

Hardware Description Language (HDL)

Verilog

What is Verilog?

- Verilog is a Hardware Description Language (HDL) used to model digital systems by coding its functionality.

Why is Verilog important in hardware design?

1. Hardware Description Language (HDL)

Verilog is a Hardware Description Language (HDL) that allows designers to describe the functionality and structure of digital circuits.

2. Simulation and Verification

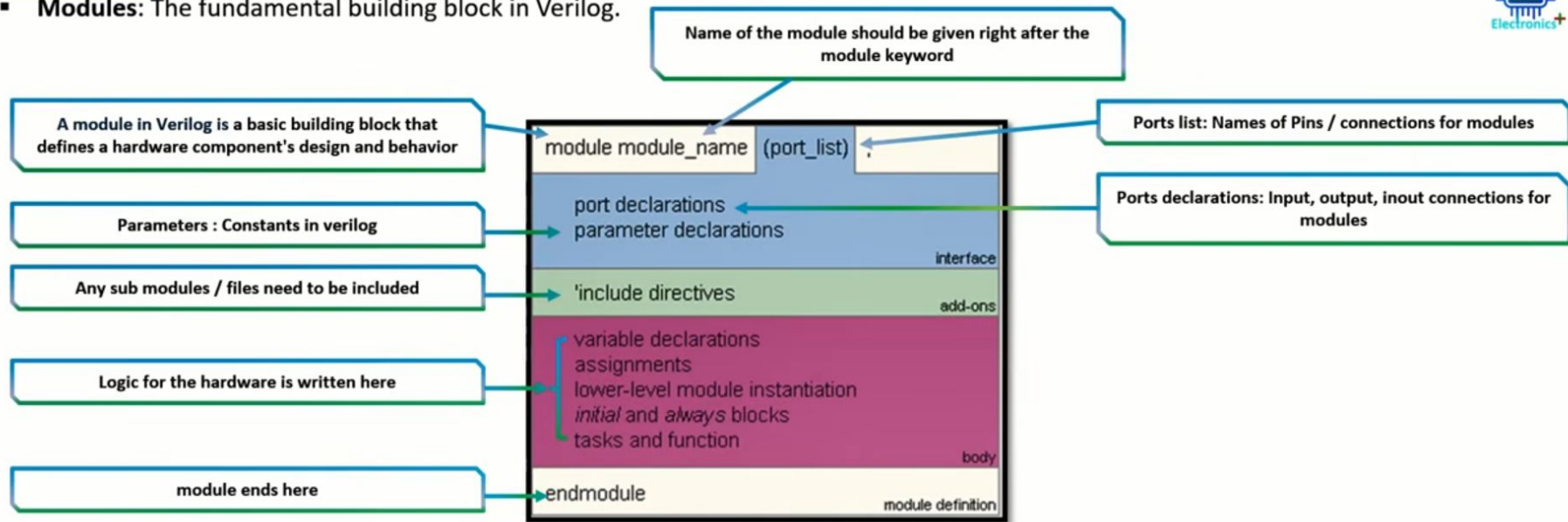
One of the most important aspects of Verilog is that it allows designers to simulate and verify their designs before implementation.

3. Synthesis for Hardware Implementation

Verilog allows for synthesis, which is the process of converting a high-level design description into a gate-level representation that can be implemented on hardware like FPGAs (Field-Programmable Gate Arrays) or ASICs (Application-Specific Integrated Circuits).

