Verilog Tutorial

Verilog Fundamentals
- History
- Data types
- Structural Verilog
- Functional Verilog

Originally designers used manual translation + bread boards for verification

Hardware design languages enabled logic level simulation and verification
Once design were written in HDLs tools could be used for automatic translation

Primary Verilog data type is a bit-vector where bits can take on one of four values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero</td>
</tr>
<tr>
<td>1</td>
<td>Logic one</td>
</tr>
<tr>
<td>X</td>
<td>Unknown logic value</td>
</tr>
<tr>
<td>Z</td>
<td>High impedance, floating</td>
</tr>
</tbody>
</table>

An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.

The Verilog keyword wire is used to denote a standard hardware net

Verilog includes ways to specify bit literals in various bases

wire [15:0] instruction;
wire [15:0] memory_req;
wire [7:0] small_net;

Absolutely no type safety when connecting nets!

4'b10_11

- Binary literals
  - 8'b0000_0000
  - 8'b0xx0_1xx1
- Hexadecimal literals
  - 32'h0a34_def1
  - 16'haxxx
- Decimal literals
  - 32'd42

We’ll learn how to actually assign literals to nets a little later
Verilog Basics

- Data types
- Structural Verilog
- Functional Verilog

A Verilog module includes a module name and a port list

module adder( A, B, cout, sum );
input [3:0] A;
input [3:0] B;
output cout;
output [3:0] sum;
endmodule

// HDL modeling of
// adder functionality

A module can instantiate other modules creating a module hierarchy

module FA( input a, b, cin 
output cout, sum );

// HDL modeling of 1 bit
// full adder functionality

endmodule
**A module can instantiate other modules creating a module hierarchy**

```verilog
defmodule adder( input [3:0] A, B, output cout, output [3:0] S );
wire c0, c1, c2;
FA fa0( ... );
FA fa1( ... );
FA fa2( ... );
FA fa3( ... );
endmodule
```

**Verilog supports connecting ports by position and by name**

**Connecting ports by ordered list**

```verilog
FA fa0( A[0], B[0], 'b0, c0, S[0] );
```

**Connecting ports by name (compact)**

```verilog
FA fa0( .a(A[0]), .b(B[0]),
cin('b0), cout(c0), sum(S[0]) );
```

**Connecting ports by name**

```verilog
FA fa0
{
  .a (A[0]),
  .b (B[0]),
cin (1'b0),
  cout (c0),
  sum (S[0])
};
```

---

**Let’s review how to turn our schematic diagram into structural Verilog**

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.
Let's review how to turn our schematic diagram into structural Verilog

module adder (input [3:0] A, B, output [3:0] s);
wire c0, c1, c2;
FA fa0(A[3], B[3], 1'b0, c0, c[0]);
FA fa1(A[2], B[2], c0, c1, s[1]);
FA fa2(A[1], B[1], c1, s[2]);
FA fa3(A[0], B[0], c2, s[3]);
endmodule

Verilog Fundamentals
- Data types
- Structural Verilog
- Functional Verilog
  - Gate level
  - Register transfer level
  - High-level behavioral

Functional Verilog can roughly be divided into three abstraction levels

Gate-level Verilog uses structural Verilog to connect primitive gates

module mux4( input a, b, c, d, input [1:0] sel, output out );
wire [1:0] sel_b;
not not0( sel_b[0], sel[0] );
not not1( sel_b[1], sel[1] );
wire n0, n1, n2, n3;
and and0( n0, c, sel[1] );
and and1( n1, a, sel_b[1] );
and and2( n2, d, sel[1] );
and and3( n3, b, sel_b[1] );
wire x0, x1;
nor nor0( x0, n0, n1 );
nor nor1( x1, n2, n3 );
wire y0, y1;
or or0( y0, x0, sel[0] );
or or1( y1, x1, sel[1] );
nand nand0( out, y0, y1 );
endmodule
Continuous assignments statements assign one net to another or to a literal

Explicit continuous assignment

```verilog
wire [15:0] netA;
wire [15:0] netB;
assign netA = 16'h3333;
assign netB = netA;
```

Implicit continuous assignment

```verilog
wire [15:0] netA = 16'h3333;
wire [15:0] netB = netA;
```

Using continuous assignments to implement an RTL four input multiplexer

```verilog
module max4( input a, b, c, d,
input [1:0] sel,
output out );
wire out, t0, t1;
assign t0 = ~( ( sel[1] & c ) | (~sel[1] & a ) );
assign t1 = ~( ( sel[1] & d ) | (~sel[1] & b ) );
assign out = ~( ( t0 | sel[0] ) & ( t1 | ~sel[0] ) );
endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!

Verilog RTL includes many operators in addition to basic boolean logic

