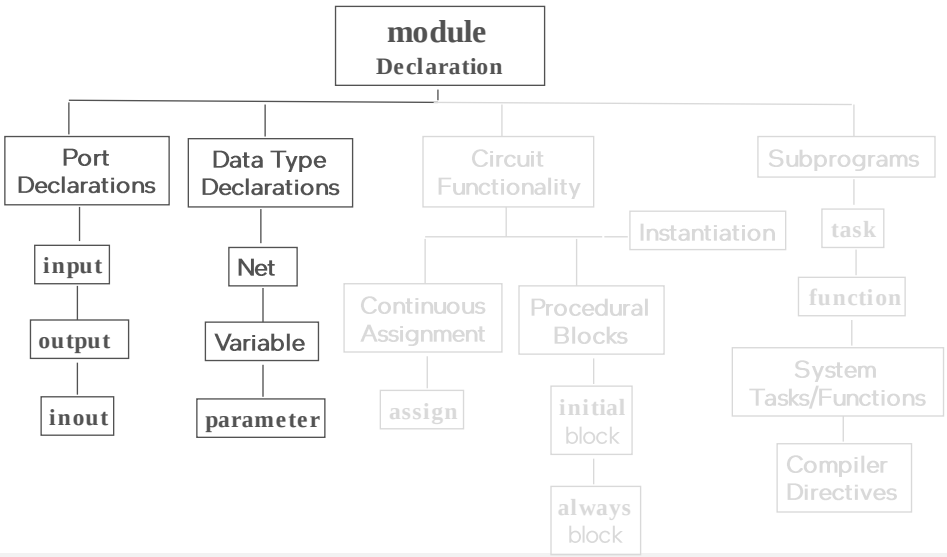
20

Verilog Model: Multiplier-Accumulator (MAC)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```


21

Let's take a look at



22

Module Declaration

- Begins with keyword **`module`**
- Provides the Verilog block (module) name
- Includes port list, if any
  - A listing of all module I/O names

```
module mult_acc (mac_out, dataa, datab, clk, aclr);
```


23

Port Declaration

- Defines the names, sizes, types & directions for all ports
- Format

```
<port_type> port_name;
```

- Example

```
input [7:0] dataa, datab;
input clk, aclr;
output [15:0] mac_out;
```

- Port types
  - **`input`** ⇒ input port
  - **`output`** ⇒ output port
  - **`inout`** ⇒ bidirectional port

24

Verilog '2001 & Later Module/Port Declaration

- Beginning in Verilog '2001, module and port declarations can be combined
  - More concise declaration section
  - Parameters (shown later) may also be included

```
module mult_acc (
  input [7:0] dataa, datab,
  input clk, aclr,
  output [15:0] mac_out
);
```


25

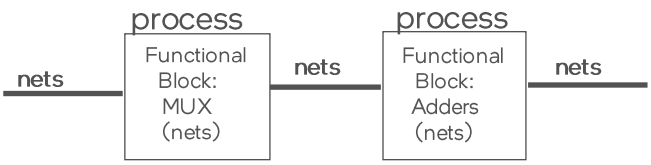
MAC (Module & Port Declarations)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```

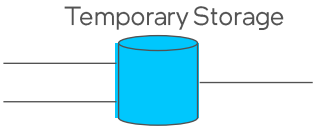

26

Data Types

- **Net data type** - represents physical interconnect between structures (activity flows)



- **Variable data type** - represents element to store data temporarily



27

Net Data Type & Net Arrays

- **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;

28

Net Data Types

Net Data Types	Functionality	Synthesis Support?
wire	Used for interconnect	Y
tri		Y
supply0	Represents constant value (i.e. power supply)	Y
supply1		Y
wand	Represents wired logic	Y
triand		Y
wor		Y
trior		Y
tri0	Tri-state node with pull-up/pull-down	Y
tri1		Y
trireg	Stores last value when not driven	N

*Note: There is no functional difference between **wire** & **tri**; **wand** & **triand**; **wor** & **trior***

29

Variable Data Types & Variable Arrays

- **reg**(1) - unsigned variable of any bit size
  - Use **reg signed** for a signed implementation(2)
- **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;

30

Variable Data Types

Variable Data Types	Functionality	Synthesis Support?
reg	Unsigned variable (by default) Use **`reg signed`** for signed representation	Y
integer	Signed variable (usually 32 bits)	Y
time	Unsigned integers (usually 64 bits) used for storing and manipulating simulation time	N
real	Double precision floating point variable	N
realtime	Double-precision floating point variable used with time	N

31

Memory

- Multi-dimensional variable array
  - Can not be a net type
- Examples:

```
reg [31:0] mem[0:1023];  // 1Kx32
reg [31:0] instr;
…
instr = mem[2];
mem [1000][5:0] = instr[5:0]; // Unsupported by synthesis
```

- Cannot write to multiple elements in one assignment

```
mem = 32'd0;
```

Illegal!!!

32

Parameter

- Value assigned to a symbolic name
- Must resolve to a constant at compile time
- Can be overwritten at compile time
  - Exception: Local parameters (**localparam**)
  - Discussed later

```
parameter size = 8;

reg [size-1:0] dataa, datab;
```


33

Verilog '2001 & later Module/Port/Parameter Declaration

- Module, port and parameter declarations can be combined
  - Illegal for local parameters

```
module mult_acc
  #(parameter size = 8)
  (
  input [size-1:0] dataa, datab,
  input clk, clr,
  output [(size*2)-1:0] mac_out
);
```

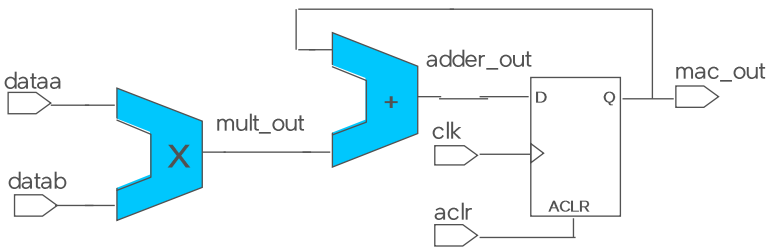

34

Data Type

- Every signal (which includes ports) must have an assigned data type
- Data types for signals must be explicitly declared in the declarations of your module
- Ports are **wire** (net) data types by default if not explicitly declared

35

Schematic Representation - MAC



36

MAC (Data Type Declarations)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```


37

Introduction to Verilog

Assigning Values – Numbers & Operators

38

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 Formats
  - 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
  - Signed ('s' or 'S') 16'shFA = signed 16-bit hex value

39

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

40

Number Extension

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

41

Short Quiz

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

42

Short Quiz Answer

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

  4'b0001, MSB is truncated (10001)

43

Operators

- Arithmetic
- Bitwise
- Reduction
- Relational
- Equality
- Logical
- Shift
- Miscellaneous

*Used in **expressions** to describe model behavior*

44

Arithmetic Operators

Operator Symbol	Functionality	Examples ain = 5 ; bin = 10 ; cin = 2'b01 ; din = 2'b0z		
+	Add, Positive	bin + cin ⇒ 11	+bin ⇒ 10	ain + din ⇒ x
–	Subtract, Negate	bin – cin ⇒ 9	-bin ⇒ -10	ain – din ⇒ x
*	Multiply	ain * bin ⇒ 50		
/	Divide*	bin / ain ⇒ 2		
%	Modulus	bin % ain ⇒ 0		
**	Exponent*	ain ** 2 ⇒ 25		

- Treats vectors as a whole value
- Results unknown if any operand is Z or X
- Carry bit(s) handled automatically if result wider than operands
- Carry bit lost if operands and results are same size

45

Bitwise Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
~	Invert each bit	~ain ⇒ 3b'010	~cin ⇒ 3'b10x
&	AND each bit	ain & bin ⇒ 3'b100	bin & cin ⇒ 3'b010
|	OR each bit	ain | bin ⇒ 3'b111	bin | cin ⇒ 3'b11x
^	XOR each bit	ain ^ bin ⇒ 3'b011	bin ^ cin ⇒ 3'b10x
^~ or ~^	XNOR each bit	ain ^~ bin ⇒ 3'b100	bin ~^ cin ⇒ 3'b01x

- Operates on each bit or bit pairing of the operand(s)
- Result is the size of the largest operand
- X or Z are both considered unknown in operands, but result maybe a known value
- Operands are left-extended if sizes are different

46

Reduction Operators

Operator Symbol	Functionality	Examples ain = 4'b1010 ; bin = 4'b10xz ; cin = 4'b111z		
&	AND all bits	&ain ⇒ 1'b0	&bin ⇒ 1'b0	&cin ⇒ 1'bx
~&	NAND all bits	~&ain ⇒ 1'b1	~&bin ⇒ 1'b1	~&cin ⇒ 1'bx
|	OR all bits	|ain ⇒ 1'b1	|bin ⇒ 1'b1	|cin ⇒ 1'b1
~|	NOR all bits	~|ain ⇒ 1'b0	~|bin ⇒ 1'b0	~|cin ⇒ 1'b0
^	XOR all bits	^ain ⇒ 1'b0	^bin ⇒ 1'bx	^cin ⇒ 1'bx
^~ or ~^	XNOR all bits	~^ain ⇒ 1'b1	~^bin ⇒ 1'bx	~^cin ⇒ 1'bx

- Reduces a vector to a single bit value
- X or Z are both considered unknown in operands, but result maybe a known value

47

Relational Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
>	Greater than	ain > bin ⇒ 1'b0	bin > cin ⇒ 1'bx
<	Less than	ain < bin ⇒ 1'b1	bin < cin ⇒ 1'bx
>=	Greater than or equal to	ain >= bin ⇒ 1'b0	bin >= cin ⇒ 1'bx
<=	Less than or equal to	ain <= bin ⇒ 1'b1	bin <= cin ⇒ 1'bx

- Used to compare values
- Returns a '1' bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

48

Equality Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
==	Equality	ain == bin ⇒ 1'b0	cin == cin ⇒ 1'bx
!=	Inequality	ain != bin ⇒ 1'b1	cin != cin ⇒ 1'bx
===	Case equality	ain === bin ⇒ 1'b0	cin === cin ⇒ 1'b1
!==	Case inequality	ain !== bin ⇒ 1'b1	cin !== cin ⇒ 1'b0

