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:

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

Absolutely no type safety when connecting nets!

Verilog includes ways to specify bit literals in various bases:

- Binary literals:
  - 8'b0000_0000
  - 8'b00x0_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;

// HDL modeling of adder functionality
endmodule

Note the semicolon at the end of the port list!

A Verilog module includes a module name and a port list:

Traditional Verilog-1995 Syntax

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

ANSI C Style Verilog-2001 Syntax

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

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
module adder( input [3:0] A, B, 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], 1'b0, c0, S[0] );
```

Connecting ports by name (compact)

```verilog
FA fa0( a(A[0]), b(B[0]),
        cin(1'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

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

- Behavioral Algorithm
- Register Transfer Level
- Gate Level

Verilog RTL includes many operators in addition to basic boolean logic:

```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

// Simple four bit adder
module adder(input [3:0] op1, op2, output [3:0] sum);
assign sum = op1 + op2;
endmodule
```

Gate-level Verilog uses structural Verilog to connect primitive gates:

```verilog
module mux4(input a, b, c, d, input [1:0] sel, output out);
wire [1:0] sel_b1;
not not0(sel_b1, sel[1]);
not not1(sel_b1, sel[0]);
wire s0, s1, s2, s3;
and and0(s0, c, sel[1]);
and and1(s1, a, sel[1]);
and and2(s2, d, sel[1]);
and and3(s3, b, sel[1]);
wire x0, x1,
nor nor0(x0, s0, s1);
nor nor1(x1, s2, s3);
wire y0, y1,
or or0(y0, x0, x1, sel[0]);
or or1(y1, x1, sel[1], y0);
and and0(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 mux4(input a, b, c, d, input [1:0] sel, output out);
wire out, x0, x1;
nor0 = (sel[1] & a) | (!sel[1] & x);
nor1 = (sel[1] & d) | (!sel[1] & b);  
out = !(!x0 & !x1 & !sel[0]);
endmodule
```

Verilog RTL operators:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Reduction</th>
<th>Logical</th>
<th>Bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - / * %</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>&amp;</td>
<td></td>
</tr>
</tbody>
</table>

- `wire [3:0] net1 = 4'b00ax;`
- `wire [3:0] net2 = 4'b1101;`
- `wire [11:0] net3 = 4'b0110net1, net2 ;`
- `wire equal = ( net3 == 12'b0000_110_000x );`

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

```verilog
module mux(
    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

Continuous and procedural assignment statements are very different

```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
```

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] & c | ~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
module mux5(
    input a, b, c, d,
    input [1:0] sel,
    output out);

    reg out, t0, t1, temp;

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

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!

What happens if we accidentally forget a signal on the sensitivity list?
Always blocks can contain more advanced control constructs

```verilog
module mux3( input a, b, c, d, input [1:0] sel, output out );
reg out;
always @(*)
begin
  if ( sel == 2'd1 )
    out = a;
  else if ( sel == 2'd2 )
    out = b;
  else if ( sel == 2'd3 )
    out = c;
  else
    default: out = 1'ba;
endcase
endmodule
```

What happens if the case statement is not complete?

```verilog
module mux3(input a, b, c, d, input [1:0] sel, output out);
reg out;
always @(*)
begin
  if ( sel == 2'd1 )
    out = a;
  else if ( sel == 2'd2 )
    out = b;
  else if ( sel == 2'd3 )
    out = c;
  else
    default: out = 1'ba;
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
  if ( sel == 2'd1 )
    out = a;
  else if ( sel == 2'd2 )
    out = b;
  else
    default: out = 1'ba;
endcase
endmodule
```

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

```verilog
wire A_im, B_in, C_in;
reg A_out, B_out, C_out;
always @(posedge clk)
  A_out = A_im;
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;
```

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

```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

- **A** evaluates, **B** evaluates, and as a consequence 1 is added to the event queue

---

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

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

Non-blocking procedural assignments add an extra event queue

Non-blocking procedural assignments add an extra event queue

The order of non-blocking assignments does not matter

Common patterns for latch and flip-flop inference
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 combinatorial logic from sequential logic
  - Combinational logic described using always @(*) and blocking = assignments and assigns statements
  - Sequential logic described with always @posedge and non-blocking = assignments
  
```
assign C_in = B_out + 1;  // Combinational logic
always @(posedge clk)  // Update of state only
  C_out <= C_in;  // (don't be tempted to add
                  // combinatorial logic into
clocked always block)
```

