Verilog HDL (VHDL :-)

- History & main concepts - structure, description styles
- Data types, declarations, operations
- Procedural & assignment statements
- If-then, case & loop statements
- Functional hierarchy - tasks & functions
- Time & events
- Parallelism; fork, join & disable statements
- Structural & behavioral descriptions
- Synthesizability & advanced topics

Michael John Sebastian Smith, "Application-Specific Integrated Circuits." Addison-Wesley.
http://www.edacafe.com/books/ASIC/ASICs.php [see ch. 11]

Ken Coffman, "Real world FPGA design with Verilog." Prentice Hall [2000]

Donald E. Thomas, Philip R. Moorby, "The Verilog® Hardware Description Language." Kluwer Academic Publishers.

James M. Lee, "Verilog Quickstart : a practical guide to simulation and synthesis in Verilog." Kluwer Academic Publishers.

TTÜ1918
Department of Computer Engineering

History

- Invented as a simulation language
- 1983/85 - Automated Integrated Design Systems (later as Gateway Design Automation)
- 1989/90 - acquired by Cadence Design Systems
- 1990/91 - opened to the public in 1990 - OVI (Open Verilog International) was born
- 1992 - the first simulator by another company
- 1993 - IEEE working group (under the Design Automation Sub-Committee) to produce the IEEE Verilog standard 1364
- May 1995 - IEEE Standard 1364-1995
- 2001 - IEEE Standard 1364-2001 - revised version
- Verilog-2005, System Verilog (2005), Verilog-AMS
- WWW
  - http://www.project-veripage.com/
  - http://www.angelfire.com/ca/verilog/
  - http://www.angelfire.com/in/verilogfaq/
- news
  - comp.lang.verilog

Hello, world!

```
module world;

  initial
    begin
      $display ( "Hello, world!" );
    end

endmodule
```

- ModelSim

```
run -all
# Hello, world!
```


Main Concepts

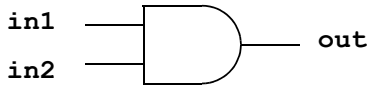
- Modules
  - modules
  - functions & tasks
- Case sensitive
  - lower case keywords
  - identifier - a sequence of letters, digits, dollar sign ($), and underscore (_)

```
identifier ::= simple_identifier | escaped_identifier
simple_identifier ::= [a-zA-Z][a-zA-Z_0-9$]*
escaped_identifier ::= \{any_ASCII_character_except_white_space} white_space
```

- No delta-delay
  - non-deterministic parallelism

Module

module *name* **(** *input_output_list* **);**
module_body
endmodule



Ports:
wire - by default (can be skipped)
reg ~ keeps content

```
// structural
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2, out;
    and u1 (out, in1, in2);
endmodule
```



```
// behavioral
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2;
    reg out;
    always @( in1 or in2 )
        out = in1 & in2;
endmodule
```

```
// data flow
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2, out;
    assign out = in1 & in2;
endmodule
```

```
module test_and2;
    reg i1, i2;    wire o;

    AND2 u2 (i1, i2, o);

    initial begin
        i1 = 0; i2 = 0;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 0; i2 = 1;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 1; i2 = 0;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 1; i2 = 1;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
    end
endmodule
```

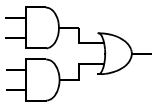
- always
- initial
- begin ... end

```
i1 = 0, i2 = 0, o = 0
i1 = 0, i2 = 1, o = 0
i1 = 1, i2 = 0, o = 0
i1 = 1, i2 = 1, o = 1
```

Results...

Example AND-OR

```
module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    wire tmp;
    and #10 u1 (tmp, in1, in2),
            u2 (undec, in3, in4);
    or #20 (out, tmp, undec);
endmodule
```



```
module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    wire tmp;
    assign #10 tmp = in1 & in2;
    wire #10 tmp1 = in3 & in4;
    assign #20 out = tmp | tmp1;
    // assign #30 out = (in1 & in2) | (in3 & in4);
endmodule
```

```
module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    reg out;

    always @(in1 or in2 or in3 or in4) begin
        if (in1 & in2)
            out = #30 1;
        else
            out = #30 (in3 & in4);
    end
endmodule
```


```
module test_and_or;
    reg r1, r2, r3, r4;
    wire o;

    and_or u2 (.in2(r2), .in1(r1), .in3(r3), .in4(r4), .out(o));

    initial begin : b1
        reg [4:0] i1234;
        for ( i1234=0; i1234<16; i1234=i1234+1 ) begin
            { r1, r2, r3, r4 } = i1234[3:0];
            #50 $display("r1r2r3r4=%b%b%b%b, o=%b",r1,r2,r3,r4,o);
        end
    end
endmodule
```