- Used to compare values
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- For equality/inequality, X or Z are both considered unknown in operands and result is always unknown
- For case equality/case inequality, X or Z are both considered distinct values and operands must match completely

49

Logical Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b000 ; cin = 3'b01x		
!	Expression not true	!ain ⇒ 1'b0	!bin ⇒ 1'b1	!cin ⇒ 1'bx
&&	AND of two expressions	ain && bin ⇒ 1'b0	bin && cin ⇒ 1'bx	
||	OR of two expressions	ain || bin ⇒ 1'b1	bin || cin ⇒ 1'bx	

- Used to evaluate single expression or compare multiple expressions
  - Each operand is considered a single expression
  - Expressions with a zero value are viewed as false (0)
  - Expressions with a non-zero value are viewed as true (1)
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

50

Shift Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b01x	
<<	Logical shift left	ain << 2 ⇒ 3'b100	bin << 2 ⇒ 3'bx00
>>	Logical shift right	ain >> 2 ⇒ 3'b001	bin >> 2 ⇒ 3'b000
<<<	Arithmetic shift left	ain <<< 2 ⇒ 3'b100	bin <<< 2 ⇒ 3'bx00
>>>	Arithmetic shift right	ain >>> 2 ⇒ 3'b111 (signed)	bin >>> 2 ⇒3'b000 (signed)

- Shifts a vector left or right some defined number of bits
- Left shifts (logical or arithmetic): Vacated positions always filled with zero
- Right shifts
  - Logical: Vacated positions always filled with zero
  - Arithmetic (unsigned): Vacated positions filled with zero
  - Arithmetic (signed): Vacated position filled with sign bit value (MSB value)
- Shifted bits are lost
- Shifts by values of X or Z (right operand) return unknown

51

Miscellaneous Operators

Operator Symbol	Functionality	Format & Examples
?:	Conditional test	`(condition) ? true_value : false_value` `sig_out = (sel == 2'b01) ? a : b`
{ }	Concatenate	ain = 3'b010 ; bin = 3'110 {ain,bin} ⇒ 6'b010110
{ { } }	Replicate	{3{3'b101}} ⇒ 9'b101101101

52

Operator Precedence

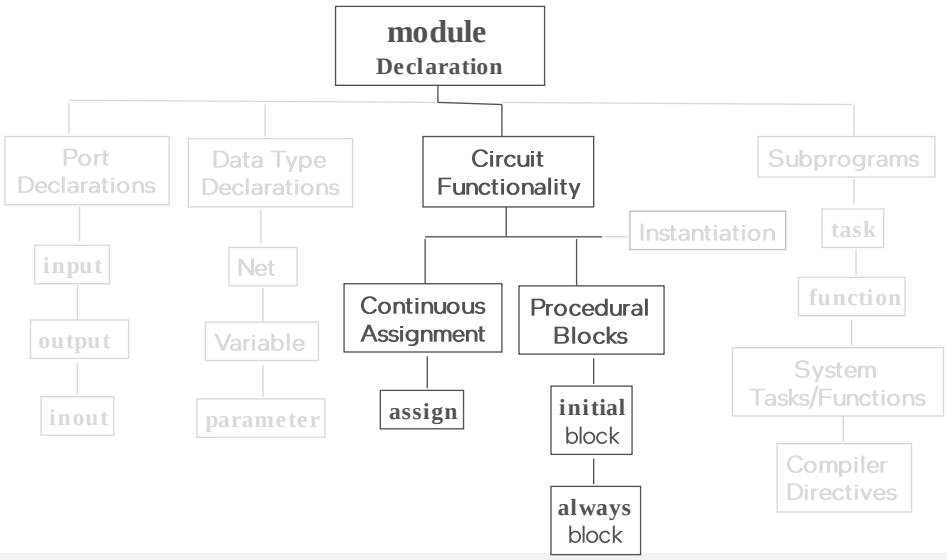
Operator(s)	Priority
+ - ! ~ & ~& etc. (unary* operators)	High
**	
* / %	
+ - (binary operators)	
<< >> <<< >>>	
< > <= >=	
== != === !==	
& (binary operator)	
^ ~^ ^~ (binary operators)	
| (binary operator)	
&&	
||	
?:	
{} {{}}	Low

- ( ) used to override default and provide clarity

** Unary operators have only one operand*

53

Let's take a look at



54

Introduction to Verilog

Behavioral Modeling –
Continuous Assignments

55

Continuous Assignments

- Model the behavior of combinatorial logic by using expressions and operators
  - Continuous assignments can be made when the net is declared

    OR

  - By using the **assign** statement

```
wire [15:0] adder_out = mult_out + out;
/*implicit continuous assignment */
```

is equivalent to

```
wire [15:0] adder_out;
assign adder_out = mult_out + out;
```


56

Continuous Assignments Characteristics

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

```
wire [15:0] adder_out = mult_out + out;
/*implicit continuous assignment */
```

is equivalent to

```
wire [15:0] adder_out;
assign adder_out = mult_out + out;
```

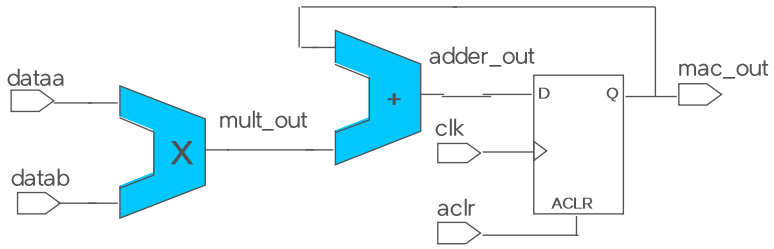

57

Continuous Assignment - Example

```
module and_func (
  input [7:0] ina, inb,
  output [7:0] out
);

assign out = ina & inb;

endmodule
```


58

Schematic Representation - MAC



59

MAC (Continuous Assignment)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```


60

Exercise 1

Please go to Exercise 1
in the Exercise Manual

61

Continuous Assignment Delay

- Use `#<value>` notation to delay updating LHS
  - Models propagation delay

```
assign #25 adder_out = mult_out + out;
```

- Behavior
  1) Operand on RHS changes
  2) RHS reads input(s) and performs expression
  3) If RHS ≠ LHS, LHS scheduled to be updated with new value after delay expires
  4) If another RHS operand changes before delay expires and a new value for LHS is calculated, then current value scheduled for LHS is cancelled and the new value for LHS is scheduled to be updated (if needed) after new delay period expires (inertial delay)
     - Requires value of RHS expression be stable for length of delay
- Ignored by synthesis

62

Introduction to Verilog

Behavioral Modeling –
Procedural Blocks

63

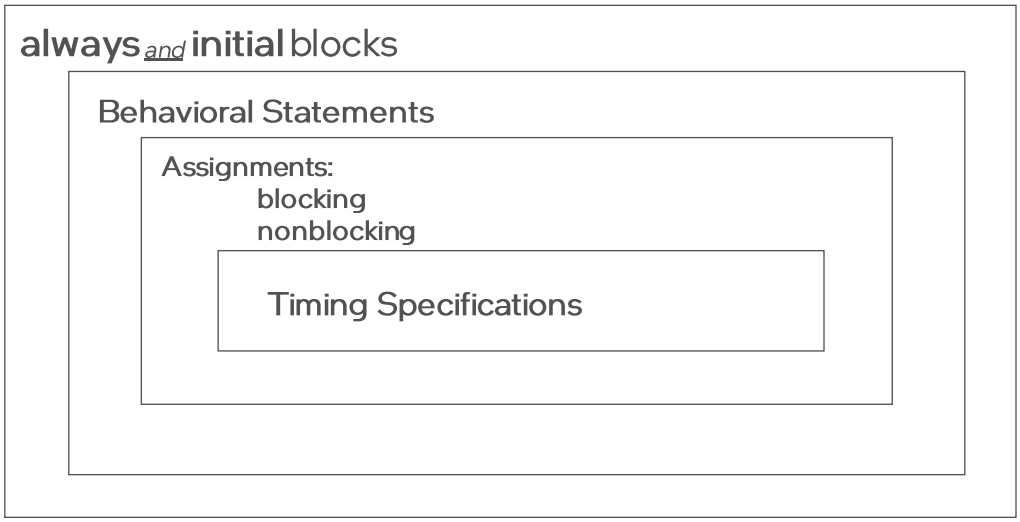
Two Procedural Blocks

- **initial**
  - Used to initialize behavioral statements for simulation
- **always**
  - 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
⇨Statements inside a process execute sequentially
⇨**always** and **initial** blocks cannot be nested

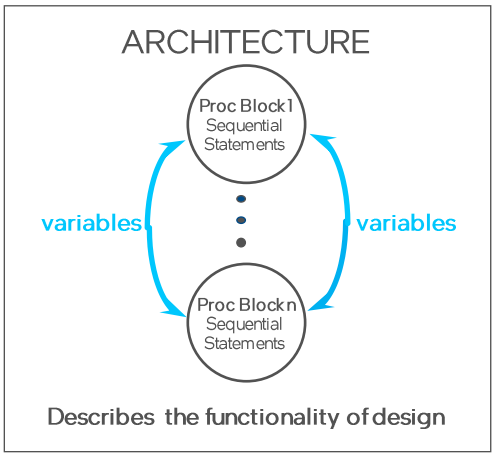
64

Two Procedural Blocks



65

Procedural Block Behavior



- Each procedural block executes in parallel with other procedural blocks
  - Order of **always/initial** blocks does not matter
- Within a procedural block, the statements are executed sequentially
  - Order of statements within an **always/initial** block does matter

66

Procedural Block Characteristics

1) LHS must be a variable data type (e.g. reg, integer, real, time)
2) LHS can be a bit-select or part-select
3) LHS can be a concatenation of any of the above
4) RHS can be expression containing net data type, variable data type or function call (or combination of)

67

initial Block

- Consists of behavioral statements
- Each **initial** block executes concurrently starting at time 0, executes only once and then does not execute again
- Must use keywords **begin** and **end** to group behavioral statements when **initial** block contains more than one behavioral statement
- Example uses
  - Initialization
  - Monitoring
  - Any functionality that needs to be turned on just once

