Verilog HDL Basics

Course Outline

- Verilog HDL Overview
- Basic Structure of a Verilog HDL Model
- Components of a Verilog HDL Module
  - Ports
  - Data Types
  - Assigning Values and Numbers
  - Operators
  - Behavioral Modeling
    - Continuous Assignments
    - Procedural Blocks
- Tasks and Functions

What is Verilog?

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

Verilog History

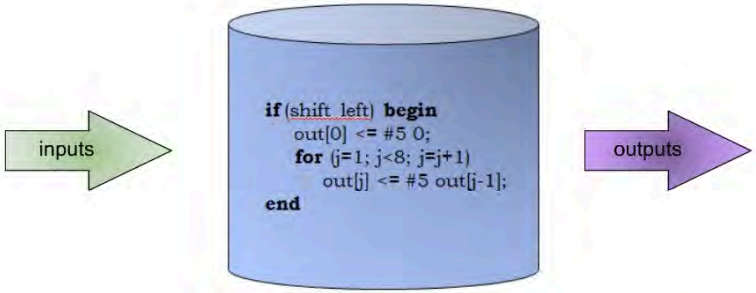
- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence purchased Gateway (Verilog-XL simulator)
- 1990 Cadence released Verilog to the public
- **O**pen **V**erilog **I**nternational (OVI) was formed to control the language specifications
- 1993 OVI released version 2.0
- 1995 IEEE accepted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

Verilog HDL Terminology

- HDL: A text based 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

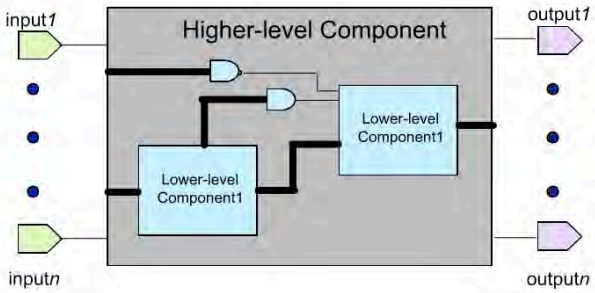
Behavior Modeling

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



Structural Modeling

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



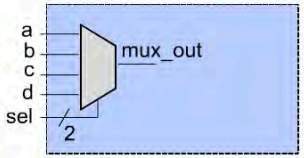
More Terminology

- 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 a RTL model of hardware into an optimized technology specific gate level implementation

