UMBC

AN HONORS UNIVERSITY IN MARYLAND

Department of Computer Science and Electrical Engineering

Intro to Verilog II

http://6004.csail.mit.edu/6.371/handouts/L0{2,3,4}.pdf

http://www.asic-world.com/verilog/

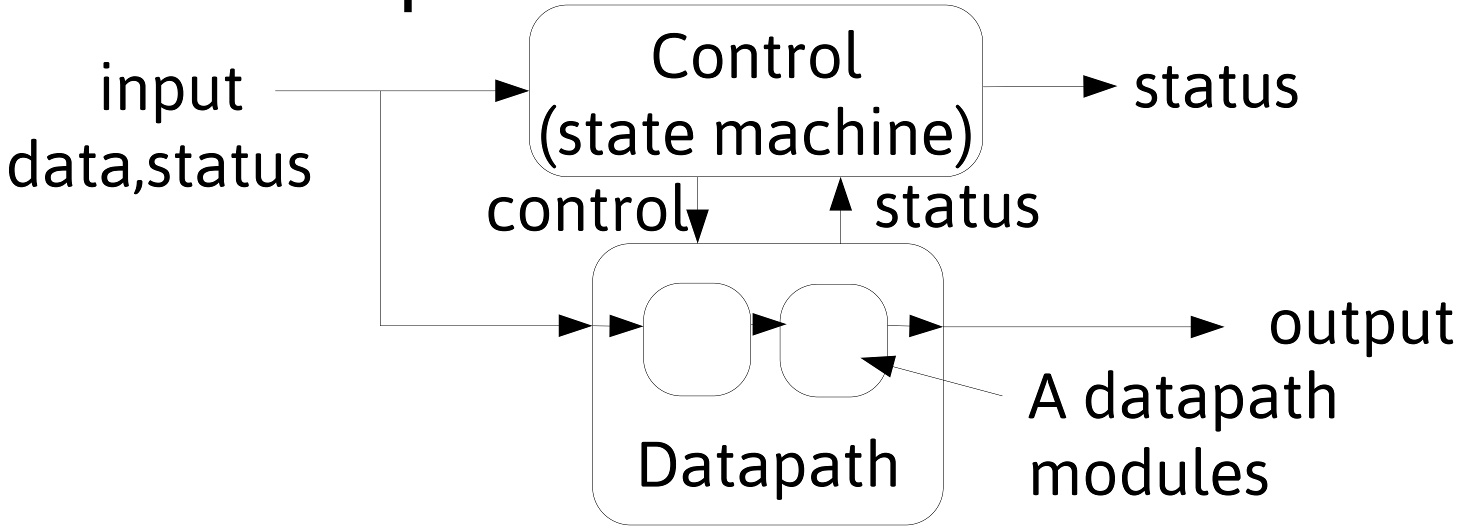
http://www.verilogtutorial.info/

Design Strategies

- For a beginner, treat Verilog as Hardware Description Language, not a software coding language. Start off learning Verilog by describing hardware for which you can design and draw a schematic; then translate this to HDL.
- Plan by partitioning the design into sections and modules and coding styles that should be used.
- Identify existing modules, memory components needed, and data-path logic, as well as the control signals required to make those operate as desired.
- Simulate each part with a testbench before putting system together. Update testbenches and resimulate them as your design evolves.

- Large memory blocks are often provided by the manufacturer to be instantiated. Smaller memory elements may be coded or embedded into other descriptions of the design
- Data-path logic can be embedded coded with data-flow, structural elements, or complex synthesizable behavioral descriptions.
- Some styles explicitly separate Comb. Logic and Seq Logic, but this is up to you.
- **Best practice is to develop a consistent approach to design, as well as a consistent coding style. It makes designing, coding, and debugging easier for you with time. An inconsistent hack-it-togheter and hack-until-it-works approach is not conducive to becoming more efficient.**

- Typically, complex control is implemented by a synthesizable behavioral case-statement-based state-machine, while simpler control could be implemented with any combinatorial description style. Data-path logic (comb. and sequential) can be integrated into the overall state machine or separated out (better for incremental simulation).
- Possible Blueprint:



Components of a modeling/description language

- Wires and registers with specified precision and sign
- Arithmetic and bitwise operations
- Comparison Operations
- Bitvector Operations (selecting multiple bits from a vector, concatenation)
- Logical Operators
- Selection (muxes)
- Indexed Storage (arrays)
- Organizational syntax elements and Precedence
- Modules (Hardware) Definition and Instantiation

