What is it?

- Features inherited from Verilog and C++.
- Decrease gap between design and verification language.
- It’s an integrated part of the simulation tool.
- No external tool, GUI, or interface (such as PLI) to run it.
- Adoption process of System Verilog is very fast.
History


- After acquired by Cadence (1989) it was made as a an open standard (1990).


- System Verilog streamlines many of the annoyances of Verilog.

- Adds high level programming language features.
Basics of HDL

module sillyfunction(input logic a, b, c, output logic y);
assign y = ~a & ~b & ~c |
a & ~b & ~c |
a & ~b & c;
endmodule
Module Layout in Verilog

module module_name (port_list);

declarations:
    port declaration (input, output, inout, ...)
data type declaration (reg, wire, parameter, ...)
task and function declaration

statements:
    initial block
    always block
    module instantiation
    gate instantiation
    UDP instantiation
    continuous assignment

endmodule
Combinational Logic

Depends only on the current input and has no memory.

/* an array of inverters */
module invA4 ( input logic [3:0] a, output logic [3:0] y);
    assign y = ~a;
endmodule

// full adder
module fa ( input logic a, b, cin, output logic s, cout);
    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (cin & (a ^ b));
endmodule
What happened to reg and wire?

We know if a signal appears on the left-hand side of $<=$ or $=$ in an `always` block, it must be declared as `reg`. Otherwise, it should be declared as `wire`. Also, input and output ports default to the `wire` type unless their type is explicitly specified as `reg`.

System Verilog introduces the `logic` type. Logic is a synonym for `reg` and avoids misleading users about whether it is actually a flip-flop.

Also `logic` can be used outside `always` blocks where a `wire` traditionally would be required. Thus nearly all signals can be `logic`.

The exception is that signals with multiple drivers (i.e. tristate bus) must be declared as a net, as we described earlier.
Combinational Logic

module gates(input logic [3:0] a, b, output logic [3:0] y1, y2, y3, y4, y5);
/* Five different two-input logic gates acting on 4 bit busses */
assign y1 = a & b; // AND
assign y2 = a | b; // OR
assign y3 = a ^ b; // XOR
assign y4 = ~(a & b); // NAND
assign y5 = ~(a | b); // NOR
endmodule

-Multipliers and shifters use substantially more area
-Division and modulus in hardware is so costly that it may not be synthesizable
Combinational Logic

module tristate(input logic [3:0] a, input logic en, output tri [3:0] y);
assign y = en ? a : 4'bz;
endmodule

Bit Swizzling:

assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101}
Combinational Logic

// mux2 using tristate
module mux2 (input logic [3:0] d0, d1,
    input logic s,
    output tri [3:0] y);
tristate  t0(d0, ~s, y);
tristate  t1(d1, s, y);
mux2  lowmux(d0, d1, s[0], low);
mux2  highmux(d2, d3, s[0], high);
mux2  finalmux(low, high, s[1], y);
endmodule

module mux4 (input logic [3:0] d0, d1, d2, d3,
    input logic [1:0] s,
    output logic [3:0] y);
logic [3:0] low, high;
mux2  lowmux(d0, d1, s[0], low);
mux2  highmux(d2, d3, s[0], high);
mux2  finalmux(low, high, s[1], y);
endmodule
module tristate(input logic [3:0] a, input logic en, output tri [3:0] y);
assign y = en ? a : 4'bz;
endmodule
New **always** statements

Previously (in Verilog) in an **always** block, forgetting an **else** leads to an un-intended latch. To avoid this mistake, System Verilog adds the following specialized statements: **always_comb**, **always_latch** and **always_ff**.

**always_comb**: modeling combinational logic behavior.

Example:

```
always_comb
a = b & c;
```

- Even though **always_comb** is similar to verilogs **always@ ***, it is different in that **always_comb** executes once at time zero where **always@ *** waits until a change occurs on a signal in the inferred sensitive list.

- Sensitive to change within the contents of a function where as **always@ *** is only sensitive to changes to the arguments of a function.
**always_latch**: modeling latched logic behavior.

Example:

```vhdl
always_latch
  if (clk)
    q <= d;
```

-The *always_latch* procedure determines its sensitivity and executes identically to *always_comb* procedure.

-Software tools can perform additional checks to warn if the behavior within an *always_latch* procedure does not represent latched logic.

**always_ff**: model synthesizable sequential logic behavior

Example:

```vhdl
always_ff @ (posedge clk iff reset == 0 or posedge reset)
begin
  r1 <= reset ? 0 : r2 + 1;
end
```

-Imposes restriction that it contains one and only one event control and no blocking timing controls.