⇒Note that though the **initial** block executes only once, the duration of **initial** block may be infinite (i.e. functionality inside may continue running for the duration of the model execution)

68

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(s) Executed
0	a = 1'b0; b = 1'b1
5	c = 1'b0;
15	d = 1'b0;
20	$finish

69

always Block

- Consists of behavioral statements
- Each **always** block executes concurrently starting at time 0 and executes continuously in a looping fashion
- Must use keywords **begin** and **end** to group behavioral statements when **always** block contains more than one behavioral statement
- Example uses
  - Modeling a digital circuit
  - Any process or functionality that needs to be executed continuously

70

always Block Example

```
module clk_gen
   #(parameter period = 50)
   (
   output reg clk
);

   initial clk = 1'b0;

   always
      #(period/2) clk = ~clk;

   initial #100  $finish;

endmodule
```

Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	$finish;

71

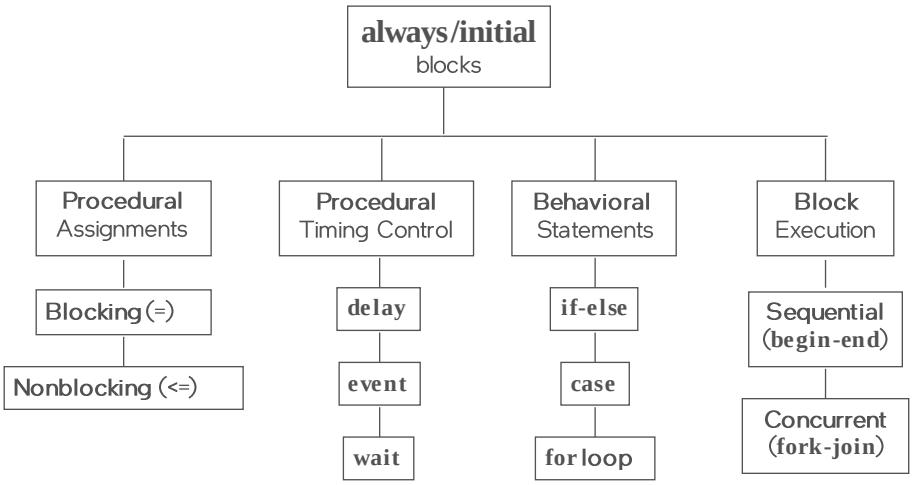
Naming Procedural Blocks

- Procedural blocks may be named by adding : <name> after begin
- Advantages
  - Allows the procedural block to be referenced in other places within the code by name
  - Allows declaration of objects local to the procedural block
  - Allows monitoring of procedural block by name in simulation tools

```
initial
begin : clock_init
   clk = 1'b0;
end

always
begin : clock_proc
   #(period/2) clk = ~clk;
end
```

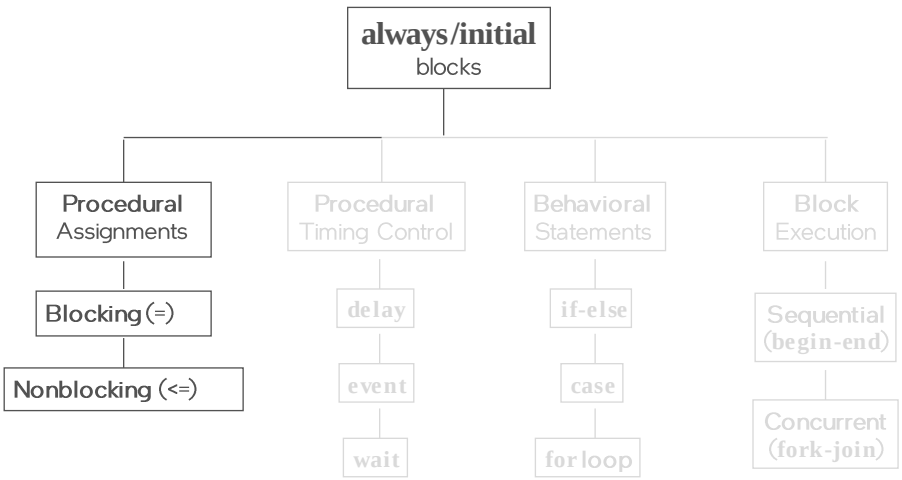

72

always/initial Blocks



73

always/initial Blocks (Procedural Assignments)



74

Procedural Assignment Statements

- Made inside the procedural blocks (**initial**/**always**)
- Update values of variable data types (i.e. **reg**, **integer**, **real**, **time**)
- Place values on a variable that will remain unchanged until another procedural assignment updates the variable with a different value

75

Procedural Assignment Types

In Verilog, there are two types of procedural assignment statements

- Blocking
- Nonblocking

76

Procedural Assignment Types (cont.)

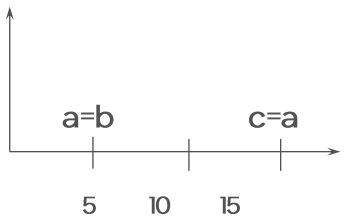
- Blocking (=): Updates LHS assignments blocking execution of other assignments in the procedural block until finished
  - RHS (inputs) sampled when statement executed
  - LHS (outputs) updated immediately or after defined delay
  - Statements following must wait until blocking statement completely finished and LHS assignments are made (including delay) to begin execution
- Nonblocking (<=): Schedules LHS assignments without blocking execution of the statements that follow in a sequential block
  - RHS (inputs) sampled when statement executed
  - LHS (outputs) scheduled to be updated at end of the time step or after delay expires
  - Statements following do not wait until nonblocking LHS assignments are made to begin execution

77

Blocking vs. Nonblocking Assignments

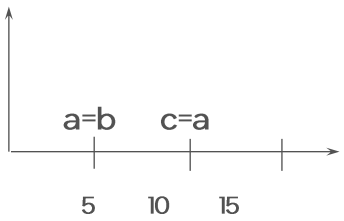
Blocking (=)

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



Nonblocking (<=)

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



78

Evaluating Blocking & Nonblocking

```
initial begin
   a = 1'b0;  //Assgnmnt0
   b = 1'b1;  //Assgnmnt1
   a <= #5 b; //Assgnmnt2
   b <= #5 a; //Assgnmnt3
end
```

This may seem confusing but it is perfectly valid Verilog and is here to illustrate the behavior of blocking and nonblocking statements

1. Assignment 0 executes and assigns **a** to 0 blocking remaining assignments
2. Assignment 1 executes and assigns **b** to 1 blocking remaining assignments
3. Assignment 2 executes and schedules **a** to take on the current value of **b** (1) after 5 time steps
4. Assignment 3 executes and schedules **b** to take the current value of **a** (0) after 5 time steps
5. Process ends and 5 time steps later **a** is updated to 1 and **b** is updated to 0

79

Notes on Blocking & Nonblocking Assignments

1) Avoid making multiple simultaneous (same time step) assignments to same variable in <u>different</u> processes
   - May causes indeterminate value (race condition)

2) For simultaneous (same time step) assignments to same variable in <u>same</u> process
   - Last blocking variable assignment overwrites previous assignments
   - Last scheduled non-blocking variable assignment overwrites previous assignments

1)
```
initial #4 a = 1'b0;
initial #4 a = 1'b1;
```

2)
```
initial begin
   a = 1'b0;
   a = 1'b1; // Last
end
initial begin
   a <= #4 1'b1;
   a <= #4 1'b0; // Last
end
```


80

Notes on Blocking & Nonblocking Assignments

1) During a time step, blocking assignments executed before nonblocking assignments
   - This is due to Verilog event queue (order in which events must be handled)
   - Exception: When non-blocking assignment triggers blocking assignment in another process
2) Not recommended to use both blocking & nonblocking assignments in same procedural block
   - Though legal Verilog, considered poor coding style
   - Requires very good understanding of Verilog event queue
   - Easy to produce incorrect or unexpected behavior

1)
```
initial #4 a = 1'b0; //
1st
initial #4 a <= 1'b1; //
2nd
```

2)
```
always begin
   a <= in * 2;
   b = a + 1'b1; // Old
'a'
   …
end
```


81

Notes Explanations

1) Procedural blocks assumed to be in parallel with other procedural blocks, thus no guarantee which assignment will be executed first; results in race condition
2) Since variable assignments are sequential, in both cases last assignment will overwrite the first whether blocking or nonblocking
3) Blocking assignment executed first, then nonblocking, so **a** will be equal to 1 at end of simulation cycle (this should avoided too!)
4) Very poor coding; **b** is actually using the previous value of **a** each time

1)
```
initial #4 a = 1'b0;
initial #4 a = 1'b1;
```

2)
```
initial begin
   a = 1'b0;
   a = 1'b1; // Last
end
initial begin
   a <= #4 1'b1;
   a <= #4 1'b0; // Last
end
```

3)
```
initial #4 a = 1'b0; //
1st
initial #4 a <= 1'b1; //
2nd
```

4)
```
always begin
   a <= in * 2;
   b = a + 1'b1; // Old
'a'
end
```


82

Last Note on Nonblocking

- Unlike with continuous assignments, non-blocking assignments allow scheduling of any number of events to the same variable at different times without cancelling or overwriting prior scheduled events

```
initial begin
  a <= 1'b0;
  a <= #2 1'b1;
  a <= #4 1'b0;
  a <= #6 1'b1;
end
```

*Each nonblocking statement schedules **a** to a value at some time in the future without disturbing the scheduling of the other statements*