Modules and Ports

keyword **module** begins a module

port list: ports must be declared to be **input**, **output**, or **inout**

```
`timescale 1ns / 1ps
//create a NAND gate out of an AND and an Inverter
module some_logic_component (c, a, b);
  // declare port signals
  output c;
  input a, b;

  // declare internal wire
  wire d;

  //instantiate structural logic gates
  and a1(d, a, b); //d is output, a and b are inputs
  not n1(c, d);    //c is output, d is input
endmodule
```

Additional internal nodes may be declared using the **wire** or **reg** word

keyword **endmodule** begins a module

nodes can be connected to nested modules or primitives or interact with procedural code

Verilog 2001: New Port Decl. Options

```
// Verilog 95 code
module memory( read, write, data_in, addr, data_out);
input   read;
input   write;
input   [7:0] data_in;
input   [3:0] addr;
output   [7:0] data_out;

reg [7:0] data_out;
```

After the port list, port direction must be declared to be **input**, **output**, or **inout** as well as the width if > 1
Type declaration: type default wire unless another type is provided

```
// Verilog 2k with direction
// and data type listed
module memory(
  input wire read,
  input wire write,
  input wire [7:0] data_in,
  input wire [3:0] addr,
  output reg [7:0] data_out
);
```

```
// Verilog 2k with no type
// in port list
module memory(
  input   read,
  input   write,
  input   [7:0] data_in,
  input   [3:0] addr,
  output   [7:0] data_out
);
```

all types declared to be wire!

Sensitivity List

- With Verilog 2001:
  - Comma separated sensitivity list
    - `always @ (posedge clk, posedge reset)`
    - `always @ (a, b, c, d, e)`
  - Shortcut for including all dependencies in a combinatorial block:
    - `always @ (*)`

A Testbench Module

```
//test the NAND gate
module test_bench; //module with no ports

  reg A, B;
  wire C;
  //instantiate your circuit
  some_logic_component S1(C, A, B);

  //Behavioral code block generates stimulus to test circuit
  initial
    begin
      A = 1'b0; B = 1'b0;
      #50
      $display("A = %b, B = %b, Nand output C = %b \n", A, B, C);
      A = 1'b0; B = 1'b1;
      #50
      $display("A = %b, B = %b, Nand output C = %b \n", A, B, C);
      A = 1'b1; B = 1'b0;
      #50
      $display("A = %b, B = %b, Nand output C = %b \n", A, B, C);
      A = 1'b1;
      B = 1'b1;
      #50
      $display("A = %b, B = %b, Nand output C = %b \n", A, B, C);
    end
endmodule
```

typically no ports

strictly internal nodes

includes an instance of the module under test

procedural code drives the stimulus to test the module under test

Delay statements separate on lines for now

Simple Testbench with Clock

```
module mydevice_tb();
  reg clk, rst;
  reg x1, x2;
  wire y1, y2;


mydevice DUT(clk,rst, y1,y2, x1,x2);

initial clk = 0;
always begin
   #50; //delay
   clk = ~clk;
end

initial begin
  #1000
  $finish;
end
```

Outputs from the module under test are simply structural connections at this level so wires are used

many signals will be reg since they are driven by procedural code

An instance of the device under test

An initial block can be used to set initial values of signals

A always block with delays can be used to drive cyclic signals

Stops simulation at T=1000

```
initial begin
  rst = 1;
  #10; //delay
  rst = 0;
end
```

Initial value and a change at T=10

```
initial begin
   y1=0;
   y2=0;

   #50; //delay
   y1=1;
   #50; //delay
   y1=0;
   y2=1;
   #50; //delay
   y1=1;
   y2=0;
end
```

Intialize signals immediately if not otherwise initialized, then add delays and assignments

We'll see other examples later, but at first avoid changing signals input to clocked blocks at the same time as the clock edge it is sensitive to

```
endmodule //end testbench module
```

This testbench includes no output statements, so it is assumed that a results waveform viewer (GUI) is used