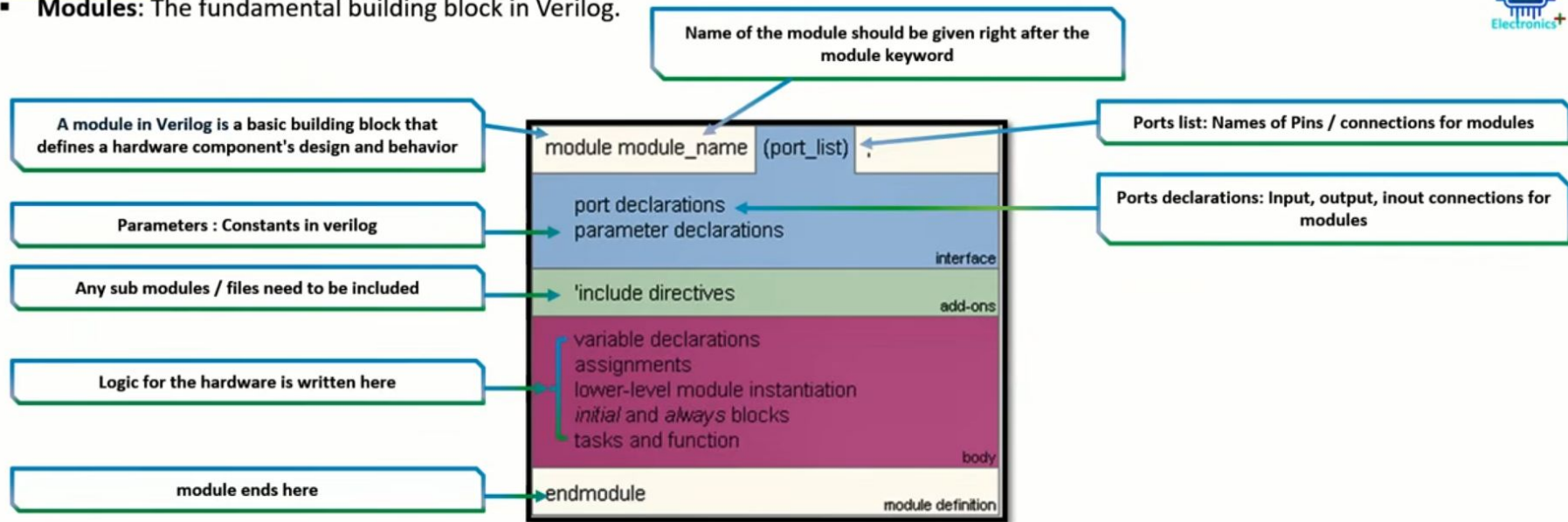
Basic Syntax

- **Modules:** The fundamental building block in Verilog.



Basic Syntax

- **Modules:** The fundamental building block in Verilog.



// This module takes two inputs and generates one output.

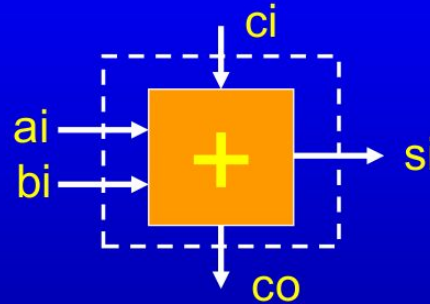
```
module and_gate(input A, input B, output C);
    assign C = A & B;
endmodule
```



Specifying Concurrency: Concurrent Signal Assignment

```
module full_adder (  
  input ai,  
  input bi,  
  input ci,  
  output si,  
  output co  
);  
  assign si = ai ^ bi ^ ci;  
  assign co = (ai & bi) | (bi & ci) | (ai & ci);  
endmodule
```

Concurrent Signal Assignments



When is Signal Assignment Executed?

Assignment executed when any signal on RHS changes

```
assign si = ai ^ bi ^ ci;  
assign co = (ai & bi) | (bi & ci) | (ai & ci);
```

Executed when
ai, bi, or ci changes

Executed when
ai, bi, or ci changes

Order of Execution

- Execution independent of specification order

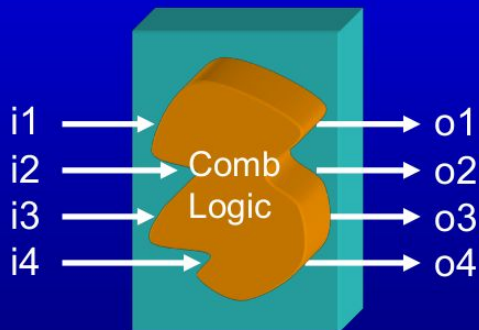
```
module full_adder (  
  // port declarations  
);  
  assign si = ai ^ bi ^ ci;  
  assign co = (ai & bi) | (bi & ci) | (ai & ci);  
endmodule
```

```
module full_adder (  
  // port declarations  
);  
  assign co = (ai & bi) | (bi & ci) | (ai & ci);  
  assign si = ai ^ bi ^ ci;  
endmodule
```

These two are equivalent

Continuous Assignment

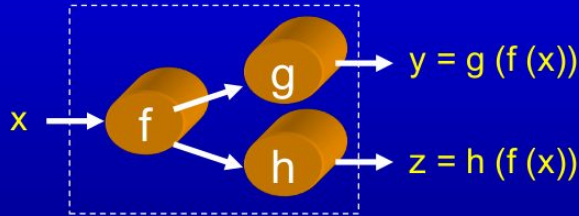
- Continuous assignments with `assign`
- One concurrent assignment for each output
- Models `combinational logic` (more on this later in the course)
 - Output is function only of input