More Information on Blocking/Non-Blocking

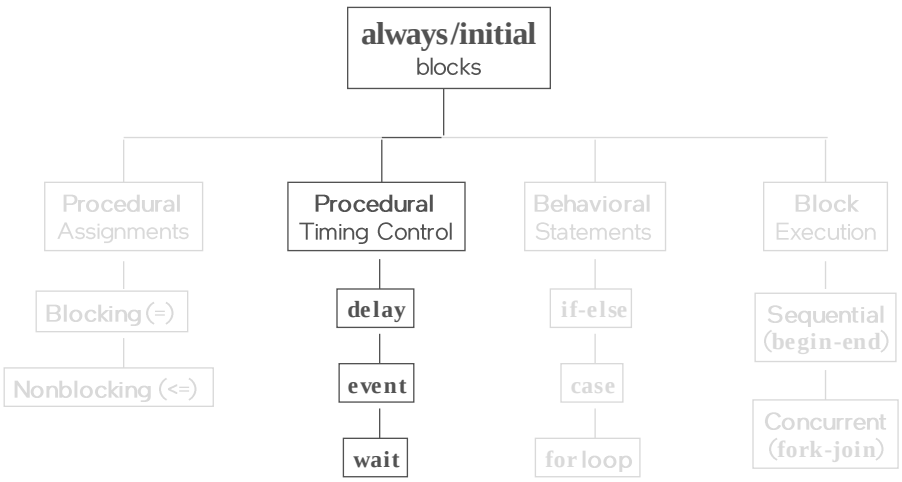
- Verilog Nonblocking Assignments With Delays, Myths & Mysteries by Cliff Cummings
  - http://www.sunburst-design.com/papers/CummingsSNUG2002Boston_NBAwithDelays.pdf
- Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill! by Cliff Cummings
  - http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf

Process Execution Time

- All procedural blocks (processes) in all modules in a design execute together
  - Process execution time starts at time 0
  - Process execution time advances after all processes at current time step have completed their current execution cycle
  - Current execution cycle ends when procedural block reaches one of the following
    - End of initial block
    - Blocking assignment with delay
    - Event control is reached
    - Wait statement
- Process execution time is used to determine the behavior of a model
  - Simulation tool: Process execution time is the same as simulation time
  - Synthesis tool: Must generate functionally equivalent physical logic/hardware

85

always/initial Blocks (Procedural Timing Control)



86

Procedural Timing Control

- Delay control
- Event control
- Wait statement
- Named event
  - Not discusssed

87

Delay Controls for Procedural Assignment

- Use `#<value>` notation to delay executing procedural assignment

- Regular (Inter-Assignment) Delay Control
- Intra-assignment Delay Control
- Zero Delay Control

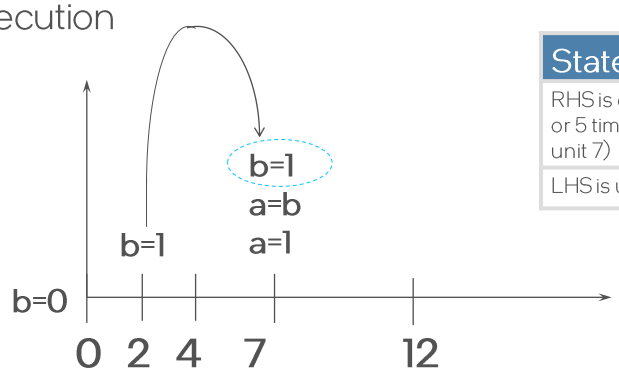
- Ignored for synthesis

88

Regular (Inter-Assignment) Delay Control

```
#5 a = b;
```

- Delays both read (RHS) and write (LHS) portions of statement execution



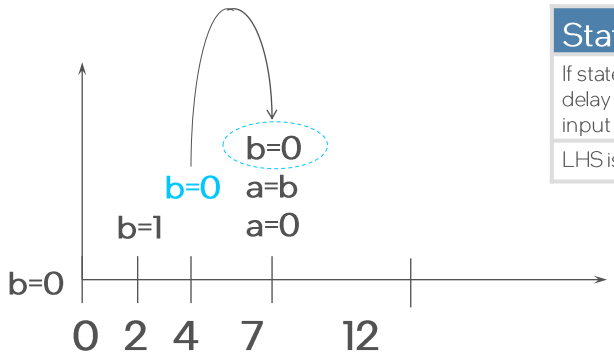
Statement Execution (1)
RHS is evaluated (i.e. **b** is read) after delay expires, or 5 time units after statement is executed (at time unit 7)
LHS is updated immediately

89

Regular (Inter-Assignment) Delay Control (cont.)

```
#5 a = b;
```

- Delays both read (RHS) and write (LHS) portions of statement execution



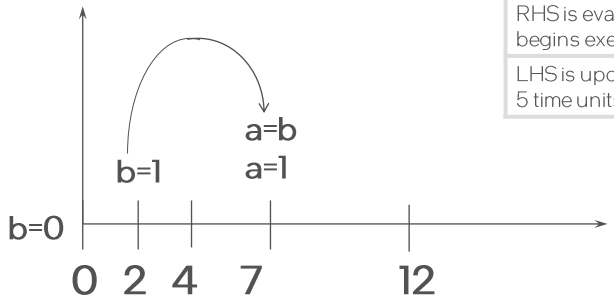
Statement Execution (2)
If statement input (RHS, or **b**) changes before delay expires, RHS picks up the new value of the input when the read actually occurs
LHS is updated immediately with new value

90

Intra-Assignment Delay Control

```
a = #5 b;
```

- Delays only write (LHS) portions of statement execution



Statement Execution (1)

RHS is evaluated (i.e. **b** is read) when statement begins execution

LHS is updated immediately after delay expires, or 5 time units later

91

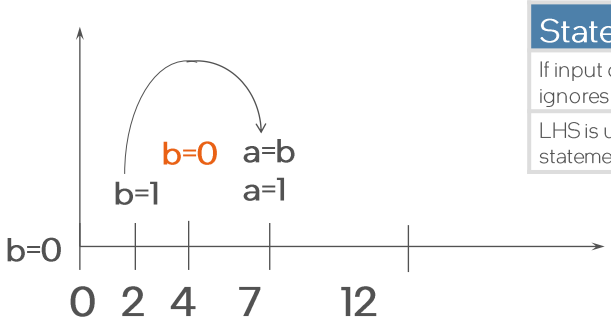
Intra-Assignment Delay Control (cont.)

```
a = #5 b;
```

<=>

```
temp = b;
#5 a = temp;
```

- Delays only write (LHS) portions of statement execution



Statement Execution (2)

If input changes before delay expires, RHS ignores change

LHS is updated with value of RHS when statement began execution

92

Example Uses of Procedural Delay

- Inter-Assignment Delay
  - Use with blocking assignment operator in testbenches to sequence test stimuli
- Intra-Assignment Delay
  - Use with nonblocking assignment operator to model transport delay (e.g. delay lines or transmission lines)
- For more details see Correct Methods for Adding Delays to Verilog Behavioral Models by Cliff Cummings
  - http://www.sunburst-design.com/papers/CummingsHDLCON1999_BehavioralDelays_Rev1_1.pdf

93

Zero Delay Control

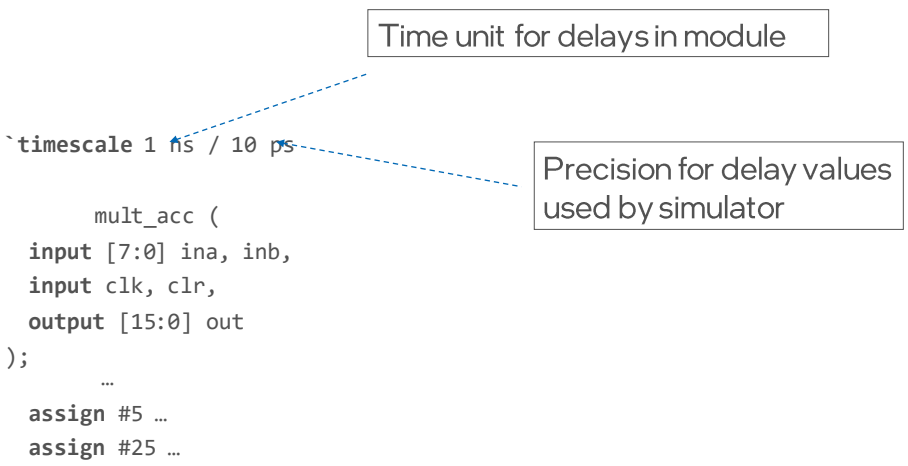
```
initial begin
  a = 0;
  b = 0;
end

initial begin
  #0 a = 1;
  #0 b = 1;
end
```

Statement Execution
All four statements will be executed at simulation time 0
Since #0 used for statements a = 1 and b = 1 have, they will be executed last.

- Provides a way of controlling the order of execution at 0 time
- Still not recommended to assign different values to a variable simultaneously or in different processes

94

`timescale Compiler Directive Preview




```
`timescale 1 ns / 10 ps

      mult_acc (
  input [7:0] ina, inb,
  input clk, clr,
  output [15:0] out
);
       …
  assign #5 …
  assign #25 …
```

Note: Other compiler directives discussed later.

95

Event Control

- Provides edge-sensitive control
- Symbolized by the @ symbol

`@(expression)`

- Pauses execution of procedural statements until event occurs (i.e. expression changes value)
- To test for multiple events (logical OR of events list)
  - Comma (,)
    - Verilog '2001 and later
  - or

Examples*

```
initial begin
    // Rising edge control (inter-assignment)
    @ (posedge clk) r1 = r2;

    // Falling edge control (intra-assignment)
    r3 = @(negedge clk) r4;

    // Either edge control
    @ (a) r5 = r6;

    // Either edge control using more
    //    complex expression
    @ (a ^ b & c) r7 = r8;

    // Logical OR of two events using comma
    @ (posedge clk, enable) r9 = r10;

    // Logical OR of two events using
    //        "or" keyword
    r11 = @ (posedge clk or enable) r12;
end
```


96

Event Control & Sensitivity Lists

- Use event control at the beginning of **always** block to control when block begins execution
  - Each execution of always block requires event to be satisfied
  - Forces always block to be "sensitive" to the items in event control
- Also referred to as a Sensitivity List
- Supported by synthesis tools

Format

```
always @(sensitivity_list) begin
   -- Statement_1
   -- ……….
   -- Statement_N
end
```

Example

```
// Process executes whenever
//    a, b, c or d changes in value
always @(a, b, c, d) begin
   #15 y = (a ^ b) & (c ~| d);
end
```


97

wait Statement

- Provides level-sensitive event control
- Uses `wait` keyword

```
wait (expression)
```

- Pauses execution of the procedural block until wait statement is satisfied
  - If level test not satisfied, procedural block pauses until it is
  - If level test already satisfied, procedural block continues execution without pausing
- Not supported by synthesis