```verilog
module max4( input a, b, c, d,
input [1:0] sel,
output out );
assign out = ( sel == 0 ) ? a :
               ( sel == 1 ) ? b :
               ( sel == 2 ) ? c :
               ( sel == 3 ) ? d
               'bx;
endmodule
```

Verilog RTL operators

```
<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>&amp;&amp;</td>
<td>|</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
</tr>
<tr>
<td>Logical</td>
<td>|</td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>&gt;=</td>
<td>&lt;</td>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>===</td>
<td>!==</td>
<td>!=</td>
<td>===</td>
</tr>
<tr>
<td>Bitwise</td>
<td>&amp;</td>
<td>^</td>
<td></td>
<td></td>
<td>^</td>
</tr>
</tbody>
</table>
```

Avoid (/ \* **) since they usually synthesize poorly
Always blocks have parallel inter-block and sequential intra-block semantics

module mux4( input a, b, c, d,
input [1:0] sel,
output out );

reg out, t0, t1;
always @( a or b or c or d or sel )
begin
    t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule

The always block is reevaluated whenever a signal in its sensitivity list changes

module mux4( input a, b, c, d,
input [1:0] sel,
output out );

reg out, t0, t1;

always @( a or b or c or d or sel )
begin
    t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule

The order of these procedural assignment statements does matter. They essentially happen sequentially!

Always blocks have parallel inter-block and sequential intra-block semantics

module mux4( input a, b, c, d,
input [1:0] sel,
output out );

reg out, t0, t1;
always @( a or b or c or d or sel )
begin
    t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule

What happens if we accidentally forget a signal on the sensitivity list?

Always blocks have parallel inter-block and sequential intra-block semantics

module mux4( input a, b, c, d,
input [1:0] sel,
output out );

reg out, t0, t1;
always @( a or b or c or X or sel )
begin
    t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!
Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4( input a, b, c, d
    input [1:0] sel,
    output out );

    reg out, t0, t1;

    always @(*)
    begin
        t0 = ~( (sel[1] & a) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) )
    end

endmodule
```

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block.

Continuous and procedural assignment statements are very different

Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

```verilog
wire out, t0, t1;
assign t0 = ~( (sel[1] & a) | (~sel[1] & a) )
assign t1 = ~( (sel[1] & d) | (~sel[1] & b) )
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) )
```

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```verilog
reg out, t0, t1, temp;
always @(*)
begin
    t0 = temp;
    t1 = temp;
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) )
end
```

Always blocks can contain more advanced control constructs

```verilog
module mux4( input a, b, c, d
    input [1:0] sel,
    output out );

    reg out;

    always @(*)
    begin
        if ( sel == 2’b0 )
            out = a;
        else if ( sel == 2’b1 )
            out = b;
        else if ( sel == 2’b2 )
            out = c;
        else if ( sel == 2’b3 )
            out = d
        else
            out = 1’hs;
    end

endmodule
```

What happens if the case statement is not complete?

```verilog
module mux3( input a, b, c
    input [1:0] sel,
    output out );

    reg out;

    always @(*)
    begin
        if sel == 3
            If sel = 3, mux will output the previous value.
        else
            What have we created?
```

```verilog
    case ( sel )
        2’b0 : out = a;
        2’b1 : out = b;
        2’b2 : out = c;
        default : out = 1’hs;
    endcase

endmodule
```
What happens if the case statement is not complete?

```verilog
module mux3(
    input a, b, c,
    input [1:0] sel,
    output out);

    reg out;
    always @( *)
    begin
        case ( sel )
            2'd0 : out = a;
            2'd1 : out = b;
            2'd2 : out = c;
            default : out = 1'b0;
        endcase
    end
endmodule
```

We can prevent creating state with a default statement.

So is this how we make latches and flip-flops?

```verilog
module latch

    (input clk,
    input d,
    output reg q);

    always @(clk)
    begin
        if (clk)
            q = d;
    end
endmodule
```

```verilog
module flipflop

    (input clk,
    input d,
    output q);

    always @(posedge clk)
    begin
        q = d;
    end
endmodule
```

To understand why we need to know more about Verilog execution semantics:

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
    A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk)
    B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk)
    C_out = C_in;
```

To understand why we need to know more about Verilog execution semantics:

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
    A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk)
    B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk)
    C_out = C_in;
```

Active Event Queue

- On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!
To understand why we need to know more about Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk)
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk)
C_out = C_in;

Active Event Queue

To understand why we need to know more about Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk)
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk)
C_out = C_in;

Active Event Queue

A evaluates, A_out changes, and as a consequence 1 is added to the event queue
To understand why we need to know more about Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk )
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk )
C_out = C_in;

Active Event Queue

A

1
B
2
C

Race Condition!

To understand why we need to know more about Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk )
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk )
C_out = C_in;

Active Event Queue

A

1
B
2
C

Event queue is emptied before we go to next clock cycle

To understand why we need to know more about Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk )
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk )
C_out = C_in;

Active Event Queue

A

1
B
2
C

We didn’t model what we expected due to Verilog execution semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk )
B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk )
C_out = C_in;

Active Event Queue

A

1
B
2
C

Event queue is emptied before we go to next clock cycle
Non-blocking procedural assignments add an extra event queue

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out <= A_in;
assign B_in = A_out + 1;

always @(posedge clk)
  B_out <= B_in;
assign C_in = B_out + 1;

always @(posedge clk)
  C_out <= C_in;
```

The order of non-blocking assignments does not matter

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out <= A_in;
assign B_in = A_out + 1;
assign C_in = B_out + 1;

always @(posedge clk)
  B_out <= B_in;
assign C_in = B_out + 1;
assign C_out <= C_in;
```

Common patterns for latch and flip-flop inference

```verilog
always @(clk or D)
begin
  if (clk)
    Q <= D;
end

always @(posedge clk)
begin
  Q <= D;
end

always @(posedge clk)
begin
  if (enable)
    Q <= D;
end
```
Writing Good Synthesizable Verilog

- Only leaf modules should have functionality
  - All other modules are strictly structural, i.e., they only wire together sub-modules
- Use only positive-edge triggered flip-flops for state
- Do not assign to the same variable from more than one always block
- Separate combinational logic from sequential logic
  - Combinational logic described using `always @(*)` and blocking = assignments and `assign` statements
  - Sequential logic described with `always @(posedge clk)` and non-blocking <= assignments

```verilog
assign C_in = B_out + 1;  // Combinational logic
always @(posedge clk)  // Update state only
  C_out <= C_in;        // don't be tempted to add
```  
  combinational logic into clocked always block)

Verilog can be used to model the high-level behavior of a hardware block

```verilog
module factorial( input [7:0] in, output reg [15:0] out );
begin
  integer num_calls;
  initial num_calls = 0;
  integer multiplier;
  integer result;
  always @(*)
  begin
    multiplier = in;
    result = 1;
    while ( multiplier > 0 ) begin
      result = result * multiplier;
      multiplier = multiplier - 1;
    end
    out = result;
    num_calls = num_calls + 1;
  end
endmodule
```

Behavioral Verilog is used to model the abstract function of a hardware

- Characterized by heavy use of sequential blocking statements in large always blocks
- Many constructs are not synthesizable but can be useful for behavioral modeling
  - Data dependent for and while loops
  - Additional behavioral datatypes: `integer, real`
  - Magic initialization blocks: `initial`
  - Magic delay statements: `#<delay>

Delay statements should only be used in test harnesses

```verilog
module mux4
begin
  input a, b, c, d;
  input [1:0] sel;
  output out;
  wire #10 t0 = ~( (sel[1] & c) | (~sel[1] & a) );
  wire #10 t1 = ~( (sel[1] & d) | (~sel[1] & b) );
  wire #10 t10 out = ~( (t10 | sel[0]) & (tl | ~sel[0]) );
endmodule
```

Although this will add a delay for simulation, these are ignored in synthesis
System tasks are used for test harnesses and simulation management

```verilog
test [ 1023:0 ] exe_filename;
initial
begin
  // This turns on VCD (plus) output
  $vcdpluson(0);
  // This gets the program to load into memory from the command line
  if ($value$plusargs("-exec=", exe_filename) )
    $readmemh(exe_filename, mem.m);
  else
    begin
      display("ERROR: No executable specified! (use -exec=filename)"");
      $finish;
    end
  // Strobe reset
  #0 reset = 1;
  #38 reset = 0;
end
```

Which abstraction is the right one?

- High-Level Behavioral
- Register Transfer Level
- Gate Level

A common approach is to use C/C++ for initial behavioral modeling, and for building test benches.

Examples

A module can be described in many different ways but it should not matter from outside

Example: mux4

Mux4: Gate-level structural Verilog

```
module mux4(input a, b, c, d, input [1:0] sel, output out);
    wire [1:0] sel;
    not sel0(sel_h[0], sel[0]);
    not sel1(sel_h[1], sel[1]);
    wire d0, d1, d2, d3,
    and sel0(d0, a, sel[0] );
    and sel1(d1, b, sel[1] );
    wire d2, d3, d5, d6,
    and sel0(d2, c, sel[0] );
    and sel1(d3, b, sel[1] );
    wire x0, x1, x2, x3, x4,
    not x0(a, x0, sel[0] );
    not x1(b, x1, sel[1] );
    wire y0, y1, y2, y3, y4,
    or eval(y0, x0, y0, sel[0] );
    or eval(y1, x1, y1, sel[1] );
    assign y2 = d2;
    assign y3 = d3;
    assign y4 = d4;
endmodule
```
Mux4: Using continuous assignments