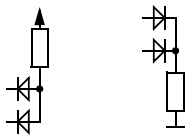
Data Types

- **Constants - decimal, hexadecimal, octal & binary**
  - **Format** `<width>'<radix><value>`
    - **<width> - optional, in bits, decimal constant**
    - **<radix> - optional, base, can be one of `b`, `B`, `d`, `D`, `o`, `O`, `h` or `H`**
    - **<value> - a sequence of symbols depending on the radix:**
      binary - `0`, `1`, `x`, `X`, `z` & `Z`
      octal - also `2`, `3`, `4`, `5`, `6` & `7`
      hexadecimal - also `8`, `9`, `a`, `A`, `b`, `B`, `c`, `C`, `d`, `D`, `e`, `E`, `f` & `F`
      decimal - `0` to `9`, but not `x` or `z`

```
15       (decimal 15)
'h15     (decimal 21, hex 15)
5'b10011   (decimal 19, binary 10011)
12'h01F    (decimal 31, hex 01F)
'b01x      (no decimal value, binary 01x)
```

  - **String constants, e.g. "my-string"**
    - **are converted to their ASCII equivalent binary format, e.g. `"ab" == 16'h5758`**
  - **Real constants - ordinary scientific notation**
    - **e.g.** `22.73, 12.8e12`

- **Physical data types**
  - **binary nets - `wire`, `wand`, `wor`, etc., and**
    - **continuously driven**
  - **registers - `reg`**
    - **"remembers" the last assignment**



  - **Registers can be assigned only inside behavioral instances**
  - **Nets are driven all the time and cannot be assigned in behavioral block**
  - **Register can be interpreted as a storage element (latch, flip-flop) but not necessarily**
  - **Nets & registers are interpreted as unsigned integers**

- **Abstract data types**
  - **`integer` - almost as a 32-bit `reg` but signed**
  - **`time` - 64-bit unsigned integer**
  - **`real` - floating point, platform depending**
  - **`event` - a special variable without value, used for synchronization**
  - **`parameter` - "named constant", set before simulation starts**

Declarations

- **Width in bits - physical variables only**
- **Arrays - only types *integer*, *real* and *reg***

```
integer i, j;
real f, d;
wire [7:0] bus;          // 1x8 bits
reg [0:15] word;         // 1x16 bits
reg arr[0:15];           // 16x1 bits
reg [7:0] mem[0:127];    // 128x8 bits
event trigger, clock_high;
time t_setup, t_hold;
parameter width=8;
parameter width2=width*2;
wire [width-1:0] ww;
// The following are illegal
wire w[0:15];            // No arrays
wire [3:0] a, [7:0] b;   // Only one width per decl.
```


Operations

```
+  -  *  /  %      (arithmetic)
>  >=  <  <=       (relational)
!  &&  ||          (logical)
==  !=             (logical equality)
?:                 (conditional)
{}                 (concatenate)
===  !==           (case equality)
~  ^  ^~  &  |     (bit-wise)
<<  >>             (shift)
```

```
+  -  !  ~     (highest)
*  /  %
+  -           (binary op.)
<<  >>
<  <=  >  >=
=  ==  !=
===  !==
&  ~&
^  ^~
|  ~|
&&
||
?:             (lowest)
```

Bit-wise as unary operations

```
^word === 1'bx
&word === 0
```

Comparisons

```
'bx == 'bx      ≡ x
'bx === 'bx     ≡ 1
```

Concatenation

```
{2'b1x, 4'h7} === 6'b1x0111
{cout, sum} = in1 + in2 + cin;
{sreg, out} = {in, reg};
{3{2'b01}} === 6'b010101
```

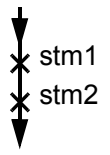
Indexing

```
reg [15:0] array [0:10];
reg [15:0] temp;
    ...
temp = array[3];
... temp[7:5] ...
// array[3][7:5] is illegal
```

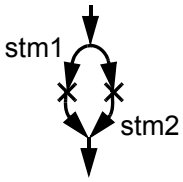

Procedural and Assignment Statements

- **Procedural statements**

```
begin
  stm1;
  stm2;
end
```




```
fork
  stm1;
  stm2;
join
```



- **Assignments**

```
lhs-expression = expression;
lhs-expression = #delay expression;
lhs-expression = @event expression;
```