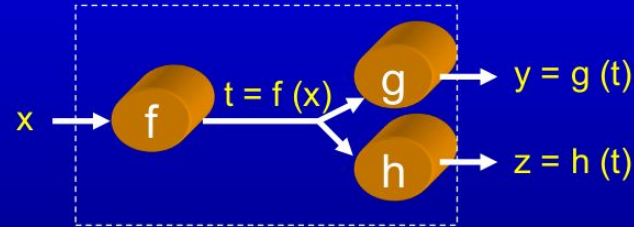
```
module comb_logic(  
    output o1, o2, o3, o4,  
    input i1, i2, i3, i4  
);  
    assign o1 = i1 & i2;  
    assign o2 = (i2 | i3) ^ (i1 & i4);  
    assign o3 = ...;  
    assign o4 = ...;  
endmodule
```

When Logic Complexity Increases

- Temporary SIGNALS needed
- Avoid redundant evaluations



Ports: x, y, z



Intermediate signal t : **Wire**

Wires

- Represent intermediate nets/wires
- Internal - not visible outside module

```
module comb_logic (  
  input i1, i2, i3, i4,  
  output o1, o2);  
  assign o1 = (i1 & i2 & i3) ^ i2;  
  assign o2 = (i1 & i2 & i3) | i4;  
endmodule;
```



```
module comb_logic (  
  Input i1, i2, i3, i4,  
  output o1, o2);  
  wire temp;  
  assign temp = (i1 & i2 & i3);  
  assign o1 = temp ^ i2;  
  assign o2 = temp | i4;  
endmodule;
```

Updation of Wires

- executed when i1, i2, or i3 changes
- executed when temp or i2 changes
- Ports (input/output) are special type of wires: visible outside the module

```
module comb_logic (  
  Input i1, i2, i3, i4,  
  output o1, o2);  
  wire temp;  
  assign temp = (i1 & i2 & i3);  
  assign o1 = temp ^ i2;  
  assign o2 = temp | i4;  
endmodule;
```



Describing Behaviour: Processes

- Signal assignment statements (Continuous assignment)
OK for simple behaviour
- Complex behaviour requires more constructs
 - conditionals (IF, CASE)
 - loops (FOR, WHILE)
- Use **always** block (Procedural assignment)

Verilog always block

- Execution within a block is sequential
- Different **always** blocks are concurrent with respect to each other
- Signal assignment is a simple special case
- Module consists of a set of **always** blocks (and signal assignments) at top level
- Different **always** blocks communicate using signals

```
module example_top (...);
```

```
    assign f = g + 1;
```

```
    always @(*) begin
        if (x)...
        else...;
    end
```

```
    integer i;
    always @(...) begin
        for (i = 0; i < 5; i = i + 1) begin
            assign a[i] = 0;
            ...
        end
    end
```

```
endmodule
```

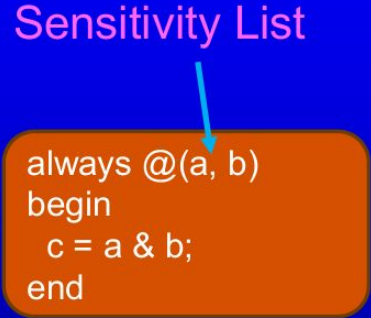
Execution Semantics

- Need to define when a procedural block (always block) is executed
 - suspending/resuming execution
 - more complex than signal assignment (“evaluate when any signal on RHS changes”)
- No notion of “completion” of execution
 - needs to emulate hardware

Sensitivity List

- Process is sensitive to signals on Sensitivity List
- Process execution suspended at end of block
- Reactivated when event (value change) occurs on any signal in sensitivity list

Sensitivity List



```
always @(a, b)
begin
    c = a & b;
end
```

Assignments within *always* block

`a = b;`

- **Blocking Assignments**

- Update is immediate, in the specified order (as in C)
- Blocks the following statements until current statement completes
- Used for **combinational logic** (defined later)

`a <= b;`

- **Non-Blocking Assignments**

- Update is **scheduled** — actual assignment on **suspension of block** (end of **time step**)
- Does not block the following statements
- Used for **sequential logic** (defined later)