Number Syntax

- Numbers in Verilog are commonly provide use a format with '
- The size is always specified as a decimal number. If no size is specified then the default size is at least 32bits and may be larger depending on the machine. Valid base formats are 'b , 'B , 'h , 'H 'd , 'D , 'o , 'O for binary, hexadecimal, decimal, and octal. Numbers consist of strings of digits (0-9, A-F, a-f, x, X, z, Z). The X's mean unknown, and the Z's mean high impedance If no base format is specified the number is assumed to be a decimal number. Some examples of valid numbers are:

```
  2'b10 // 2 bit binary-specified number
   'b10 // at least a 32-bit binary number
      3 // at least a 32-bit decimal number
  8'hAf // 8-bit hexadecimal-specified
-16'd47 // negative decimal-specified number
```

Logical Primitives

- Here is a list of logic primitives defined for Verilog:

Gate	Parameter List	Examples

nand
nor
and or
xor
xnor
	scalable, requires at least 2 inputs(output, input1, input2,... inputx)	and a1(C,A,B);
		nand na1(out1,in1,in2,in3,in4);

notbuf	(output, input)	not inv1(c,a);
notif0b ufif0	control signal active low(output, input, control)	notif0 inv2(c,a, control);
notif1b ufif1	control signal active high(output, input, control)	not inv1(c,a, control);

Continuous Assignment

- If you have a lot of various logic, the gate primitives of the previous section are tedious to use because all the internal wires must be declared and hooked up correctly. Sometimes it is easier to just describe a circuit using a single Boolean equation. In Verilog, Boolean equations which have similar timing properties (and synthesis results) as the gate primitives are defined using a continuous assignment statement using the **=** operator.

```
wire d;
and a1(d, a, b);
not n1(c, d);
```

can be replaced with one statement:

```
assign c = !(a && b);
```

notice that wire d was not required here

Implicit Assignment

- Assignments can also be made during the declaration of a wire. In this case the assign keyword is implicitly assumed to be there for example:

```
wire d;
assign d = a || b; //continuous assignment

wire d = a || b;   //implicit continuous assignment
```

Behavioral Design with Initial and Always blocks

- Behavioral code is used to describe circuits at a more abstract level then the structural level statements we have studied. A module can contain several initial and always procedural blocks. These behavioral blocks contain statements that control simulation time, data flow statements (like if-then and case statements), and blocking and non-blocking assignment statements.
- An **initial** block executes once at the beginning of a simulation.
- An **always** block continuously repeats its execution during a simulation
  - Its execution may be conditional if a <u>sensitivity list</u> is provided
    - If signals are directly provided, one or multiple changes to those signals at a given point in time allow the block to be evaluated once.
      If **posedge** or **negedge** are provided then the type of change (~ edge type) that triggers evaluation is restricted
- <u>Assuming no delay statements are included in the procedural code:</u> the keywords **begin** and **end** may be used to encapsulate a description of an algorithm using a block of “sequential code”….the code is just a description of a desired behavior and not necessarily the implementation itself – the entire description is evaluated in one instant in time (takes 0 time to complete)
  syntax-wise, use **begin** and **end** like **{** and **}** in C

Structural Data Types: **wire** and **reg** and the others

- Verilog data types called nets which model hardware connections between circuit components. The two most common structural data types are wire and reg.
- A **wire** is like a real wire in a circuit . Its purpose is to make circuit network connections. Its value at every instant in time is decided by the driver connected to it. The driver may be assigned through a structural connection to a primitive or module or a continuous assignment statement.
- Module ports of type **input** and **inout** are always of type wire. This type decision is ignorant of the external connection driving the signal.
- Module ports of type **output** may be **wire** (network connection) or **reg** (a variable)**,** depending on the coded driver. If driver is described using procedural code then use type **reg**.
- In procedural code, the reg type hold their values until another value is put on them.
- The declarations for wire and reg signals are inside a module but outside any initial or always block.
- The default state of a **reg** is 'x' (unknown), and the for a **wire** is 'z'.
- If you need a special strength type operation use special net keyword **wand**, **wor**, **tri**, **triand**, **trior**, **trireg**.