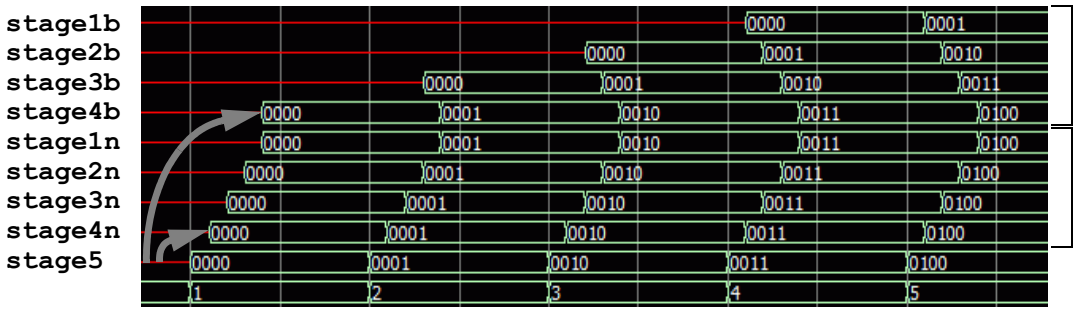
Blocking

```
lhs-expression <= expression;
lhs-expression <= #delay expression;
lhs-expression <= @event expression;
```

Non-blocking

```
always @(stage2b or stage3b or
    stage4b or stage5) begin
  stage1b = #1 stage2b;
  stage2b = #1 stage3b;
  stage3b = #1 stage4b;
  stage4b = #1 stage5;
end
```

```
always @(stage2n or stage3n or
    stage4n or stage5) begin
  stage1n <= #1 stage2n;
  stage2n <= #1 stage3n;
  stage3n <= #1 stage4n;
  stage4n <= #1 stage5;
end
```



Conditional Statements

```
if ( bool-expr )
  statement
else
  statement
```

```
case ( expr )
expr [, expr]* : statement
default: statement
endcase
```

- **Case**
  - **bit by bit comparison (like ===)**
  - **casez - 'z' is interpreted as don't care**
  - **casex - 'z' & 'x' are interpreted as don't care**

Loop Statements

```
module for_loop;
  integer i;
  initial
    for (i=0;i<4;i=i+1) begin
      ...
    end
endmodule
```

```
module while_loop;
  integer i;
  initial begin
    i=0;
    while (i<4) begin
      ...
      i=i+1;
    end
  end
endmodule
```


Loops (cont.)

```
module repeat_loop(clock);
  input clock;
  initial begin
    repeat (5)
      @(posedge clock);
    $stop;
  end
endmodule
```

```
module forever_loop(a,b,c);
  input a, b, c;
  initial forever begin
    @(a or b or c)
      if ( a+b == c ) $stop;
  end
endmodule
```

Functional Hierarchy

- **Tasks**

```
task tsk;
  input i1, i2;
  output o1, o2;
  $display("Task tsk, i1=%0b, i2=%0b",i1,i2);
  #1 o1 = i1 & i2;
  #1 o2 = i1 | i2;
endtask
```

- **Access:** `tsk(a,b,c,d);`
- **A task may have timing control construct**

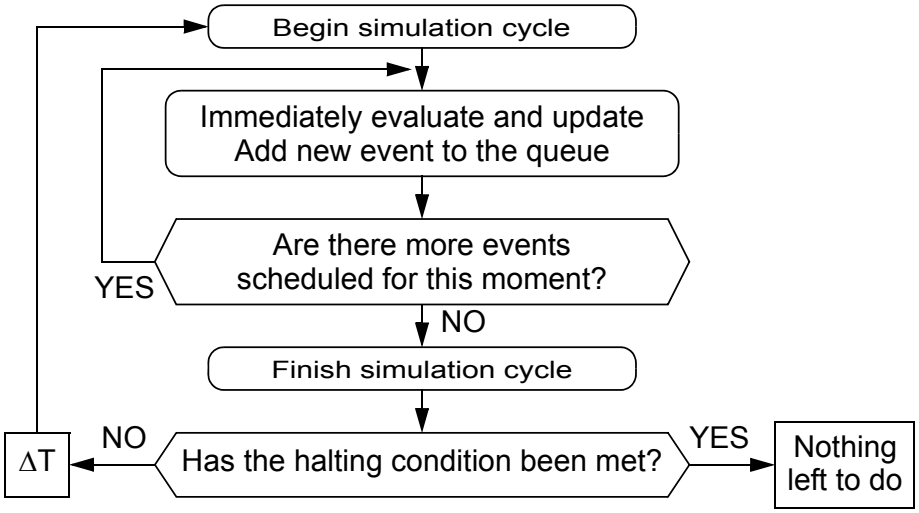
- **Functions**

```
function [7:0] func;
  input i1;
  integer i1;
  reg [7:0] rg;
  begin
    rg=i1+2;
    func=rg;
  end
endfunction
```

- **Access:** `x = func(n);`
- **A function may not have timing control construct - executed in zero simulation time**