Swap using Blocking Assignment

```
module swap();  
  reg [7:0]a, b;  
  
  reg [7:0]temp;  
  
  initial begin  
    a=8'd7;  
    b=8'd4;  
  
    #10;  
    temp = b;// blocking  
    b = a;// blocking  
    a = temp;// blocking  
  end  
  
  initial begin  
    $monitor(" Values of a =%d b =%d", a, b);  
  end  
endmodule
```

a 4

b 7

temp

4

Swap using Non Blocking Assignment

```
module swap();  
  reg [7:0]a, b,c;  
  
  
  initial begin  
    a=7;  
    b=4;  
  
    #10;  
    a <= b; //non blocking  
    b <= a; // non blocking  
  
  end  
  
  initial begin  
    $monitor(" Values of a =%d b =%d", a, b);  
  end  
endmodule
```

Data Types

- **Net:** Wire : Represents a continuous value. **The default value of the net is Z.**
- **Reg:** Represents a value stored over time (used in procedural blocks). **The default value of the reg is X.**
- **Integer:** Stores 32-bit signed integers.
- **Real:** Stores floating-point numbers.

```
module example(input clk, input A, input B, output C);  
    wire w;        // wire declaration  
    reg r;         // reg declaration  
  
    assign w = A & B; // wire continuous assignment  
  
    always @ (posedge clk) begin  
        r = A;        // reg procedural assignment  
    end  
  
endmodule
```

Assignment types in verilog

Types	Continuous assignment	Procedural assignment
Where it's used	Outside of procedural blocks	Inside of procedural blocks
What it does	Drives values onto nets	Updates register and memory data types
When it happens	Automatically active at time zero	Evaluated when the statement is encountered
How it works	Continuous assignments automatically update the variable whenever the right-hand side operand changes. example, assign a = b & c	assignments made within procedural blocks like "always" or "initial" "blocking" or "non-blocking" depending on their execution behavior within a block of code

Operators in Verilog

- **Logical Operators:** &, |, ^, ~
- **Relational Operators:** ==, !=, >, <
- **Arithmetic Operators:** +, -, *, /
- **Shift Operators:** <<, >>

```
module adder(input [3:0] A, input [3:0] B,
output [3:0] Sum, output Cout);
```

```
assign {Cout, Sum} = A + B;
```

```
endmodule
```

Logical Operators	Reduction Operators	Relational Operators
& bitwise AND	& reduce via AND	== equal
bitwise OR	~& reduce via NAND	!= not equal
^ bitwise XOR	reduce via OR	> greater than
^^ bitwise XNOR	~ reduce via NOR	>= greater than or equals
~ bitwise NOT	^ reduce via XOR	< less than
&& boolean AND	^^ reduce via XNOR	<= less than or equals
boolean OR		
! boolean NOT		
Arithmetic Operators	Shift Operators	Other Operators
+ addition	>> shift right	{ } concatenate
- subtraction	<< shift left	{N{ }} replicate N times
	>>> arithmetic shift right	

Descriptions in verilog

Different levels of abstraction and methodologies / styles used to describe hardware

Gate-Level Description :

- model the design using logical gates (like AND, OR, NOT).
- Verilog supports built-in primitive gates modeling.
- This is a more detailed and explicit description, closely mimicking hardware components.

```
module MUX2to1 (
    input a,        // First input
    input b,        // Second input
    input sel,      // Selector input
    output y        // Output
);
    wire not_sel, and1, and2;

    // NOT gate
    not (not_sel, sel);

    // AND gates
    and (and1, a, not_sel);
    and (and2, b, sel);

    // OR gate to combine the results
    or (y, and1, and2);
endmodule
```

Dataflow Description :

- uses assign statements to model the flow of data through the circuit.
- Mainly used to describe combinational circuits
- It's an intermediate level of abstraction between behavioral and gate-level designs.

```
module MUX2to1 (
    input a,        // First input
    input b,        // Second input
    input sel,      // Selector input
    output y        // Output
);
    // Dataflow description using a
    // conditional expression
    assign y = (sel == 1'b1) ? b : a;
endmodule
```

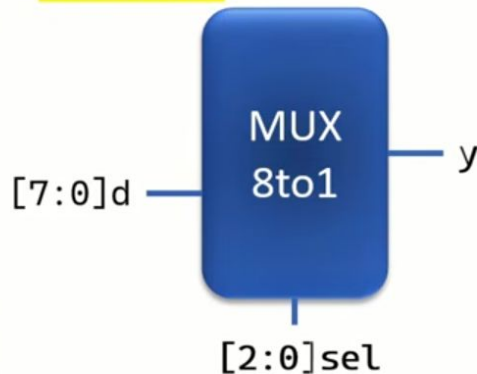
sel	y
0	a
1	b

Behavioral Description:

- Describes the functionality of the hardware using a high-level approach with always blocks and if-else, case and other statements.
- Used to describe complex circuits (primarily sequential)

```
module MUX2to1 (
    input a,        // First input
    input b,        // Second input
    input sel,      // Selector input
    output reg y    // Output (declared
                    // as reg because it will be assigned
                    // in always block)
);
    always @ (sel or a or b) begin
        if (sel)
            y = b; // If sel is 1,
        else
            y = a; // If sel is 0,
    end
endmodule
```

8:1 Mux



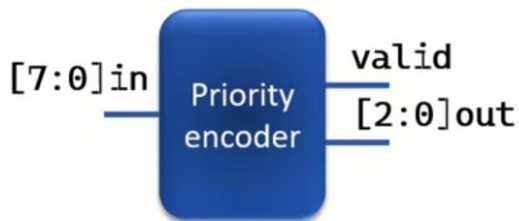
sel	y
000	d[0]
001	d[1]
010	d[2]
011	d[3]
100	d[4]
101	d[5]
110	d[6]
111	d[7]

www.youtube.com – to exit full screen, press Esc



```
module mux_8to1_behavioral (  
    input wire [7:0] d, // 8 data inputs  
    input wire [2:0] sel, // 3-bit select input  
    output reg y // Output  
);  
always @(*) begin  
    case (sel)  
        3'b000: y = d[0];  
        3'b001: y = d[1];  
        3'b010: y = d[2];  
        3'b011: y = d[3];  
        3'b100: y = d[4];  
        3'b101: y = d[5];  
        3'b110: y = d[6];  
        3'b111: y = d[7];  
        default: y = 1'b0;  
    endcase  
end  
endmodule
```

8to3 Priority Encoder : using if else

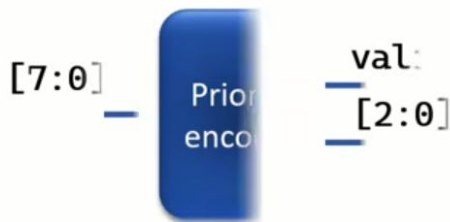


[7:0]In	out	valid
10000000	111	1
01000000	110	1
00100000	101	1
00010000	100	1
00001000	011	1
00000100	010	1
00000010	001	1
00000001	000	1
00000000	XXX	0

```
module Priority_Encoder_8to3 (  
    input [7:0] in, // 8-bit input  
    output reg [2:0] out, // 3-bit output (encoded value)  
    output reg valid // Valid signal: indicates if at least one input is active  
);  
    always @(*) begin  
        // Check the highest priority input first  
        if (in[7]) begin  
            out = 3'b111; valid = 1; // Input 7 is active  
        end  
        else if (in[6]) begin  
            out = 3'b110; valid = 1; // Input 6 is active  
        end  
        else if (in[5]) begin  
            out = 3'b101; valid = 1; // Input 5 is active  
        end  
        else if (in[4]) begin  
            out = 3'b100; valid = 1; // Input 4 is active  
        end  
        else if (in[3]) begin  
            out = 3'b011; valid = 1; // Input 3 is active  
        end  
        else if (in[2]) begin  
            out = 3'b010; valid = 1; // Input 2 is active  
        end  
        else if (in[1]) begin  
            out = 3'b001; valid = 1; // Input 1 is active  
        end  
        else if (in[0]) begin  
            out = 3'b000; valid = 1; // Input 0 is active  
        end  
        else begin  
            out = 3'bXXX; // Default output value if no input is active  
            valid = 0; // Default valid signal as 0  
        end  
    end  
end  
endmodule
```



8to3 Priority Encoder : using casex



In [7:0]	out	valid
10000000	111	1
01000000	110	1
00100000	101	1
00010000	100	1
00001000	011	1
00000100	010	1
00000010	001	1
00000001	000	1
00000000	XXX	0

- Casex statement treats x and z as dont cares.
- x will be matched to 0 or 1 or z or x and z will be matched to 0 or 1 or x or z.

```
module Priority_Encoder_8to3 (  
    input [7:0] in, // 8-bit input  
    output reg [2:0] out, // 3-bit output (encoded value)  
    output reg valid // Valid signal: indicates if at least one input is  
    active  
);  
  
    always @(*) begin  
        casex (in) // 'x' allows wildcards, any undefined ('x') or high-  
        impedance ('z') values are ignored  
            8'b1xxxxxxx: out = 3'b111; // Input 7 is active  
            8'b01xxxxxx: out = 3'b110; // Input 6 is active  
            8'b001xxxxx: out = 3'b101; // Input 5 is active  
            8'b0001xxxx: out = 3'b100; // Input 4 is active  
            8'b00001xxx: out = 3'b011; // Input 3 is active  
            8'b000001xx: out = 3'b010; // Input 2 is active  
            8'b0000001x: out = 3'b001; // Input 1 is active  
            8'b00000001: out = 3'b000; // Input 0 is active  
            default: out = 3'b000; // No input is active  
        endcase  
  
        valid = (in != 8'b00000000); // Set valid to 1 if any input is active  
    end  
endmodule
```