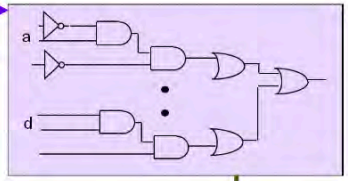
RTL Synthesis

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

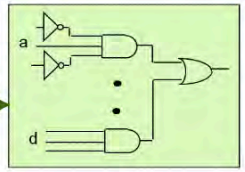
inferred



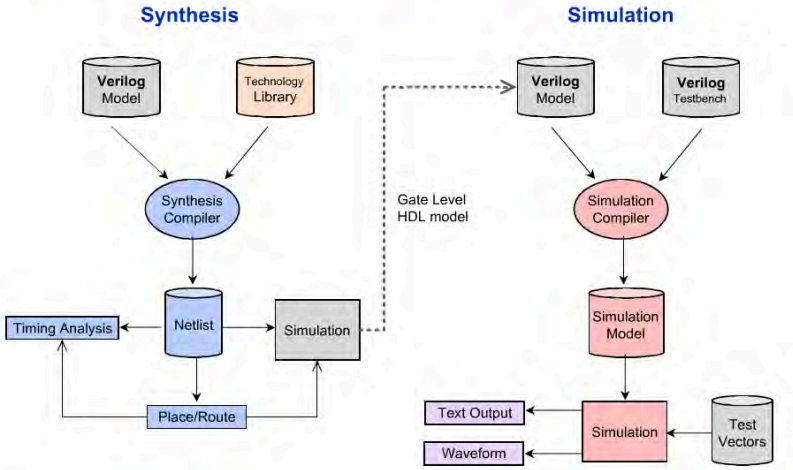
Translation



Optimization



Typical RTL Synthesis & RTL Simulation Flows



Verilog HDL Basics

Module Structure

Verilog - Basic Modeling Structure

module *module_name (port_list);*

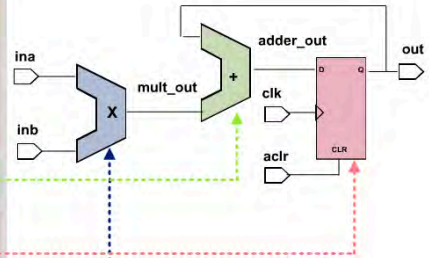
- *port declarations*
- *data type declarations*
- *circuit functionality*
- *timing specifications*

endmodule

- Begins with keyword **module** & ends with keyword **endmodule**
- Case-sensitve
- 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)

Verilog HDL Model: Demonstration Example

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

input [7:0] ina, inb;
input clk, aclr;                    Ports and
output [15:0] out;                  data types

wire [15:0] mult_out, adder_out;
reg [15:0] out;

assign adder_out = mult_out + out;

always @ (posedge clk or posedge aclr)
          if (aclr)    out = 16'h0000;
          else         out = adder_out;

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

endmodule
```



Module and Port Declaration

- Module Declaration:
  - Begins with keyword **module**
  - Provides module name
  - Includes port list, if any
- Port Types:
  - **input** ⇒ input port
  - **output** ⇒ output port
  - **inout** ⇒ bidirectional port
- Port Declarations:

  <port_type> *<port_name>*;

```
module mult_acc (out,
                 ina, inb,
                 clk, aclr);
input [7:0] ina, inb;
input clk, aclr;
output [15:0] out;

…

endmodule
```

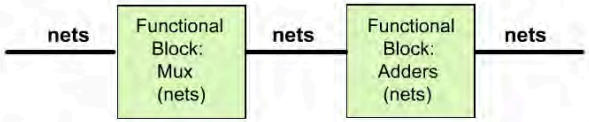

Verilog-2001 & later Module/Port Declaration

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

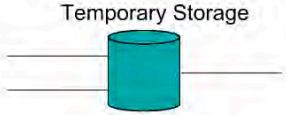
```
module mult_acc
(
   input [7:0] ina, inb,
   input clk, aclr,
   output [15:0] out
);
…
endmodule
```


Data Types

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



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



Net Data Type

Type	Definition
wire	Represents a node or connection
tri	Represents a tri-state node
supply0 supply1	Logic 0 Logic1

- Bus Declarations:
  <data_type> [*MSB* : *LSB*] <*signal name*> ;
  <data_type> [*LSB* : *MSB*] <*signal name*> ;
- Examples:
  - **wire** [7 : 0] out ;
  - **tri** enable;

Variable Data Types

- Variables can be any one of the following:
  - **reg** - unsigned variable of any bit size
    - Use **reg signed** for signed implementation
  - **integer** : signed 32-bit variable
  - **real, time, realtime** : no synthesis support
- Can be assigned only within a procedure, a task or a function
- Bus Declarations:
  reg [*MSB* : *LSB*] <*signal name*>;
  reg [*LSB* : *MSB*] <*signal name*>;
- Examples:
  reg [7 : 0] out ;
  integer count;

Module Instantiation

- Instantiation Format:

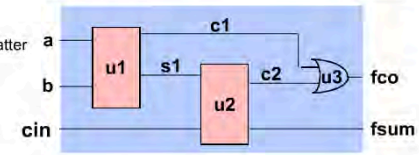
<component_name> #<delay> <instance_name> (port_list);

- ***<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

Connecting Module Instantiation Ports

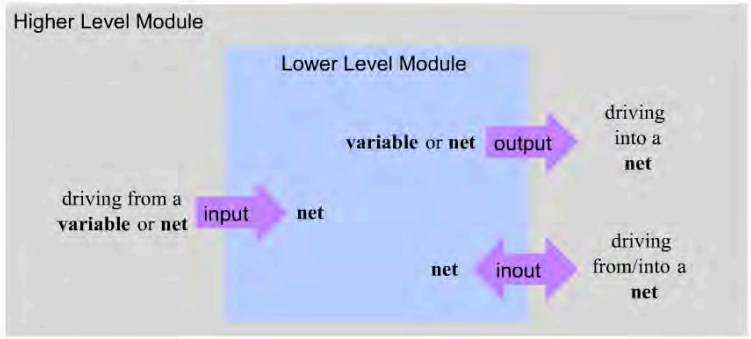
- Two methods to define port connections
  - By ordered list
  - By name
- By ordered list (1st half adder*)
  - Port connections defined by the order of the port list in the lower-level module declaration
    - **module** half_adder (co, sum, a, b);
  - Order of the port connections does matter
    - **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**