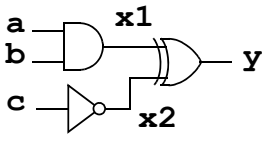
Time and Events

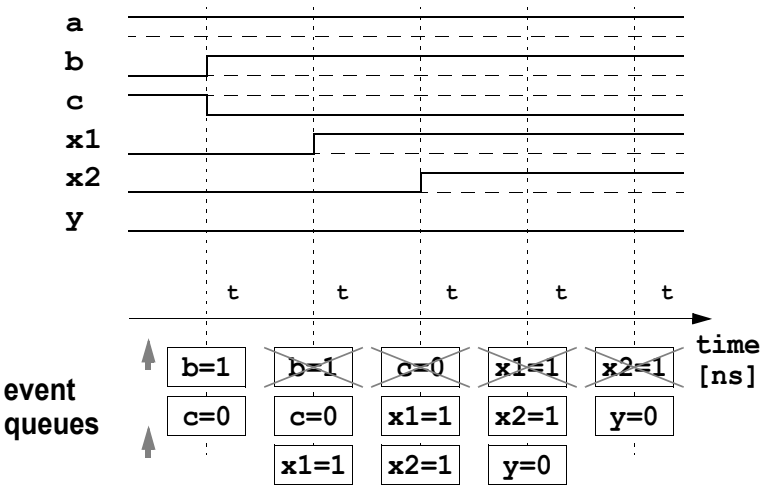
Zero-delay simulation model



Zero-delay simulation model (example #1)

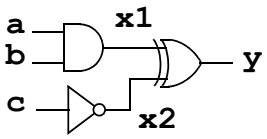
```
assign x1 = a & b;
assign x2 = ! c;
assign y  = x1 ^ x2;
```

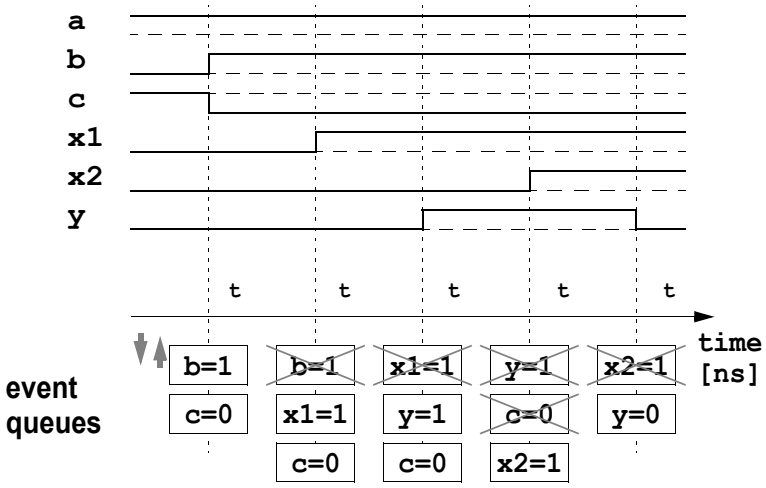




Zero-delay simulation model (example #2)

```
assign x1 = a & b;
assign x2 = ! c;
assign y  = x1 ^ x2;
```





Non-Deterministic Behavior

```
module stupidVerilogTricks (f,a,b);
  input  a, b;
  output f;
  reg    f, q;

  initial    f = 0;

  always @(posedge a)   #10 q = b;

  not ( qBar, q );

  always @q   f = qBar;

endmodule
```

```
q=0
f=qBar=b=1
a=0


a=1
#10 q=1    [b==1]

f==?

1) qBar=0    [q==1]
   f=0

2) f=1       [qBar==1]
   qBar=0    [q==1]
```

Timing Control

- Suspending execution for a fixed time period
- Suspending execution until an event occurs
- Suspending execution until an expression comes true
  - level sensitive event control

```
#expression
@event-expression
wait ( expression )
```

→

```
variable [ or variable ... ]
posedge one-bit-variable
negedge one-bit-variable
event-variable
```

```
@(posedge clk) #5 out=in;
```


Event Control

```
module event_control;
  event e1, e2;
  initial @e1 begin
    $display("I am in the middle.");
    ->e2;
  end
  initial @e2
    $display("I am the last one...");
  initial begin
    $display("I am the first!");
    ->e1;
  end
endmodule
```

```
I am the first!
I am in the middle.
I am the last one...
```

Timing Control Inside Assignments

```
state = #clk_period next_state;
```

≡

```
temp = next_state;
#clk_period state = temp;
```

```
state = @my_event next_state;
```