D Flip Flop

clk	reset	d	q
↑	0	x	0
↑	1	1	1
↑	1	0	0



```
module D_FF (input clk, input reset, input d, output reg q);
    always @(posedge clk) begin
        if (reset)
            q <= 0;
        else
            q <= d;
        end
    endmodule
```

SR Flip Flop

clk	rst	S	R	Q
1	1	x	x	0
1	0	0	0	Q (hold)
1	0	0	1	0
1	0	1	0	1
1	0	1	1	Z

```
module sr_flipflop
( input S, // Set input
  input R, // Reset input
  input clk, // Clock input
  output reg Q // Output
);
always @(posedge clk)
begin
    case ({S, R})
        2'b00: Q <= Q; // No change (hold the value)
        2'b01: Q <= 1'b0; // Reset (R = 1, S = 0), set Q to 0
        2'b10: Q <= 1'b1; // Set (S = 1, R = 0), set Q to 1
        2'b11: Q <= 1'bZ; // Invalid state (S = 1, R = 1), output is undefined
    endcase
end
endmodule
```

Synchronous reset and Asynchronous reset

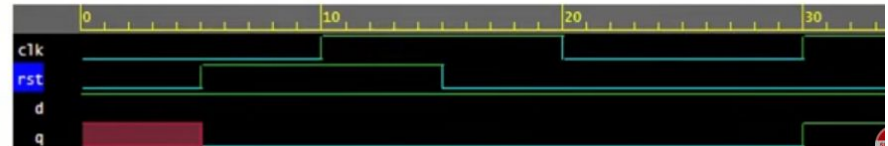
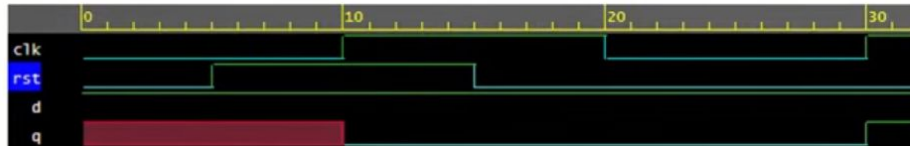


D flipflop using Synchronous reset

```
module d_ff (  
    input clk, rst,  
    input d,  
    output reg q  
);  
  
always@(posedge clk)  
    //Synchronous reset  
    begin  
  
        if(rst) q ≤ 0; //active high reset  
        else    q ≤ d;  
  
    end  
  
endmodule
```

D flipflop using Asynchronous reset

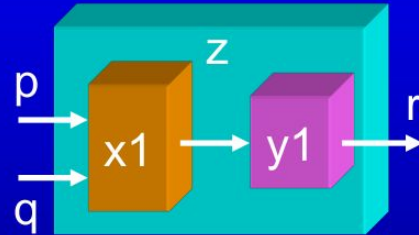
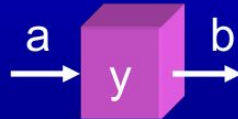
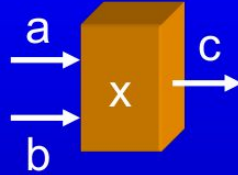
```
module d_ff (  
    input clk, rst,  
    input d,  
    output reg q  
);  
  
always@(posedge clk or posedge rst)  
    //asynchronous reset  
    begin  
  
        if(rst) q ≤ 0; //active high reset  
        else    q ≤ d;  
  
    end  
  
endmodule
```



Hierarchy

```
module x (  
  input a,  
  input b,  
  output c  
);  
  assign c = a & b;  
endmodule
```