```
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
```



Port Connection Rules

- The diagram shows the type requirements for port connection when modules are instantiated



Parameter

- Value assigned to a symbolic name
- Must resolve to a constant at compile time
- Can be overwritten at compile time
- localparam – same as parameter but cannot be overwritten

```
parameter size = 8;
localparam outsize = 16;
reg [size-1:0] dataa, datab;
reg [outsize-1:0] out
```

- Verilog-2001 style, include with module declaration

```
module mult_acc
  #(parameter size = 8)
  (...);
```


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 numer
  - Octal ('o or 'O) 'o21 = 32-bit wide octal number
  - Signed ('s' or 'S') 16'shFA = signed 16-bit hex value

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

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

**** Check synthesis tool for support***

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

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

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

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

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

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

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

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*

Verilog HDL Basics

Making Assignments

Continuous Assignment Statements

- Model the behavior of Combinatorial Logic by using expressions and operators

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

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

is equivalent to

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

```
assign #5 adder_out = mult_out + out
```


Procedural Assignment 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

Initial Block

- Consists of behavioral statements
- An **initial** block starts at time 0, executes only once during simulation, and then does not execute again.
- Keywords **begin** and **end** must be used if block contains more than one statement
- Statements inside execute sequentially
- the **initial** block executes only once, but the duration of the **initial** block maybe infinite.
- Example uses
  - Initialization
  - Monitoring
  - Any functionality that needs to be turned on just once

Always Block

- Consists of behavioral statements
- Blocks executes concurrently starting at time 0 and continuously in a looping fashion
- Keywords **begin** and **end** must be used if block contains more than one statement
- Behavioral statements inside an initial block execute sequentially
- Example uses
  - Modeling a digital circuit
  - Any process or functionality that needs to be executed continuously

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;

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

- Reside inside of procedural blocks
- Update values of **reg**, **integer**, **real**, **time, or realtime** variables **(i.e. Left Hand side type)**

Blocking vs. Nonblocking Assignments

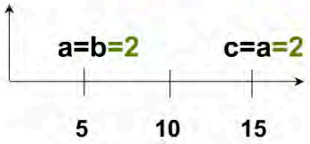
Blocking (=)

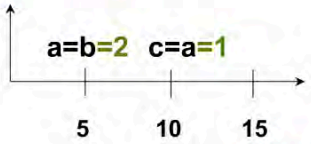
```
initial
  begin
      a = #5 b;
      c = #10 a;
  end
```

Nonblocking (<=)

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

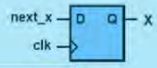
Assuming initially a=1 and b=2



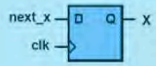


Blocking vs. Nonblocking Assignments

```
always @( posedge clk )
begin
  x = next_x;
end
```

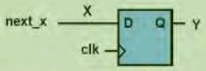



```
always @( posedge clk )
begin
  x <= next_x;
end
```

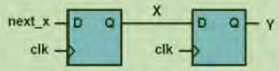


Same Behavior