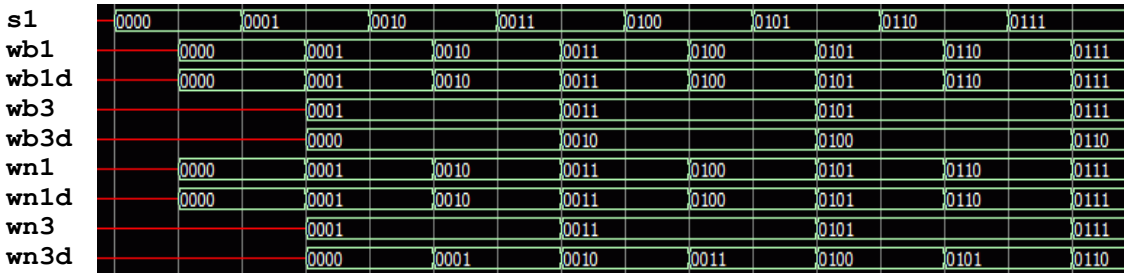
≡

```
temp = next_state;
@my_event state = temp;
```


Timing Control Inside Assignments (cont.)

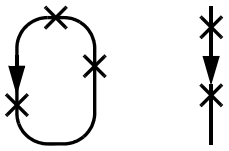
```
always @(s1) #1 wb1  =    s1;
always @(s1)    wb1d = #1 s1;
always @(s1) #3 wb3  =    s1;
always @(s1)    wb3d = #3 s1;
```

```
always @(s1) #1 wn1  <=    s1;
always @(s1)    wn1d <= #1 s1;
always @(s1) #3 wn3  <=    s1;
always @(s1)    wn3d <= #3 s1;
```

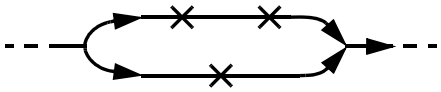


Parallelism

- **Structural parallelism**
  - **modules**
  - **continuous assignments (data-flow style)**
  - **behavioral instances (always & initial blocks)**

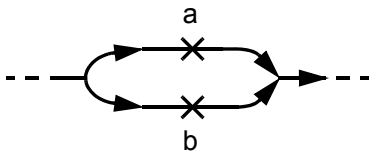


- **Behavioral parallelism**
  - **fork & join**
  - **disable**



fork & join

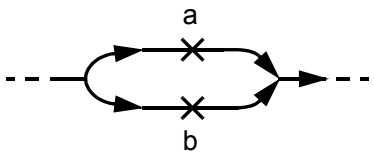
```
module fork_join;
  event a, b;
  initial begin
    // ...
    fork
      @a ;
      @b ;
    join
    // ...
  end
endmodule
```



- **continues when both events, *a* and *b*, occur**

fork & join + disable

```
module fork_join;
  event a, b; // Block name!
  initial begin
    // ...
    fork : block1
      @a disable block1;
      @b disable block1;
    join
    // ...
  end
endmodule
```



- continues when either *a* or *b* occurs

disable

```
begin : break
  for (i=0;i<1000;i=i+1) begin : continue
    if (a[i]==0) disable continue;  // i.e. continue
    if (b[i]==a[i]) disable break;  // i.e. break
    $display("a[",i,"]=",a[i]);
  end
end
```

- disable <block_name>
  - removes the rest of events associated with the block
  - named blocks and tasks only
- named blocks
  - local variables allowed

Structural and Behavioral Descriptions

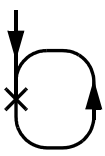
- Structural - created from lower level modules
- Data-flow - combinational logic
  - keyword `assign`
- Behavioral - algorithms etc.
  - keywords `initial` & `always`

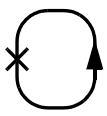
Behavioral

```
initial begin
  forever
  @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)   zero = 1;
    else            zero = 0;
  end
end
```

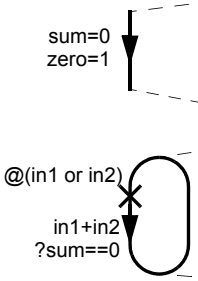
≡

```
always
  @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)   zero = 1;
    else            zero = 0;
  end
```



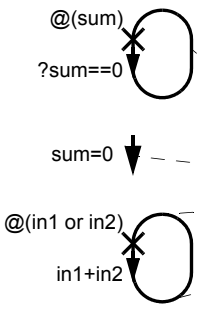


Behavioral (cont.)