```verilog
module mux4( input a, b, c, d,
            input [1:0] sel,
            output out );

  wire out, t0, t1;
  assign t0 = ~( sel[1] & c ) | (~sel[1] & a );
  assign t1 = ~( sel[1] & d ) | (~sel[1] & b );
  assign out = ~( t0 | sel[0] ) & ( t1 | ~sel[0] );

endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!

---

Mux4: Behavioral style

```verilog
// Four input multiplexer
module mux4( input a, b, c, d,
            input [1:0] sel,
            output out );

  assign out = { sel == 0 ) ? a : ( sel == 1 ) ? b : ( sel == 2 ) ? c : ( sel == 3 ) ? d ;

endmodule
```

If input is undefined we want to propagate that information.

---

Mux4: Using always block

```verilog
module mux4( input a, b, c, d,
            input [1:0] sel,
            output out );

  reg out, t0, t1;
  always @( a or b or c or d or sel ) begin
    t0 = ~( sel[1] & c ) | (~sel[1] & a );
    t1 = ~( sel[1] & d ) | (~sel[1] & b );
    out = ~( t0 | sel[0] ) & ( t1 | ~sel[0] );
  end

endmodule
```

The order of these procedural assignment statements does matter. They essentially happen sequentially!

---

Mux4: Always block permit more advanced sequential idioms

```verilog
module mux4( input a, b, c, d,
            input [1:0] sel,
            output out );

  reg out;
  always @( * ) begin
    case ( sel )
      0'd0 : out = a;
      0'd1 : out = b;
      0'd2 : out = c;
      0'd3 : out = d;
      default : out = 1'hx;
    endcase
  end

endmodule
```

Typically we will use always blocks only to describe sequential circuits.
**Parametrized mux4**

```verilog
module mux4 #(parameter WIDTH = 1) (input [WIDTH-1:0] a, b, c, d, sel, out);
begin
    assign out = (sel[1:0] == 0) ? a : b; // Default value
endmodule
```

**Flip-flops**

```verilog
module FF (input clk, input d, output q);
    always @(posedge clk)
        begin
            q <= d;
        end
endmodule
```

**Register**

```verilog
module register #(parameter WIDTH = 1) (input clk, input [WIDTH-1:0] d, input en, output [WIDTH-1:0] q);
    always @(posedge clk or negedge resetN)
        begin
            if (~resetN)
                q <= 0;
            else if (enable)
                q <= d;
        end
endmodule
```
Register in terms of Flip-flops

Do they behave the same? yes

Static Elaboration: Generate

Implementing the control logic finite state machine in Verilog

A simple state machine for valid/ready signals
Implementing the control signal outputs for the finite state machine

```verilog
always @(*) begin
// default control signals
A_mux_sel = A_MUX_SEL();
A_en = 1'b0;
Bmux_sel = B_MUX_SEL();
B_en = 1'b0;
input_available = 1'b1;
end

WAIT : begin
A_mux_sel = A_MUX_SEL_IN;
A_en = 1'b1;
B_mux_sel = B_MUX_SEL_IN;
B_en = 1'b1;
input_available = 1'b1;
end

CALC : if (A_l0_B) begin
A_mux_sel = A_MUX_SEL_B;
A_en = 1'b1;
B_mux_sel = B_MUX_SEL_A;
B_en = 1'b1;
else if (!B_zero)
A_mux_sel = A_MUX_SEL_SUB;
A_en = 1'b1;
end

DONE : result_ready = 1'b1;
endcase
```

Implementing the state transitions for the finite state machine

```verilog
always @(*) begin
// Default is to stay in the same state
state_next = state;

case (state)
    WAIT : if (input_available)
        state_next = CALC;
    CALC : if (!B.zero)
        state_next = DONE;
    DONE : if (result_taken)
        state_next = WAIT;
endcase
```

Take away points

- Structural Verilog enables us to describe a hardware schematic textually
- Verilog can model hardware at three levels of abstraction: **gate level**, **register transfer level**, and behavioral
- Understanding the Verilog execution semantics is critical for understanding blocking vs non-blocking assignments
- Designers must have the hardware they are trying to create in mind when they write their Verilog
- Parameterized models provide the foundation for reusable libraries of components
- Use explicit state to prevent unwanted state inference and to more directly represent the desired hardware