Undeclared Nets

- In Verilog 1995, default data type is net and its width is always 1 bit.
- This can be dangerous for two reasons…
  - a simple typing mistake can declare a new variable instead of an intended connection to an existing net causing a confusing error message or lead to a coding mistake
  - forgetting a declaration can lead to 1-bit wires which loose information

```
wire a,b,c,d,y;
mylib_and2(w1,a,b);
mylib_and2(w2,c,d);
mylib_and2(y,w1,w2);
```

```
wire [7:0] a; wire [7:0] b; wire [7:0] d;
wire [7:0] e;
c=a+b; //one bit!!!!
e=c+d;
```

- In Verilog 2001 the width is adjusted automatically
- In Verilog 2001, we can disable default data type by using a directive at the top of the code:

```
`default_net_type none
```

Verilog 2001: signed reg type, reg init., new operators

- Register data type is now called a variable, as the previous name of register created a lot of confusion for beginners. Also it is possible to specify an initial value for the register/variable data type.

```
reg a = 0; // v2k allows to init variables
reg b, c, d = 0; //just init d
```

- New signed reg.

```
// reg data type can be signed in v2k
// We can assign with signed constants
reg signed [7:0] data = 8'shF0;
```

Behavioral Design with blocking and non-blocking assignment statements

- There are 2 kinds of assignment statements:
  - blocking using the **=** operator, and
  - non-blocking using the **<=** operator.
- Blocking assignments act like sequential code statements and make an assignment when they are encountered
- Non-blocking schedule assignments to happen at some time in the future. They are called non-blocking because statements the follow can be evaluated before the actual assignment happens.
- Here are some examples:

Beginner Tips for Procedural Code

http://www.sunburst-design.com/papers/

1. When modeling sequential logic, use non-blocking assignments.
   registerA <= b+c;

2. When modeling latches, use non-blocking assignments. **(actually don't code any latches for now. If you see any synthesis message for latches, eliminate them.)**

3. When modeling combinatorial logic with an always block, use blocking assignments. a=b+c;

4. Separate combinatorial and sequential logic into separate always blocks **(as much as reasonably possible)** to avoid accidental registers and latches.

5. When modeling both sequential and combinatorial logic within the same always block, use non-blocking assignments for registers and minimally use blocking statements for intermediate combinatorial logic.

6. Do not mix blocking and non-blocking assignments to the same variable in the same always block.

7. Do not make assignments to the same variable from more than one always block.

Guideline: Use non-blocking for EVERY register

```
module dffb (q, d, clk, rst);
   output q;
   input d, clk, rst;
   reg q;
   always @(posedge clk)
      if (rst) q = 1'b0;
      else q = d;
endmodule
```

It is better to develop the habit of coding all sequential always blocks, even simple single-block modules, using nonblocking assignments.

```
module dffx (q, d, clk, rst);
   output q;
   input d, clk, rst;
   reg q;
   always @(posedge clk)
      if (rst) q <= 1'b0;
      else q <= d;
endmodule
```

Combinatorial and Registered-Output Logic

Combinatorial:

```
reg y;
always @(a,b)
   y = a & b;
```

could also have used y<= a & b; but we will follow a convention explained later to use blocking for all combinatorial logic

Sequential (registered-output combinatorial logic):

```
reg q;
always @(posedge clk)
   q <= a & b;
```

3 common organizations for sequential and combinatorial logic behavioral coding

- Separate always blocks for combinatorial and sequential logic
    - Comb. assignments use blocking statements
    - Seq. assignments use non-blocking statements

- Sequential and combinatorial logic in same block with combinatorial logic embedded in sequential assignments
    - Seq. assignments use non-blocking statements

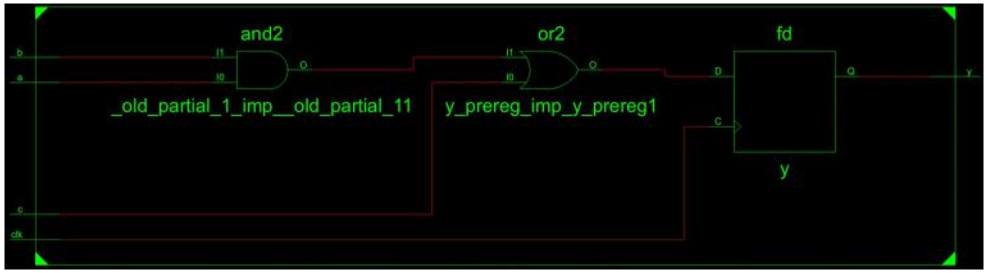
- Sequential and combinatorial logic in same block with both combinatorial and sequential assignments
    - Comb. assignments use blocking statements
    - Seq. assignments use non-blocking statements

AND-OR Examples

Combinatorial and Sequential Separated:

```
reg y,y_prereg,partial;
always @(a,b,c) begin
   partial = a & b;
   y_prereg = c | partial;
end

always @(posedge clk) begin
   y <= y_prereg;
end
```