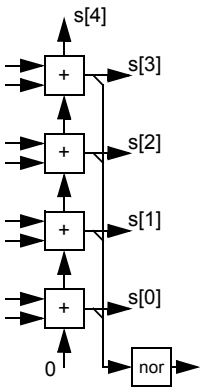

```
module adder4(in1,in2,sum,zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg [4:0] sum;
  reg zero;
  initial begin
    sum = 0;
    zero = 1;
  end
  always @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)  zero = 1;
    else           zero = 0;
  end
endmodule
```


Behavioral (cont.)




```
module adder4(in1,in2,sum,zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg [4:0] sum;
  assign zero = (sum==0) ? 1 : 0;
  initial sum = 0;
  always @(in1 or in2)
    sum = in1 + in2;
endmodule
```

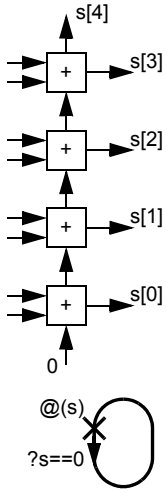
Structural




```
module adder4 (in1, in2, s, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] s;
  output zero;
  fulladd u1 (in1[0],in2[0], 0, s[0],c0);
  fulladd u2 (in1[1],in2[1],c0, s[1],c1);
  fulladd u3 (in1[2],in2[2],c1, s[2],c2);
  fulladd u4 (in1[3],in2[3],c2, s[3],s[4]);
  nor u5 (zero,s[0],s[1],s[2],s[3],s[4]);
endmodule
```

```
module fulladd (in1, in2, cin, sum, cout);
  input in1, in2, cin;
  output sum, cout;
  assign { cout, sum } = in1 + in2 + cin;
endmodule
```


Combined - Mixed Mode




```
module adder4 (in1, in2, s, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] s;
  output zero;  reg zero;
  fulladd u1 (in1[0],in2[0], 0, s[0],c0);
  fulladd u2 (in1[1],in2[1],c0, s[1],c1);
  fulladd u3 (in1[2],in2[2],c1, s[2],c2);
  fulladd u4 (in1[3],in2[3],c2, s[3],s[4]);
  always @(s)
    if (s == 0)  zero = 1;
    else         zero = 0;
endmodule
```

Advanced Topics

- Parameterized Modules
- Compiler Control
- Memory Images
- User Primitives
- More About Assignments
- More About Nets
- More About Gates
- Synthesizable Verilog

Parameterized Modules

```
module xorx (xout, xin1, xin2);
  parameter width = 4,
            delay = 10;
  output [1:width] xout;
  input  [1:width] xin1, xin2;

  assign #(delay)  xout = xin1 ^ xin2;
endmodule
```

```
// 8 bits, delay 10
xorx #(8) (vout,vin1,
    {b0,b1,b2,b3,b4,b5,b6,b7});
```

```
// 4 bits, delay 20
xorx #(4,20) (vout,vin1,
    {b0,b1,b2,b3});
```

Compiler Control

- ‘define <macro_label> <replacement>
- ‘ifdef <macro_label>
  // code...
  ‘endif
- `‘include *”verilog-file”*`

```
‘include ”design.def”
 ...
‘ifdef DEBUG_MODE     /* Debugging ...  */
  initial #1 begin
    $display(”\n Time:  Address   Data”);
  end
  always @(clk) begin
    $display(”%t:  %h  %h”,
        $time,address,data);
  end
‘endif
```


Compiler Control (cont.)

```
parameter  WORD_SIZE = 32;
‘define    WORD     [WORD_SIZE-1:0]
  // ...

reg ‘WORD address, data;
  // ...
```

```
// <time_unit>/<time_precision>
‘timescale 1 ns / 1 ns
```

```
‘timescale 10 ns / 0.1 ns
// ...
#7.748; // delay 77.5 ns
```

- ModelSim SE/PE/XE User’s Manual
- IEEE Std 1364-1995 compiler directives

Memory Images

- $readmemb
- $readmemh

```
  ...
reg [DSIZE-1:0] MEM [0:MAXWORDS-1];
  ...
  $readmemh("PROG.FILE",MEM);
  ...
```

```
@0000     // Hexadecimal address
// Code       _IC_._ _RD.__ __.S1_ _.S2__ .___IMM_____
00000000  // 0000.0 000.00 00.000 0.0000 .0000.0000.0000  add.f   -,0,0
00000000  // 0000.0 000.00 00.000 0.0000 .0000.0000.0000  add.c.f -,0,0
2040007f  // 0010.0 000.01 00.000 0.0000 .0000.0111.1111  add.t   R1,127,127
20820ffd  // 0010.0 000.10 00.001 0.0000 .1111.1111.1101  add.t   R2,R1,-3
60c40000  // 0110.0 000.11 00.010 0.0000 .0000.0000.0000  add.c.t R3,R2,0
20043000  // 0010.0 000.00 00.010 0.0011 .0000.0000.0000  add.t   -,R2,R3
```


User Primitives

```
primitive MUX_4_2 (Y,D0,D1,D2,D3,S1,S2);
  input D0,D1,D2,D3,S1,S2;
  output Y;
  table // D0 D1 D2 D3 S1 S2 : Y
    0 ? ? ? 0 0 : 0 ;
    1 ? ? ? 0 0 : 1 ;
    ? 0 ? ? 0 1 : 0 ;
    ? 1 ? ? 0 1 : 1 ;
    // ...
    ? ? ? 0 1 1 : 0 ;
    ? ? ? 1 1 1 : 1 ;
  endtable
endprimitive
```

User Primitives (cont.)