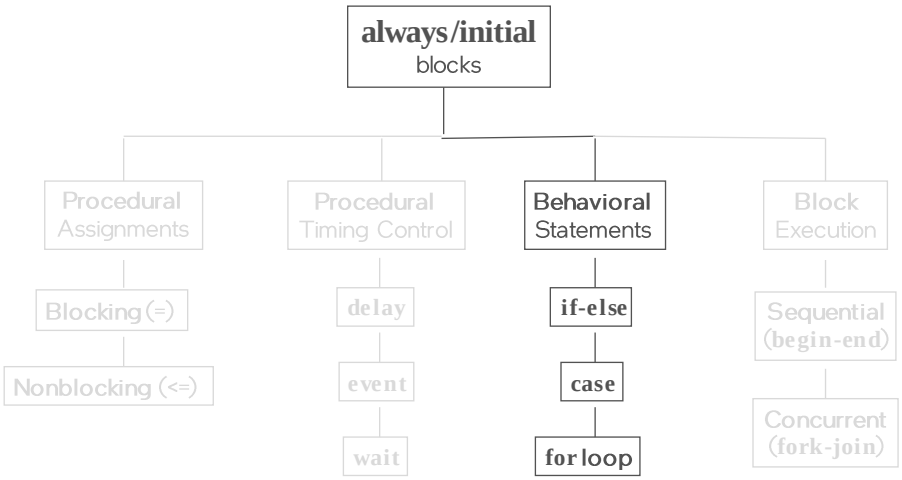
Example

```
initial begin
   …
   wait (gate) r1 = r2;
//Assignment0
   wait (!gate) r3 = r4;
//Assignment1
   …
end
```

- *Assignment0 must pause until gate is true (1) before r1 takes on value of data. Statement does not pause if gate already true*
- *Assignment1 must pause until gate is false (0) before r3 takes on value of data. Statement does not pause if gate already false*

98

always/initial Blocks (Behavioral Statements)



99

Behavioral Statements

- Describe behavior and express order
- Must be used inside procedural block

- Behavioral Statements
  - **if-else** statement
  - **case** statement
  - Loop statements

100

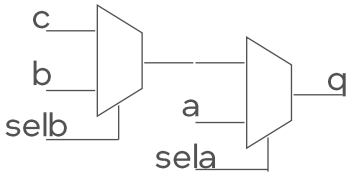
if-else Statements

- Format:

```
if <condition1>
   {sequence of statement(s)}
else if <condition2>
   {sequence of statement(s)}
      …
else
      {sequence of statement(s)}
```

- Example:

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



101

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 final "**else**" clause are evaluated

102

Exercise 2

Please go to Exercise 2
in the Exercise Manual

103

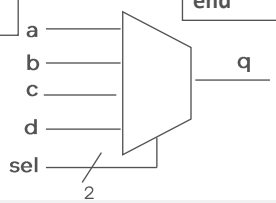
Case Statement

- Format:

```
case {expression}
   <condition1> :
      {sequence of statements}
   <condition2> :
      {sequence of statements}
   …
   default : -- (optional)
      {sequence of statements}
endcase
```

- Example:

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



104

case Statement

- Conditions are evaluated in order
- First matching value is chosen
- Treats both X and Z as actual logic values
- **default** clause represents all other possible conditions that are not specifically stated

- Verilog does not require (though it is recommended) that
  - All possible conditions be considered
  - All conditions be unique

105

Two Other Forms of case Statements

- **casez**
  - Treats both Z and ? in the case conditions as don't cares

```
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'b1z0x**, then **high_lvl = 3**

- **casex**
  - Treats X, Z, and ? in the case conditions as don't cares, instead of logic values

```
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'b1z0x**, then **high_lvl = 3**

106

Exercise 3

Please go to Exercise 3
in the Exercise Manual

107

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

108

forever and repeat Loops

- **forever** loop - executes continually

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

Clock with period of 50 time units

Not synthesizable!

- **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

109

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
```

Counts from 0 to 100
Exits loop at count 101

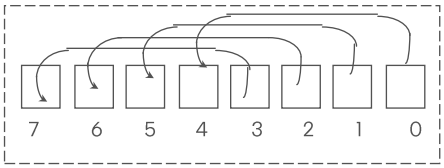
Not synthesizable!

110

for Loop

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

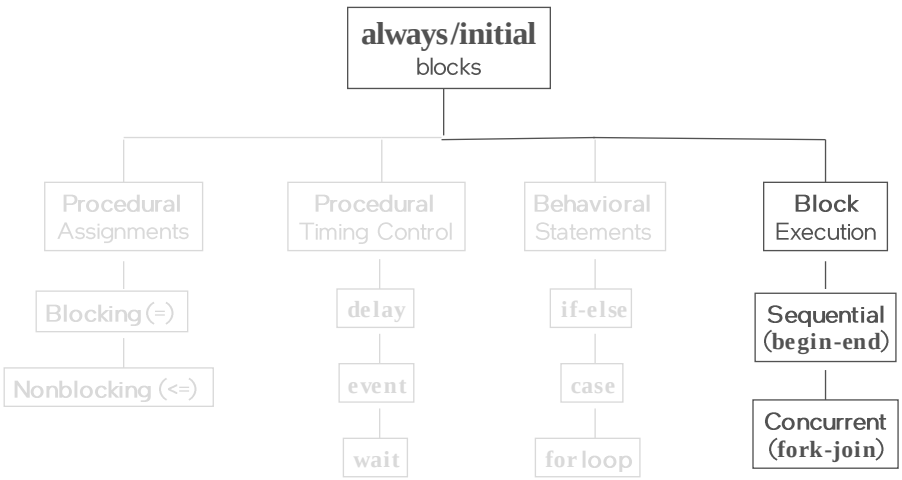
4-bit Left Shifter




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

always @(inp, 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
end
```


111

always/initial Blocks (Block Execution)



112

Block Execution

- Groups statements together within a procedural block so they are perceived as a single statement
- Two types of block execution
  - Sequential Blocks
  - Parallel Blocks

113

Block Execution Types

- Sequential Blocks
  - Statements between **begin** and **end** execute sequentially
  - If there are multiple behavioral statements inside an **initial** or **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** or **always** block and you want the statements to execute in parallel, the statements must be grouped using the keywords **fork** and **join**
  - Not supported by synthesis
    - Use nonblocking assignments to achieve this behavior

114

Sequential vs. Parallel Blocks

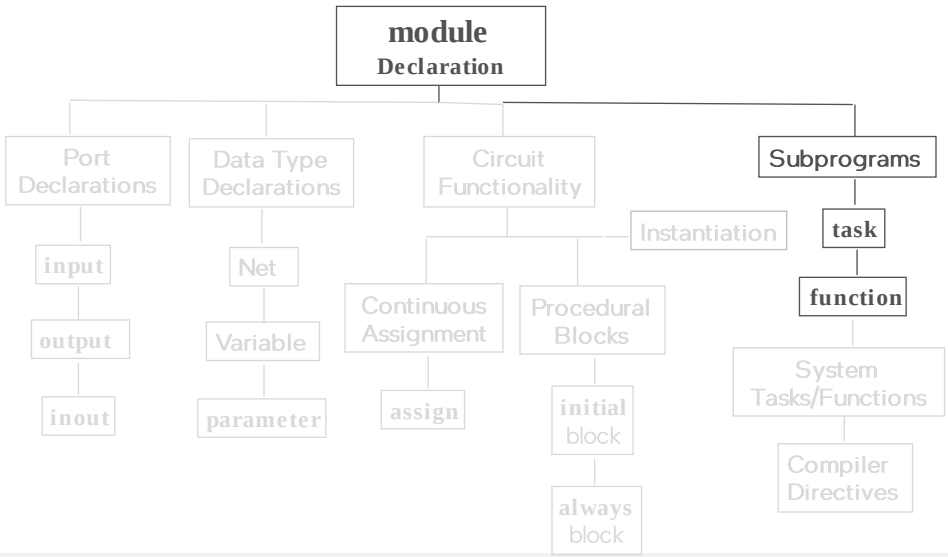
- Sequential vs. Parallel Blocks can be nested

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

Time	Statement(s) Executed
10	a = 1'b1;
15	b = 1'b1;
20	c = 1'b1;
25	e = 1'b1;
30	d = 1'b1;

115

Let's take a look at



116

Introduction to Verilog

Behavioral Modeling –
Tasks & Functions

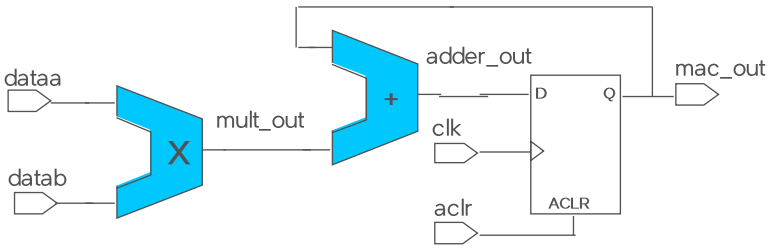
117

Verilog Functions and Tasks

- Function and Tasks are subprograms
- Consist of behavioral statements (like a procedural block)
- Defined within a module
- Uses
  - Replacing repetitive code
  - Enhancing 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);`

118

Create a Function for the Multiplier



119

Function Definition - Multiplier

```
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
```


120

Example Function Invocation

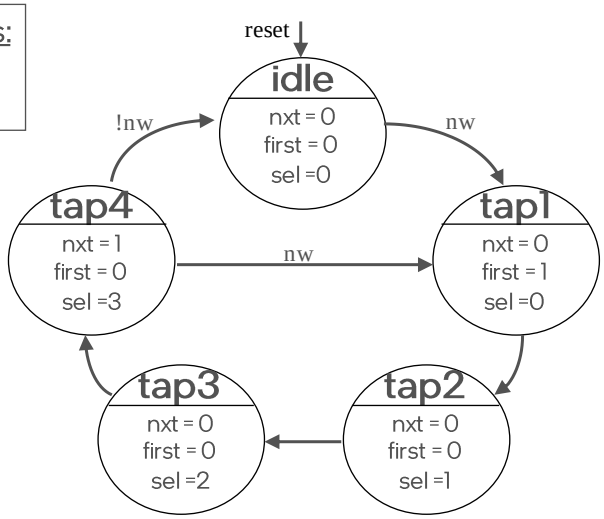
```
`timescale 1 ns/ 10 ps

module mult_acc (
   input [7:0] dataa, datab,
   input clk, aclr,
   output reg [15:0] mac_out
);

   wire [15:0] mult_out, adder_out;

   parameter set = 10;
   parameter hld = 20;
```

```
   assign adder_out = mult_out + mac_out;

   always @ (posedge clk, posedge aclr)
begin
      if (clr)
         mac_out <= 16'h0000;
      else
         mac_out <= adder_out;
   end

   assign mult_out = mult (dataa, datab)

endmodule
```