```
always @( posedge clk )
begin
  x = next_x;
  y = x;
end
```




```
always @( posedge clk )
begin
  x <= next_x;
  y <= x;
end
```



Different Behavior

Blocking/Nonblocking Rule of Thumb

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

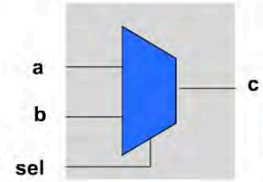
- This avoids confusion and unintended hardware implementations during RTL synthesis

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

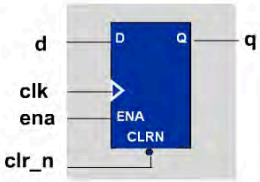


** is a Verilog shortcut to manually having to add all inputs*

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

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

Sensitivity list does not include the **d** *or* **ena** *inputs, only the clock and asynchronous control signals*



Behavioral Statements

- Must be used inside a procedural block (initial or always)
- Behavioral Statements
  - **If-else** statement
    - Conditions are evaluated in order from top to bottom
    - Prioritization
  - **case** statement
    - Conditions are evaluated at once
    - No Prioritization
  - Loop statements
    - used for repetitive operations

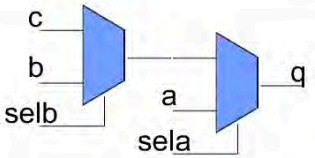
if-else Statements

- Format:

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

- Example:

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



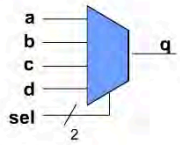
case Statement

- Format:

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

- Example:

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



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**

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

while Loop

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

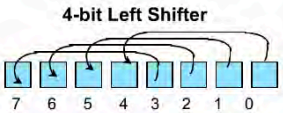
```
initial begin
    count = 0;
    while (count < 101) begin
        $display ("Count = %d", count);
        count = count + 1;
    end
end
```

Counts from 0 to 100
Exits loop at count 101

Not synthesizable!

for Loop

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




```
// 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
```


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
```

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
```


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
```


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);`

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
```

Function Invocation - MAC

```
module mult_acc (
      input [7:0] ina, inb;
      input clk, clr;
      output reg [15:0] mac_out
);

wire [15:0] mult_out, adder_out;
parameter set = 10;
parameter hld = 20;

assign adder_out = mult_out + out;
always @ (posedge clk or posedge clr)
            if (clr)     out <= 16'h0000;
            else         out <= adder_out;

// Function Invocation
assign    mult_out = mult(ina, inb);

endmodule
```


Task Example

```
module module_name;

task add; // task definition
            input a, b;  // 2 input argument ports
            output c;    // 1 output argument port
begin
            c = a + b ;
end
endtask



initial
begin: init1
            reg p;
            add(1, 0, p);
            $display("p= %b", p);
end

endmodule
```

Task definition

Task invocation

Values are passed in the order they appear

Functions vs. Tasks

Functions	Tasks
■ Always execute in zero time – Cannot pause their execution – Cannot 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	■ 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

Demonstration using Modelsim

Demo should open automatically;
click link if it doesn't

Class Summary

- This class covered the following topics:
  - Basic structure of a Verilog HDL model
  - Components of a Verilog HDL module
    - Ports
    - Data types
    - Assigning values and numbers
    - Operators
    - Behavioral modeling
      - Continuous assignment statements
      - Procedural blocks and assignments
    - Overview of tasks and functions

Reference Material

- Advanced Synthesis cookbook
  - http://www.altera.com/literature/manual/stx_cookbook.pdf
- Quartus II handbook, Volume 1, design and synthesis
- Introduction to Verilog HDL, an 8-hour instructor led class from Altera
- Advanced Verilog HDL Design Techniques, an 8-hour instructor led class from Altera

Altera Technical Support

- Reference Quartus II software on-line help
- Consult ALTERA applications (factory applications engineers)
  - Hotline: (800) 800-EPLD (7:00 a.m. - 5:00 p.m. PST)
  - Mysupport: www.altera.com/mysupport
- Field applications engineers: contact your local altera sales office
- Receive literature by mail: (888) 3-ALTERA
- Wiki: www.alterawiki.com
- Forum: www.alteraforum.com
- FTP: ftp.altera.com
- World-wide web: www.altera.com
  - Use solutions to search for answers to technical problems
  - View design examples

Thank You