- one bit wide ports
- `wire` - combinational
- `reg` - sequential

```
0     logic 0
1     logic 1
x     unknown
?     either 0, 1 or x  (input ports only)
b     either 0 or 1     (input ports only)
-     no change (outputs of sequential primitives)
(xy)  value change x,y=0,1,x,? or b
*     any value change (same as (??))
r     rising edge on input (01)
f     falling edge on input (10)
p     positive edge ((01),(0x) or (x1))
n     negative edge ((10),(1x) or (x0))
```


More About Assignments

- Behavioral assignments
  - assign <assignment>
    - reg type only
  - deassign <lvalue>
    - undoes behavioral assignment
  - force <assignment>
    - reg & net types
    - stronger than assign
  - release <lvalue>
    - reg & net types
    - undoes force statement

```
<continous_assignment> ::=
    assign [<drive_strength>] [<delay2>] <list_of_net_assignments>;
```

More About Nets

```
<net_declaration> ::=
    <net_type> [scalared|vectored] [<strength>]
        [<range>] [<delay>] <variable_list>;

<net_type> ::=  wire | tri | wand | wor | triand | trior |
                tri0 | tri1 | supply0 | supply1 | trireg
```

- wire, tri - no logic function (only difference is in the name)
- wand, wor, triand, trior - wired logic (wand==triand, wor==trior)
- tri0, tri1 - connections with resistive pull
- supply0, supply1 - connections to a power supply
- trireg - charge storage on a net

```
scalared - single bits are accessible (default)
vectored - single bits are not accessible

<range> ::= [ <msb>:<lsb> ]
```


More About Nets - Delays

```
<delay> ::=  #<delay_value> | #(<delay_value>) | <delay2> | <delay3>

<delay2> ::=  #(<delay_value>,<delay_value>)
<delay3> ::=  #(<delay_value>,<delay_value>,<delay_value>)

<delay_value> ::=  <unsigned_number> | <parameter_identifier> |
                   <constant_mintypmax_expression>

<constant_mintypmax_expression> ::=
    <constant_expression>:<constant_expression>:<constant_expression>
```

- **Delays**

```
<delay>
<rise_delay> <fall_delay>
<rise_delay> <fall_delay> <turnoff_delay>
```

More About Nets - Strength

```
<strength> ::=  <charge_strength> | <drive_strength>

<charge_strength> ::= (small) | (medium) | (large)

<drive_strength> ::=  (<zero_strength>,<one_strength>) |
                      <one_strength>,<zero_strength>)

<zero_strength> ::=  supply0 | strong0 | pull0 | weak0 | highz0

<one_strength>  ::=  supply1 | strong1 | pull1 | weak1 | highz1
```


More About Gates

```
<gate_instantiation> ::=
    <gate_type> [<drive_strength>] [<delay>] [<label>] (<terminals>);

<gate_type> ::=  and | nand | or | nor | xor | xnor |
                 buf | not | bufif0 | bufif1 | notif0 | notif1 |
                 nmos | pmos | rnmos | rpmos |
                 tran | rtran | tranif0 | tranif1 | rtranif0 | rtranif1 |
                 cmos | rcmos | pullup | pulldown
```

- and, nand, or, nor, xor, xnor - simple logic gates (output, input1, input2[,...])
- buf, not - simple buffers (output, input)
- bufif0, bufif1, notif0, notif1 - three-state drivers (output, data-input, control-input)
- nmos, pmos, rnmos, rpmos - transistors (output, data-input, control-input)
- tran, rtran - true bidirectional transmission gates (inout1, inout2)
- tranif0, tranif1, rtranif0, rtranif1 - true bidirectional transmission gates (io1, io2, control-input)
- cmos, rcmos - transmission gates (data-output, data-input, n-channel-control, p-channel-control)
- pullup, pulldown - drive strengths (logic-1/logic-0) (output)
- r<type> - relatively higher impedance when conducting

More About Gates

```
<drive_strength> ::=  (<zero_strength>,<one_strength>) |
                      <one_strength>,<zero_strength>)

<zero_strength> ::=  supply0 | strong0 | pull0 | weak0 | highz0

<one_strength>  ::=  supply1 | strong1 | pull1 | weak1 | highz1

<delay> ::=  #<delay_value> | #(<delay_value>) | <delay2> | <delay3>
```


Verilog vs. VHDL ?

or VHDL vs. Verilog ?