121

Create Task for State Machine Output

Inputs:
reset
nw



Outputs:
sel
first
nxt

122

Task Definition – State Machine Output

```
task stm_out;
   output reg nxt, first;
   output reg [1:0] sel;
   input [2:0] filter;
   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
```


123

Task Invocation – State Machine

```
module stm_fir (
    input clk, reset, nw,
    output reg nxt, first,
    output 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);
endmodule
```


124

Functions vs. Tasks

Functions

- Always execute in zero time
  - Cannot pause their execution
  - Can not contain any delay, event, or timing control statements
- Must have at least one input argument
  - Inputs may not be affected by function
- Arguments may not be outputs and inouts
- Always return a single value
- May call another function but not a task

Tasks

- 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
- Modify zero or more values
- May call functions or other tasks

125

Review - Behavioral Modeling

Continuous Assignment

```
module full_adder4 (
   output [3:0] fsum,
   output fco,
   input [3:0] a, b,
   input cin
);

assign {fco, fsum} = cin + a + b;

endmodule
```

Procedural Block

```
module fll_add4 (
   output reg [3:0] fsum,
   output reg fco,
   input [3:0] a, b,
   input cin
);

always @(cin or a or b)
   {fco, fsum} = cin + a + b;

endmodule
```

Will produce the same logical model and functionality

126

Introduction to Verilog

Behavioral Modeling –
RTL Processes

127

RTL Processes

- If you remember, RTL is a synthesizable behavioral coding style
- RTL coding style involves two types of procedural blocks (processes)
  - Combinatorial Process
  - Clocked Process

128

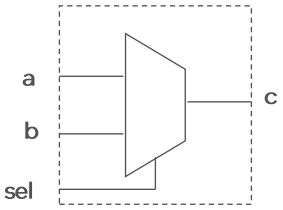
Two Types of RTL Processes

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

```
always @ (a, b, sel)
always @ *
```

Sensitivity list includes all inputs used In the combinatorial logic

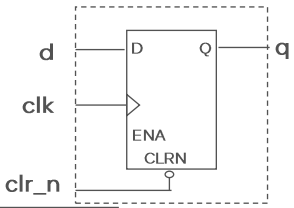
** is a Verilog shortcut to manually having to add all inputs*



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

```
always @(posedge clk, negedge clr_n)
```

Sensitivity list does not include the **d** *input, only the clock and asynchronous control signals*



129

Functional Latch vs. Functional Flipflop

Level-Sensitive Latch

```
module latch (
  input d, gate,
  output reg q
);

  always @(d, gate)
    if (gate)
      q = d ;

endmodule
```

Edge-Triggered Flipflop

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

  always @(posedge clk)
    q <= d ;

endmodule
```


130

Synchronous vs. Asynchronous

Synchronous Preset & Clear

```
module dff_sync (
   input d, clk, sclr, spre,
   output reg q
);

   always @(posedge clk) begin
      if (sclr)
         q <= 1'b0;
      else if (spre)
         q <= 1'b1;
      else
         q <= d;
   end

endmodule
```

Asynchronous Clear

```
module dff_async (
   input d, clk, aclr,
   output reg q
);

   always @(posedge clk,
                posedge aclr) begin
      if (aclr)
         q <= 1'b0;
      else
         q <= d;
   end

endmodule
```


131

Clock Enable

Clock Enable

```
module dff_ena (
   input d, enable, clk;
   output reg q
);

/* If clock enable port does not exist in
   target technology, then a mux in
   front of the d input is generated */

   always @( posedge clk )
      if (enable)
         q <= d;

endmodule
```


132

Functional Counter

```
module cntr (
   input aclr, clk,
   input [7:0] d,
   input [1:0] func,  // Controls functionality
   output reg [7:0] q
);

   always @ (posedge clk, posedge aclr) begin
      if (aclr)
         q <= 8'h00;
      else
         case (func)
            2'b00: q <= d;  // Loads counter
            2'b01: q <= q + 1;  // Counts up
            2'b10: q <= q - 1;  // Counts down
         endcase
   end

endmodule
```


133

Blocking/Nonblocking Rule of Thumb

- Use **blocking operator** (=) for **combinatorial logic**
- Use **nonblocking operator** (<=) for **sequential logic**

- This avoids confusion and unintended hardware implementations during RTL synthesis

134

MAC (Behavioral Modeling)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```

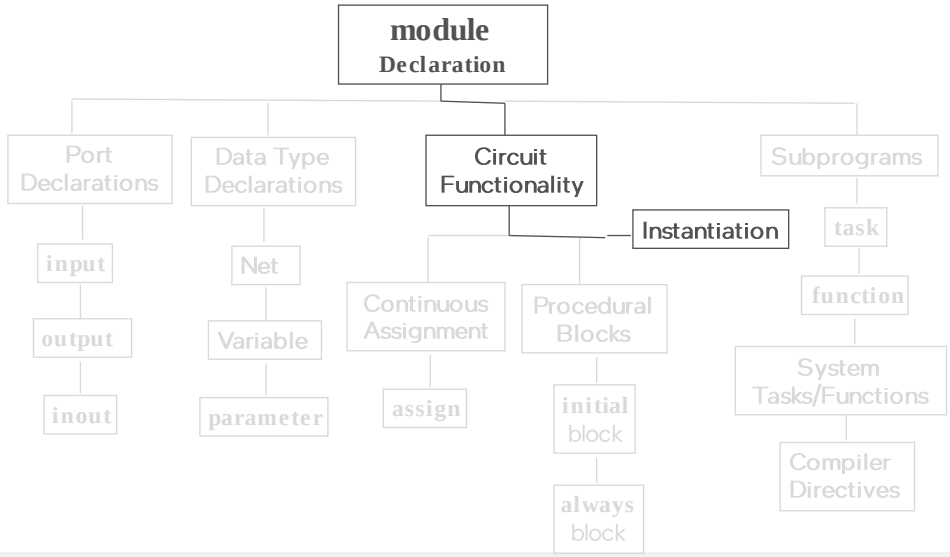

135

Exercise 4

Please go to Exercise 4
in the Exercise Manual

136

Let's take a look at



137

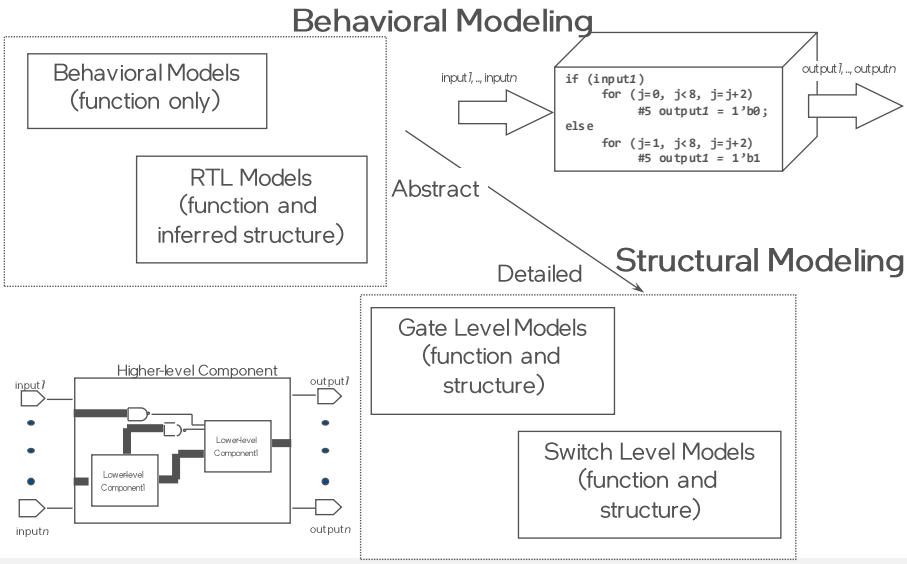
Introduction to Verilog

Structural Modeling

intel.

138

Levels of Abstraction



139

Structural Modeling

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

140

Verilog Structural Modeling

- *Gate-level modeling* - instantiating built-in Verilog gate primitives
  - and, nand, or, nor, xor, xnor
  - buf, bufif0, bufif1, not, notif0, notif1
- User-defined primitives – instantiating primitives created by designer
  - Not discussed; see Appendix for examples
- *Module instantiation* - instantiating user-created lower-level designs (components)
- 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

141

Gate Level 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

142

Instantiation of Gate Primitives

- Instantiation Format:

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

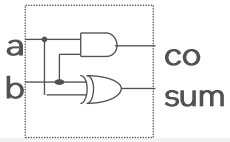
  - *<gate_name>*
    - The name of gate (e.g. AND, NOR, BUFIF0…)
  - *#delay*
    - Delay through gate
    - Optional
  - *<instance_name>*
    - Unique name applied to individual gate instance
    - Optional
  - *(port_list)*
    - List of signals to connect to gate primitive

143

Connecting Gate Primitive Ports

- 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 (
   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
```



144

Module Instantiation

- Instantiation Format:

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

  - *<component_name>*
    - The module name of your lower-level component
  - *#delay*
    - Delay through component
    - Optional
  - *<instance_name>*
    - Unique name applied to individual component instance
  - *(port_list)*
    - List of signals to connect to component

145

Connecting Module Instantiation Ports

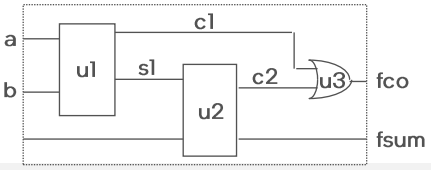
- Two methods to define port connections
  - By ordered list
  - By name