-Software tools can perform additional checks to warn if the behavior within an *always_ff* procedure does not represent sequential logic.
Sequential Logic

A logic circuit whose output depends not only on the present input but also on the history of the input.

```
module flop (C, D, Q);
  input C, D;
  output Q;
  reg Q;
  always @(posedge C)
    begin
      Q <= D;
    end
  endmodule

module flopr(input logic clk,
               input logic reset,
               input logic [3:0] d,
               output logic [3:0] q);
  // synchronous reset
  always_ff @(posedge clk)
    begin
      if (reset) q <= 4'b0;
      else q <= d;
    end
  endmodule
```
module flopr_a(input logic clk,
    input logic reset,
    input logic [3:0] d,
    output logic [3:0] q);

    // asynchronous reset
    always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else if (en) q <= d;

    endmodule

Sequential Logic
module counter(input logic clk,
           input logic reset,
           output logic [3:0] q);
logic [3:0] nextq;
flopr qflop (clk, reset, nextq, q);
adder inc(q, 4'b0001, nextq);
endmodule
Combinational Logic with Always Statements

- `always_comb` executes once at time zero where `always@ *` waits until a change occurs on a signal in the inferred sensitive list.

- Re-evaluates the statements inside the always block anytime any of the signals on the right hand side of `<=` (non-blocking) or `=` (blocking) inside the always statement change.

- This is preferred way of describing combinational logic.
Combinational Logic

Depends only on the current input and has no memory.

/* an array of inverters */
module invA4 (input logic [3:0] a, output logic [3:0] y);
    assign y = ~a;
endmodule

// full adder
module fa (input logic a, b, cin, output logic s, cout);
    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (cin & (a ^ b));
endmodule
Non-Blocking

If the following guidelines are not followed, it is possible to write code that appears to work in simulation, but synthesizes to incorrect hardware.

-Using `always_ff @(posedge clk)` and non blocking assignments to model synchronous sequential logic.

-A group of non-blocking assignments is evaluated concurrently: All of the expressions on the right-hand sides are evaluated before any of the left hand sides are updated.

-Use continuous assignments to model simple combinational logic.

```verilog
always_ff @(posedge clk)
begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
end
assign y = s ? d1 : d0;
```
Blocking

```vhdl
always_comb
begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
end
```

- Using `always_comb` and blocking assignments to model more complicated combinational logic where the always statement is helpful.

- Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

- A group of blocking assignments are evaluated in the order they appear in the code, just as one would except in a standard programming language.
Testbenches

Testbenches is an HDL module used to test another module, called the device under test (DUT). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called test vectors.

```verilog
module testbench1;
logic a, b, c;
logic y;
endmodule

// instantiate device under test
sillyfunction dut(a, b, c, y);

// apply inputs one at a time
initial begin
  a = 0; b = 0; c = 0; #10;
  c = 1; #10;
  b = 1; c = 0; #10;
  c = 1; #10;
  a = 1; b = 0; c = 0; #10;
  c = 1; #10;
  b = 1; c = 0; #10;
  c = 1; #10;
end

module sillyfunction(input logic a, b, c,
                      output logic y);
assign y = ~a & ~b & ~c |
           a & ~b & ~c |
           a & ~b & c;
endmodule
```

enable this statement at the beginning of simulation and execute it only once

module testbench1();
logic a, b, c;
logic y;
endmodule

// instantiate device under test
sillyfunction dut(a, b, c, y);

// apply inputs one at a time
initial begin
  a = 0; b = 0; c = 0; #10;
  c = 1; #10;
  b = 1; c = 0; #10;
  c = 1; #10;
  a = 1; b = 0; c = 0; #10;
  c = 1; #10;
  b = 1; c = 0; #10;
  c = 1; #10;
end
endmodule
module sillyfunction(input logic a, b, c, output logic y);
  assign y = ~a & ~b & ~c |
           a & ~b & ~c |
           a & ~b & c;
endmodule

module testbench2();
  logic a, b, c;
  logic y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1; #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("010 failed.");
    c = 1; #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1; #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("110 failed.");
    c = 1; #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
THANK YOU

Reference: CMOS VLSI Design A Circuit And Systems Perspective:
By Neil H.E. Weste and David Money Harris
module divideby3FSM(input logic clk,  
    input logic reset,  
    output logic y);  
logic [1:0] state, nextstate;  

// State Register  
always_ff @(posedge clk)  
    if (reset) state <= 2'b00;  
    else state <= nextstate;  

// Next State Logic  
always_comb  
    case (state)  
        2'b00: nextstate = 2'b01;  
        2'b01: nextstate = 2'b10;  
        2'b10: nextstate = 2'b00;  
        default: nextstate = 2'b00;  
    endcase  

// Output Logic  
assign y = (state == 2'b00);  
endmodule
module historyFSM(input logic clk, 
   input logic reset, 
   input logic a, 
   output logic x, y);

typedef enum logic [2:0] 
   {S0, S1, S2, S3, S4} statetype;

logic [2:0] state, nextstate;

always_ff @(posedge clk)
if (reset) state <= S0;
else state <= nextstate;

always_comb
  case (state)
    S0: if (a) nextstate = S3;
        else nextstate = S1;
    S1: if (a) nextstate = S3;
        else nextstate = S2;
    S2: if (a) nextstate = S3;
        else nextstate = S2;
    S3: if (a) nextstate = S4;
        else nextstate = S1;
    S4: if (a) nextstate = S4;
        else nextstate = S1;
    default: nextstate = S0;
  endcase

//output logic
  assign x = (state[1] & ~a) | 
            (state[2] & a);
  assign y = (state[1] & state[0] & ~a) | 
            (state[2] & state[0] & a);
endmodule