Implicit Mix of Seq. and Comb. Logic:

```
reg y,partial;
always @(posedge clk) begin
   partial = a & b;
   y <= c | partial;
end
```

Explicit Mix of Seq. and Comb. Logic:

```
reg y,y_prereg,partial;
always @(posedge clk) begin
   partial = a & b;
   y_prereg = a | partial;
   y <= y_prereg;
end
```

AND-OR Examples with async active low clr signal

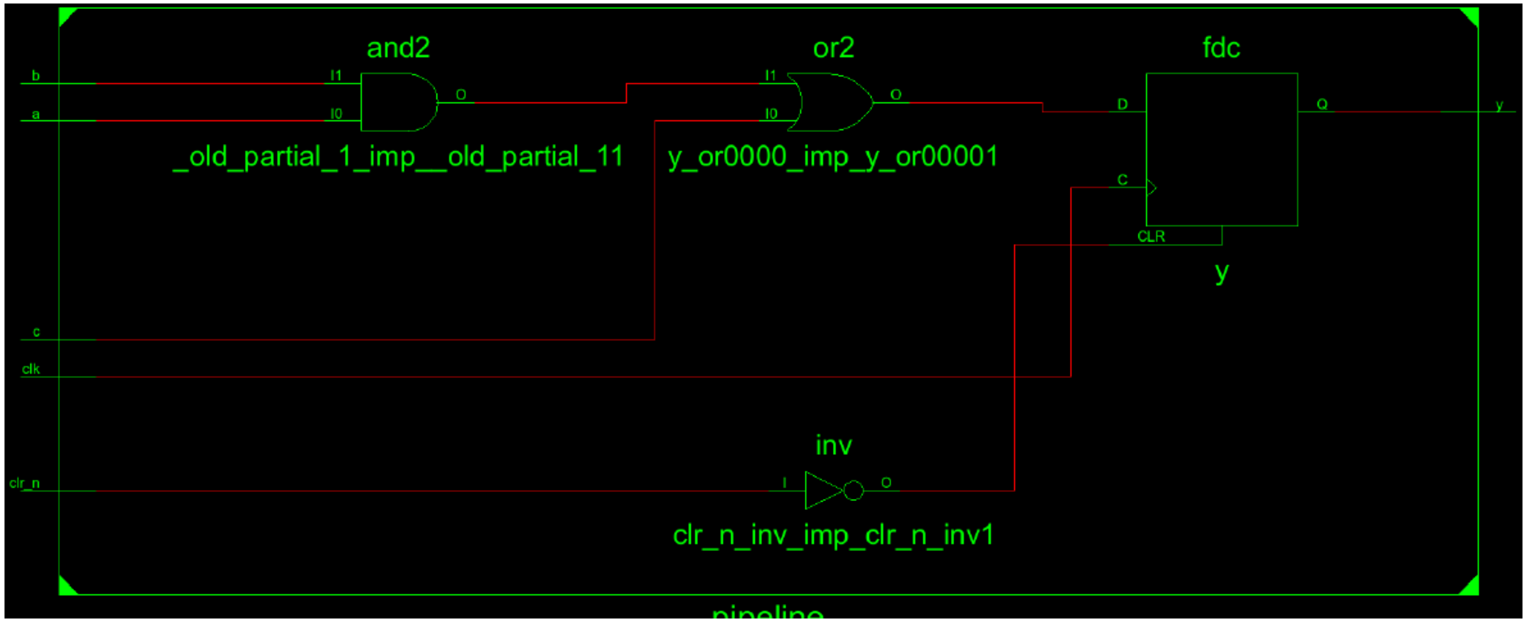
<u>Combinatorial and Sequential Logic Separated</u>:

```
reg y,y_prereg,partial;
always @(a,b,c) begin
   partial = a & b;
   y_prereg = a | partial;
end

always @(posedge clk, negedge clr_n) begin
  if (!clr_n) y <= 1'b0;
  else y <= y_prereg;

end
```

async control signals must appear in the sensitivity list



Both of these generate an error in Xilinx ISE ERROR:Xst:899 - "top.v" line 28: The logic for <partial> does not match a known FF or Latch template. The description style you are using to describe a register or latch is not supported in the current software release.

```
reg y,partial;
always @(posedge clk, negedge clr_n)
begin
  partial = a & b;
  if (!clr_n) y <= 1'b0;
  else y <= a | partial;
end
```

```
reg partial,y_prereg;

always @(posedge clk, negedge clr_n)
begin
  if (!clr_n) begin
    partial = a & b;
    y_prereg = c | partial;
    y <= 1'b0;
  end
  else begin
    partial = a & b;
    y_prereg = c | partial;
    y <= y_prereg;
  end
end
endmodule
```