- By ordered list (1st half adder*)
  - Port connections defined by the order of the port list in the lower-level module declaration
    - `module half_adder (co, sum, a, b);`
  - Order of the port connections does matter
    - co -> c1, sum -> s1, a -> a, b -> b

- By name (2nd half_adder*)
  - Port connections defined by name
  - Recommended method
  - Order of the port connections does not matter
    - a -> s1, b -> cin, sum -> fsum, co ->c2

Note: This is the half-adder module from slide 143

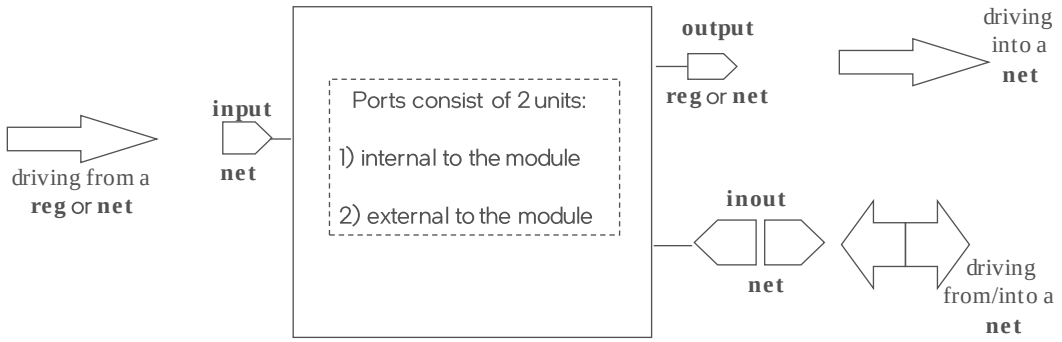
```
module full_adder (
   output fco, fsum,
   input cin, a, b
);
   wire c1, s1, c2;

   half_adder  u1 (c1, s1, a, b);
   half_adder  u2 (.a(s1),  .b(cin),
      .sum(fsum), .co(c2));
   or  u3(fco, c1, c2);

endmodule
```



146

Port Connection Rules

These are the requirements for port connections when modules are instantiated within other modules



147

Overwriting Parameters

- If a lower-level module contains parameters, there are two methods for overwriting its parameter values during instantiation
  - Done at compile time as parameters must resolve to constant values during compilation
- defparams
- Module instance parameter assignment

148

Defparam

- Use **defparam** statement and hierarchical name to overwrite parameters for module instantiations
- May be placed outside Verilog module (i.e. another file)

```
module full_adder (
   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
```


149

Module Instance Parameter Assignment

- Parameter definition occurs during module instantiation
- Introduced in Verilog '2001
- Recommended method as it is easier to read and not prone to error as defparams

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

   wire c1, s1, c2;

   half_adder  #(4, 6)
      u1 (c1, s1, a, b);
   half_adder  #(.and_delay(3), .xor_delay(5))
      u2 (.a(s1),  .b(cin), .sum(fsum), .co(fco));
   or  u3(fco, c1, c2);

endmodule
```

Ordered list method

Name method (recommended)

To just use the (default) value defined in the lower-level module, in the name method, use empty value (e.g. `.and_delay( )`) or leave parameter assignment out entirely

150

MAC (Module Instantiation)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```


151

MAC (Module Instantiation & Local Parameter)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    localparam mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab),
.mult_out(mult_out));

endmodule
```

*Name **mult_size** defined as local parameter so it cannot accidentally be overwritten at compile time*

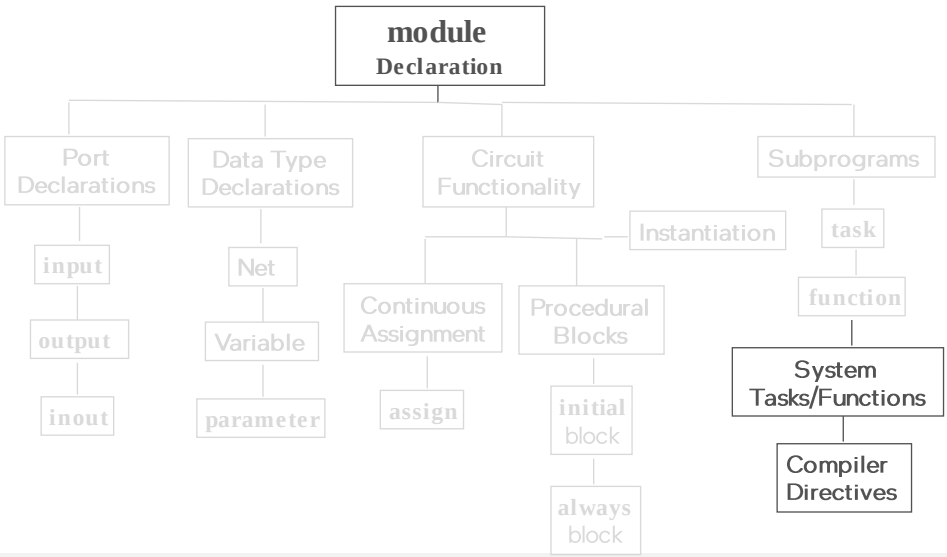
152

Exercise 5

Please go to Exercise 5
in the Exercise Manual

153

Let's take a look at



154

Introduction to Verilog

Compiler Directives & System Tasks

155

Compiler Directives & System Tasks

- Compiler directives
  - Commands issued to direct or control compiler behavior before or during compilation
  - Indicated by the ` (back tick) character
    - Key to the left of 1 key on keyboard, not the apostrophe!
  - Some may be placed within Verilog module, others must be placed outside Verilog module
- System tasks & functions
  - Built-in Verilog tasks and functions
  - Begin with $ character
  - Provide a variety of useful capabilities
- We will cover a few of the compiler directives or system tasks and functions to give you an idea of the capabilities
  - Find many more with a quick web search

156

Compiler Directive Examples

- `timescale
- `include
- `define/ `undef
- `ifdef/ `ifndef/ `elsif/ `else/ `endif

157

`timescale Compiler Directive

- Defines module timing using two values
  - *<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 supported integers
- Must be placed outside of module definition

- Ex: **`timescale** 1 ns / 10 ps

158

`include Compiler Directive

- Includes entire contents of another Verilog source file
- Use to incorporate commonly used library or definition files

```
`include half_adder.v // Same as typing the entire
                      //     half_adder.v file here

module full_adder (
   output fco, fsum,
   input cin, a, b
);
   wire c1, s1, c2;
   …
```


159

`define and `undef Compiler Directives

- Define and undefine macros that perform text substitution
- Use macro by typing `macro_name
- Anything can be substituted
  - Numbers
  - Characters and strings
  - Comments
  - Keywords
  - Operators
- Can be placed outside or inside of module definition
- Can pass arguments
- Have global visibility
  - Once defined, can be used within any Verilog file read afterwards

```
`define HADD_DELAY_VAL1 #(4,6)
`define HADD_DELAY_VAL2 #(3,5)
`define MY_OR(or_out, or_ina, or_inb)  \
          or #5 (or_out, or_ina, or_inb)

module full_adder (
   output fco, fsum,
   input cin, a, b
);
   wire c1, s1, c2;

   half_adder `HADD_DELAY_VAL1
      u1 (c1, s1, a, b);
   half_adder `HADD_DELAY_VAL2
      u2 (.a(s1),  .b(cin), .sum(fsum), .co(c2));
   `MY_OR (fco, c1, c2);

endmodule

`undef MY_OR
```


160

`ifdef / `ifndef / `elsif / `else / `endif Compiler Directives

- Provide support for conditional compilation
  - Ex. Support different variations of a module
  - Ex. Choose different sets of stimulus
- Allow designer to compile Verilog statements based on whether macros have been defined
  - Any valid Verilog statements can be placed within
- Definitions
  - **`ifdef** *<macro_name>* : code compiled if macro defined
  - **`ifndef** *<macro_name>* : code compiled if macro not defined
  - **`elsif** *<macro_name>* : code compiled if macro defined and previous **`ifdef**/**`ifndef** not satisfied
  - **`else** : code compiled if previous **`ifdef**/**`ifndef** not satisfied
  - **`endif** : end code compilation region (one per **`ifdef**/**`ifndef**)
- Can be placed outside or inside of module definition
- Can be nested

161

Conditional Compilation Example

```
// Conditional Compilation

`ifdef  TEST  // Compile counter_test only if macro TEST has been defined
  module counter_test;
  …
  endmodule

`else  // Compile the module counter as default
  module counter;
  …
  endmodule

`endif
```


162

System Tasks and Functions

- Simulation control & time
- Display control
- Math functions
  - Not discussed
- Conversion
  - Not discussed
- File I/O
  - Not discussed

163

Simulation Control & Time

- $stop task
  - Pauses simulation
  - Simulator still running
  - Add argument 1 or 2 to print message about simulator state
- $finish task
  - Stop simulation
  - Shut down simulator
  - Add argument 1 or 2 to print message about simulator state
- $time function
  - Returns current time in simulation

164

Display Control

- All display controls ignored for synthesis
- $display(…) task
  - Writes formatted message to simulator display whenever task is called
  - Example

```
$display("%0d : \n  Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

- $monitor(…) task
  - Writes formatted message to simulator display whenever any monitor inputs change in value (except $time)

```
$monitor("%0d : \n  Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

monitor inputs

165

Introduction to Verilog

Summary

166

Summary

- Verilog is a hardware description language used to model hardware
  - This was an introduction
    - The specification has 560 pages!
- Basic building block is the **module** statement
- All objects must have a defined data type whether net or variable
- Two types of assignments are the continuous and the procedural assignments
- RTL is synthesizable behavioral Verilog code
- When writing RTL code, the two types of procedural blocks are combinational and sequential

167

Introduction to Verilog

Appendix

168

Appendix Topics

- User-defined primitives (UDPs)
- Verilog simulation
- Timing specifications
- Gate delays for primitives

169

User-Defined Primitives (UDP)