Feature	Verilog	VHDL
Type declaration	weak	strong
User defined types	- (macros)	++
User defined operators	--	++
Archives / libraries	- (simulator)	library & use
Reusability	- (include)	++ (package)
Pre-compilation	+ (limited macros)	- (alias)
Flexibility of constructions	@ operation level (predefined gates)	@ data level (attributes)
Usability & Synthesizability	RTL & lower level(s)	RTL & higher levels
Standardization	+ (simulator, now IEEE)	++ (IEEE)
Programming language	C (K&R)	Ada (OO Pascal)

```
Date: Wed, 09 Aug 2000 21:15:18 GMT
From: eml@riverside-machines.com.NOSPAM
Newsgroups: comp.lang.verilog
Subject: Re: verilog vs VHDL

On Wed, 09 Aug 2000 11:07:20 -0700, Kevin Cameron x3251 <dkc@galaxy.nsc.com> wrote:

>Religon aside, there are some fundamental differences between Verilog & VHDL:
>
>1. Bidirectional devices
>   Verilog supports bidirectional signal flow in primitives, VHDL does
>   not support it at all.
>   => You can't do transistor-level (CMOS) stuff easily in VHDL.
>
>2. Signal Resolution
>   Resolution in Verilog is flat, and in VHDL is hierarchical. VHDL
>   additionally allows port-bound signal conversion.
>   => Signals in VHDL do not (in general) behave the same as physical
>   wires.
>
>3. User defined types
>   Not supported in Verlog.
>
>My opinion is that VHDL is Good/OK as an abstract simulation language (which
>you can use with Synthesis etc.) but near useless as a hardware (implementation)
>verification tool. So if you need to do the latter, you may want to stick with
>Verilog throught your design flow.
>
>Items 1 & 2 above become more of a problem with VHDL-AMS.
>
>Kev.

There are some fundamental differences, and some surprising
similarities. If you don't mind me saying so, your list above doesn't
do justice to either. Point (1) is relevant only to a (very) small
number of users, and point (2), as far as I can see, has no practical
significance. I don't want to get involved in a VHDL/Verilog flame,
and I want to do this in 10 minutes, rather than several hours, but
you should also consider, among many other things:

1) Scheduling is fundamentally different. The practical effect of
this is that Verilog sims should be faster, but can suffer from
non-determinism.

2) Structural hierarchy is a lot more flexible in VHDL, via the
use of entity/architecture pairs, and configurations. It's easier to
manage designs using packages and libraries.

3) There is no fundamental difference in what can be synthesised
between the 2 languages, contrary to frequent statements here. Many of
the useful extra syntax features of VHDL (see (11)) are also
synthesisable. Both VHDL and Verilog have synthesis subsets.
```



```
4) VHDL is much more strongly typed than Verilog; scope is
rigourously defined; you have to declare just about everything before
you use it; and so on. Some argue that this is a bad thing; others
that it's essential. VHDL's types are also extensible, through
enumerated types. This is both useful and synthesisable.

5) VHDL is more verbose than Verilog. Sometimes there's a good
reason for this (see (2) and (4) above); sometimes there isn't.
Whether this is a significant problem is debatable.

6) VHDL's and Verilog's antecedents are Ada and C, respectively.
This makes some of Verilog familiar to most engineers. However,
contrary to popular belief, there is very little direct similarity
between Verilog and C.

7) VHDL doesn't have system tasks or compiler directives. The
system tasks have to be implemented through explicit coding, use of
libraries, or through a PLI. Compiler directives require an external
preprocessor.

8) VHDL doesn't have a standard PLI. Sim vendors have their own PLIs.

9) VHDL doesn't make a distinction between nets and registers; it
has signals instead. The confusion between blocking  and non-blocking
assignments is avoided for a number of reasons, including tighter
scoping, and more rigorous scheduling.

10) It's easier to parameterise VHDL designs.

11) VHDL has a richer syntax; examples that come to mind are
generates, conditional and selected signal assignments,
multi-dimensional arrays, more usable functions and procedures/tasks,
easier initialisation, ANSI-C style declarations, attributes,
enumerated types. These are all synthesisable.

12) VHDL has nothing equivalent to a fork-join, or block disabling.

13) Some ASIC vendors may not let you use VHDL. This is a
significant problem for VHDL, but can generally be got around, if only
by generating Verilog gate-level netlists.

14) Verilog 2000 will fix some of the more obvious syntax omissions. There is also
an updated VHDL on the way, but it won't be soon.

15) You should be able to produce a simple design quicker if
you're learning Verilog. However, the total area under the learning
curve is much the same for both languages.

16) In short, VHDL is much more rigorous as a computer language,
is better defined, and provides a more complete single-language
simulation and synthesis solution. The price you pay is more
verbosity, a steeper learning curve, possibly slower simulation, and
possible marginalisation in some parts of the world. You're more
likely to get a job in Europe with VHDL, and in the US with Verilog.

Evan
```