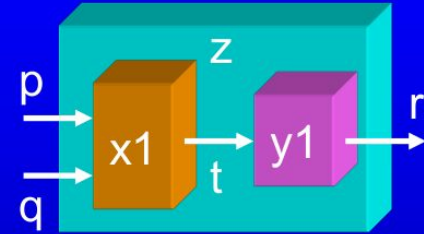
```
module y (  
  input a,  
  output b  
);  
  assign b = ~a;  
endmodule
```



z contains
instances of
x and y

Instantiation and Interconnection - 1

```
module z ( input p, input q, output r);  
  
    wire t; // Internal wire to connect submodules  
    // First submodule instantiation: x  
    x x1 (  
        .a(p),  
        .b(q),  
        .c(t)  
    );  
  
    // Second submodule instantiation: y  
    y y1 (  
        .a(t),  
        .c(r)  
    );  
endmodule
```



Port
association:
a - t
c - r

Temporary signal

Instantiation

Port Mapping: Positional and Named

Positional mapping
OK for short port lists

```
x x1 (p, q, t);
```

Named mapping
preferred for long port lists

```
x x1 (  
    .a(p),  
    .b(q),  
    .c(t)  
);
```

In both cases, specify complete port mapping. Else...

Instantiation of sub modules :



Full Adder using Half Adders

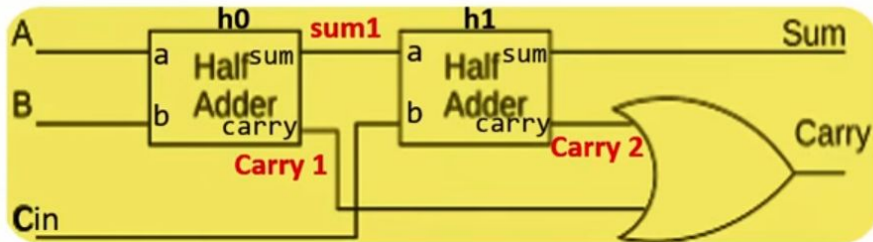
// Design code for Half Adder

```
module half_adder(sum, carry, a, b);  
input a, b;  
output sum, carry;  
assign carry = a&b;  
assign sum = a^b;  
endmodule
```

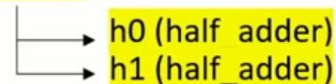


// Design code : Full Adder using Half Adders

```
module full_adder(sum, carry, A, B, Cin);  
output sum, carry;  
input A, B, Cin;  
  
wire sum1, carry1, carry2;  
  
half_adder h0(sum1, carry1, A, B);  
half_adder h1(sum, carry2, sum1, Cin);  
or o1 (carry, carry1, carry2);  
  
endmodule
```



full_adder



Timing and Delays :

www.youtube.com – to exit full screen, press Esc



Delay: This is a simple delay that specifies a time delay after an event. It's the most common way to introduce delays in Verilog.

```
`timescale 1ns / 1ps // Specifies that the time unit is 1ns, with a precision of 1ps
```

```
a = b; // Assign 'b' to 'a'
#5;    // Wait for 5 time units
c = a; // Assign 'a' to 'c' after 5 time units
```

```
assign #10 y = a & b; // 'y' is assigned the value of 'a & b' after 10 time units
```

```
and #(10) (out, in1, in2); // AND gate with 10-time unit delay
```

```
always @(posedge clk) begin // Trigger on rising edge of 'clk'
    // Action
end
```

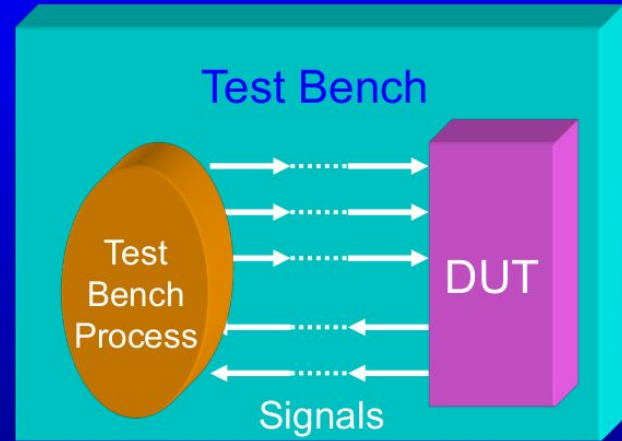
```
always @ (posedge clk) begin
    #5 x = y; // x is assigned the value of y with a 5-time unit delay
end
```

Test Benches

- Purpose - test correctness of Design Under Test (DUT)
 - provide input stimulus
 - observe outputs
 - compare against expected outputs
- Test Bench is also a Verilog model

Test Bench Modelling - 2

- Test bench instantiates the DUT
- Stimulus generation and output monitoring in separate Verilog process
- Signals are connected to DUT's ports