- Allows users to define custom Verilog primitives
- Defined using truth tables
- Supports both combinatorial and sequential logic
- UDPs instantiated exactly like built-in primitive
- Characteristics
  - Only 1 output
  - Must have at least 1 input but no more than 10
  - Must appear outside of module definition

170

UDP – Mux (Combinatorial)

```
primitive mux (mux_out, sel, ina, inb);  // combinatorial
   output mux_out;
   input sel, ina, inb;

   table
      // sel  ina  inb  mux_out
           0   0    ? :    0;   // '?' means don't care
           0   1    ? :    1;
           1   ?    0 :    0;
           1   ?    1 :    1;
           x   0    0 :    0;
           x   1    1 :    1;
   endtable
endprimitive
```

Note: Order of ports in table determined by declaration order, not from comment. Inputs are listed first followed by: and then output

171

UDP – Latch (Level-Sensitive)

```
primitive latch (q, gate_n, data); // level sensitive, active
low
   output reg q;
   input gate_n, data;

   initial q = 1'b0; // Output is initialized to 1'b0.
             // Change 1'b0 to 1'b1 for power up Preset
   table
      // gate_n data current_state next_state
            0    1         :?:         1;
            0    0         :?:         0;
            1    ?         :?:         -;      // '-' = no
change
   endtable
endprimitive
```


172

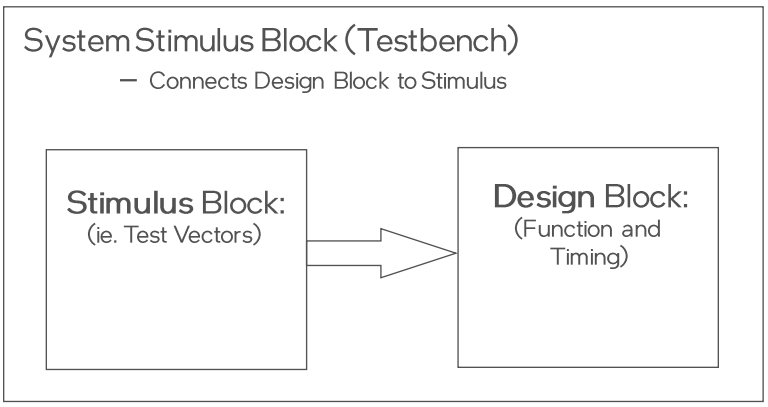
UDP – Register (Edge-Triggered)

```
primitive d_edge_ff (q, clock,data); //edge triggered, active high
   output 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
```


173

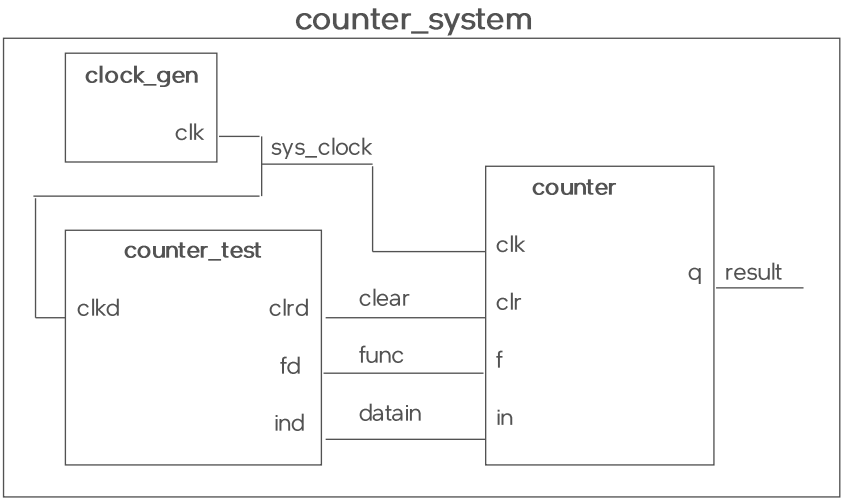
Verilog Simulation Environment

- Contains a Stimulus Block and a Design Block



174

Example - Verilog Simulation Setup



175

System Test Block

```
module              ;   // Top Level Testbench
wire clear, sys_clock;
wire [1:0] func;
wire [7:0] datain, result;

clock_gen      system_clock (sys_clock);
counter_test  op_test (.clrd(clear), .fd(func), .ind(datain), clkd(sys_clock));
counter          DUT     (.q(result), .clk(sys_clock), .clr(clear), .f(func), .d(datain));

initial begin
   $display("\t\t       Time    clear   sys_clock  func  datain  result");
   $monitor($time,,clear,,,,,,,,,sys_clock,,,,,,,,,func,,,,,,datain,,,,,result);
   #1300  $finish;
end

endmodule
```


176

Clock Generation Block

```
module clock_gen(sys_clk);
output sys_clk;
reg sys_clk;
parameter period = 100;

initial
   clk = 1'b0;
always
   #(period/2) clk = ~clk;

endmodule
```


177

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;

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
```


178

Design Block

```
module        (q, clk, clr, f, d);
input clk, clr;
input [1:0] f;
input [7:0] d;
output [7:0] q;
reg [7:0] q;
parameter set = 4, hold = 1;

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
```


179

Timing Specifications

180

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

181

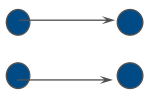
Parallel Connection

- Parallel Connection - specified by the symbol (=>)
- Format:

```
<source> => <destination> = <delay>
```

```
// Single bit a and b
a => b = 5;
```

Source Destination



```
// 2-bit Vector a and b
a => b = 5;

is equivalent to

a[0] => b[0] = 5;
a[1] => b[1] = 5;
```

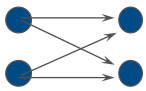

182

Full Connection

- Full Connection - specified by the symbol (*>)
  - Each bit in the source is connected to each bit in the destination
- Format:

```
<source> *> <destination> = <delay>
```

Source Destination



```
// 2-bit Vector a and b
a *> b = 5;

is equivalent to

a[0] *> b[0] = 5;
a[0] *> b[1] = 5;
a[1] *> b[0] = 5;
a[1] *> b[1] = 5;
```


183

Specparam

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

```
specify

     specparam  a_to_b  =  5;

           a  =>  b  =  a_to_b;

end specify
```


184

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
```

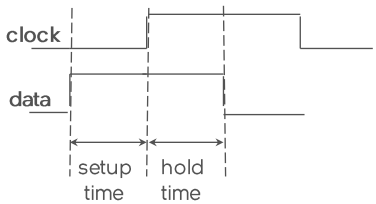

185

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 dat_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 dat_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
```


186

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

<module_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
```


187

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);
```


188

Additional Training and Support Resources

Visit the Intel® FPGA YouTube channel for more than 1000 trainings and quick videos

Visit the Intel FPGA training website for eLearning courses narrated in a slide-by-slide player

Enroll in free instructor-led courses that include interaction and hands-on lab exercises

Still have questions? Visit our forums that are monitored by skilled applications engineers

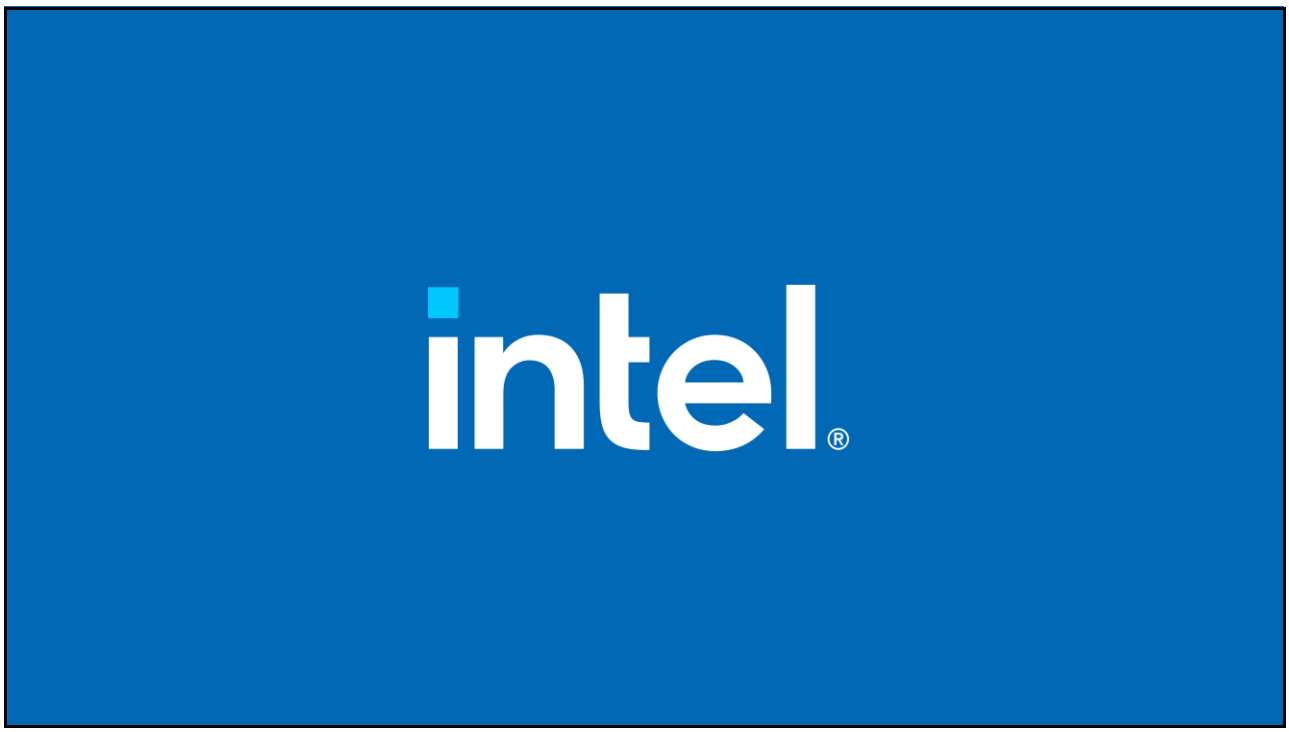
189

Legal Disclaimers/Acknowledgements

- Intel technologies may require enabled hardware, software or service activation
- No product or component can be absolutely secure
- Your costs and results may vary
- Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.
- *Other names and brands may be claimed as the property of others
- OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos

190

191



192