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>

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 );
    integer num_cells;
    integer num_cells = 6;
    integer multiplier;
    integer result;
    integer @() begin
        multiplier = in;
        result = 1;
        while ( multiplier > 0 ) begin
            result = result * multiplier;
            multiplier = multiplier - 1;
            end
        out = result;
        num_cells = num_cells + 1;
    end
endmodule
```

Delay statements should only be used in test harnesses

```verilog
module mux4
    (    input a,    // initial statement
         input b,    // Variables of type integer
         input c,    // Data dependent
         input d,    // while loop
         input [1:0] sel,    // Although this will add a delay for
         output out )    // simulation, these are ignored in
     ;
    wire #10 w0 = ( sel[1] & c ) | (~sel[1] & a );
    wire #10 w1 = ( sel[1] & d ) | (~sel[1] & b );
    wire #10 w2 = ( ~0 | sel[0] ) & ( ~1 | ~sel[0] );
    endmodule
```

System tasks are used for test harnesses and simulation management

```verilog
reg [ 20:0 ] error_filename;
initial
    begin
    // This turns on VCD (gdb) output
    $display("\n"); // This gets the program to load into memory from the command line
    if ( $valueof("error", error_filename ) )
        $fdisplay("error_filename, error_filename ");
    begin
        $display( "ERROR: No executable specified! (use error_filename)" );
    $finish;
end
// Extra reset
#5 reset = 1;
#3 reset = 0;
end
```

Which abstraction is the right one?

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

Designers usually use a mix of all three! Early on in the design process they might use mostly behavioral models. As the design is refined, the behavioral models begin to be replaced by dataflow models. Finally, the designers use automatic tools to synthesize a low-level gate-level model.

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

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

Example: mux4

Mux4: Gate-level structural Verilog

Mux4: Using continuous assignments

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

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

Mux4: Behavioral style

```
// Four input multiplexer
module mux4( input a, b, c, d, input [1:0] sel, output out );
    assign out = ( sel[1] == 0 ) ? a : ( sel[1] == 1 ) ? b :
                 ( sel[0] == 2 ) ? c : ( sel[0] == 3 ) ? d :
                 1'b0;
endmodule
```

If input is undefined we want to propagate that information.

Mux4: Using always block

```
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[0] & a );
            t1 = ~( sel[1] & d ) & ~( sel[0] & b );
            out = ~( t0 | sel[0] ) & ( t1 | sel[1] );
        end
endmodule
```

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

Mux4: Always block permit more advanced sequential idioms

```
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
            if ( sel[1] == 2'b1 )
                t0 = a;
            else if ( sel[1] == 2'b0 )
                t0 = b;
            else if ( sel[0] == 2'b0 )
                t0 = c;
            else if ( sel[0] == 2'b1 )
                t0 = d;
            sel = t0;
            sel = ~sel;
            t0 = ~t0;
            out = 1'b1;
        end
endmodule
```

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

```verilog
module mux4 #(parameter WIDTH = 1) (input W[WIDTH-1:0] a, b, c, d,
output W[WIDTH-1:0] z, enable);

wire [WIDTH-1:0] out, z0, z1;
assign z0 = W[0] ? ~z : a;
assign z1 = W[1] ? ~z : a;
assign out = W[0] ? z1 : z0;
endmodule
```

**Flip-flops**

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

**Register in terms of Flip-flops**

```verilog
module register #(parameter WIDTH = 1) (input clk, input [WIDTH-1:0] d, output [WIDTH-1:0] q);

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

**Static Elaboration: Generate**

```verilog
module register #(parameter WIDTH = 1) (input clk, input [WIDTH-1:0] d, input en, output [WIDTH-1:0] q);

genvar i;
generate
  for (i = 0; i < WIDTH; i = i + 1)
  begin
    regE FF_ff(.clk(clk), .d(d[i]), .en(en), .q(q[i]));
  end
endgenerate
endmodule
```
A simple state machine for valid/ready signals

Implementing the control logic finite state machine in Verilog

Implementing the control signal outputs for the finite state machine

Implementing the state transitions for the finite state machine

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