Testbench in Verilog

Half Adder

// Design code in dataflow modeling

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

```
input a, b;
```

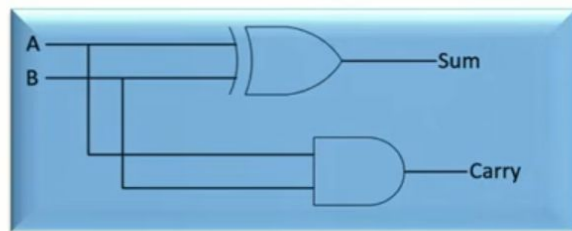
```
output sum, carry;
```

```
// assign used in dataflow modelling
```

```
assign carry = a&b;
```

```
assign sum = a^b;
```

```
endmodule
```



```
// Testbench
```

```
module half_adder_tb;
```

```
reg x,y;
```

```
wire s, c;
```

```
//half_adder dut (x, y, s, c);
```

```
half_adder dut
```

```
(.a(x), .b(y), .sum(s), .carry(c)); //design instantiation
```

```
//stimulus
```

```
initial begin
```

```
    x=0; y=0;
```

```
#10 x=0; y=1;
```

```
#10 x=1; y=0;
```

```
#10 x=1; y=1;
```

```
#20 $finish;
```

```
end
```

```
initial begin
```

```
$monitor( $time, "x=%b y=%b sum=%b  
carry=%b", x,y,s,c);
```

```
end
```

```
endmodule
```



- Tasks and functions provide the ability to reuse the common piece of code from several different places in a design.
- In Verilog, **tasks** and **functions** are essential constructs that help modularize and reuse code.

A **task** in Verilog can perform more complex operations that may involve delays and multiple inputs and outputs.

- ✓ Tasks are primarily used for procedural code that requires side effects, such as simulation tasks or communicating with other modules.

Key Features of Tasks:

- Can contain delays (#, @, or wait).
- Can have multiple inputs and outputs.
- Tasks are invoked using task and endtask keywords.
- They do not return a value but can modify their output arguments.
- Can be executed concurrently.

A **function** in Verilog is used when you need to compute and return a value.

- ✓ Functions are primarily used for combinational logic and cannot include timing controls like delays.

Key Features of Functions:

- Cannot contain delays (#, @, wait are not allowed).
- Can only have input arguments (no output or inout arguments).
- Functions are invoked using function and endfunction keywords.
- Must return a value (using the return mechanism or by assigning to the function name).
- Functions are combinational by nature and compute outputs in zero simulation time.

Example for Tasks

```
// an example design illustrating how to count the zeros in a byte.
module zero_count_task (data, out);
input  [7:0] data;
input  [7:0] data1;
input  [7:0] data3;
output reg [3:0] out; // output declared as register

always @(data) begin
    count_0s_in_byte(data, out); // task declaration or usage
    count_0s_in_byte(data1, out); // task declaration or usage
    count_0s_in_byte(data1, out); // task declaration or usage
end
```

```
// task definition is here.
task count_0s_in_byte(input [7:0] data_in, output reg [3:0] count);
integer i;
begin // task body
    count = 0;
    for (i = 0; i <= 7; i = i + 1)
        if (data_in[i] == 0) count= count + 1;
end

endtask
```

```
endmodule
```

Example for Functions

```
//Define a module that contains the function calc_parity
module parity;
...
reg [31:0] addr; //32bit
reg parity;
//Compute new parity whenever address value changes
always @(addr)
begin
    parity = calc_parity(addr); //First invocation of calc_parity
    $display("Parity calculated = %b", calc_parity(addr) ); //Second invocation of
calc_parity
end
...
...
//define the parity calculation function
function calc_parity;
input [31:0] address; //32
begin
    //set the output value appropriately. Use the implicit
    //internal register calc_parity.
    calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
...
...
endmodule
```

Compiler Directives in verilog



- The compiler directives are similar to C language preprocessor directives that may be used to specify certain information and ask the compiler to process it.
- Compiler directives in Verilog are statements that instruct the compiler on how to compile code.
- In Verilog, compiler directives are introduced by the backtick (`)
- Its declaration can be put outside of the module and its scope may not be limited to a single module.

Compiler directives	Description
<code>`define</code>	To define text macros. (Similar to <code>#define</code> in C language)
<code>`include</code>	To include entire content from another Verilog file into the existing file during compilation. (Similar to <code>#include</code> in C language).
<code>`ifdef...`endif</code> <code>`ifndef...`else...`endifn</code>	Conditional compiler directives that behave as if..else conditional statements.
<code>`timescale</code>	To specify time units and precision for the module