The following code is more compact than the initial separated version, but leads to warnings

```
reg y,y_prereg,partial;
always @(posedge clk, negedge clr_n)
begin
  if (!clr_n) y <= 1'b0;
  else begin
    partial = a & b;
    y_prereg = a | partial;
    y <= y_prereg;
  end
end
```

implied registers and latches are trimmed since they are only used inside this procedural code block and feed into another signal

If the are used outside, additional sequential logic would be generate to provide the saved values externally

```
WARNING:Xst:646 - Signal <y_prereg> is assigned but
never used. This unconnected signal will be trimmed
during the optimization process.
WARNING:Xst:646 - Signal <partial> is assigned but
never used. This unconnected signal will be trimmed
during the optimization process.
```

Constants

- Avoid magic numbers and use parameters
  - parameter a=31; //int
  - parameter a=32,b=31; //ints
  - parameter byte_size=8, byte_max=bytesize-1; //int
  - parameter a =6.22; //real
  - parameter delay = (min_delay + max_delay) /2 //real
  - parameter initial_state = 8'b1001_0110; //reg

Arrays, Vectors, and Memories

- Verilog supports three similar data structures called Arrays, Vectors, and Memories. Arrays are used to hold several objects of the same type. Vectors are used to represent multi-bit busses. And Memories are arrays of vectors which are accessed similar to hardware memories. Read the following examples to determine how to reference and use the different data structures.

..arrays

```
// Arrays for integer, time, reg, and vectors of reg

integer i[3:0];   //integer array with a length of 4
time    x[20:1]; //time array with length of 19
reg     r[7:0];   //scalar reg array with length of 8

c = r[3]; //the 3rd reg value in array r is assigned to c
```

…vectors

```
//*** Vectors are multi-bit words of type reg or net


reg  [7:0] MultiBitWord1;    // 8-bit reg vector with MSB=7 LSB=0

wire [0:7] MultiBitWord2;    // 8-bit wire vector with MSB=0 LSB=7

reg [3:0] bitslice;

reg a;                  // single bit vector often referred to as a scalar

....     //referencing vectors

a = MultiBitWord1[3]; //applies the 3rd bit of MultiBitWord1 to a

bitslice = MultiBitWord1[3:0]; // applies the 3-0 bits of
                               // MultiBitWord1 to bitslice
```

Memories

```
//*** Memories are arrays of vector reg ********************************
reg [7:0] ram[0:4095];      // 4096 memory cells that are 8 bits wide


//code excerpt from Chapter 2 SRAM model
input [11:0] ABUS;  // 12-bit address bus to access all 4096 memory cells
inout [7:0] DATABUS;// 8-bit data bus to write into and out of a memory cell
reg  [7:0] DATABUS_driver;
wire [7:0] DATABUS = DATABUS_driver; //inout must be driven by a wire
....
for (i=0; i < 4095; i = i + 1)  // Setting individual memory cells to 0
  ram[i] = 0;
end
....
ram[ABUS] = DATABUS;  //writing to a memory cell
....
DATABUS_driver =  ram[ABUS]; //reading from a memory cell
```

Operators

- Here is a small selection of the Verilog Operators which look similar but have different effects.
- **Logical Operators** evaluate to TRUE or FALSE.
- **Bitwise operators** act on each bit of the operands to produce a multi-bit result.
- **Unary Reduction** operators perform the operation on all bits of the operand to produce a single bit result.
- See also http://www.asic-world.com/verilog/operators1.html
- http://www.asic-world.com/verilog/operators2.html

Operator	Name	Examples
!	logical negation	
~	bitwise negation	
&&	logical and	
&	bitwise and	abus = bbus&cbus;
&	reduction and	abit = &bbus;
~&	reduction nand	
||	logical or	

Operator	Name	Examples
|	bitwise or	c=a|b;
|	reduction or	c = |b;
~|	reduction nor	c = ~|b;
^	bitwise xor	c = ^b;
^	reduction xor	c = ^b;
~^ ^~	bitwise xnor	c = a~^b;
~^ ^~	reduction xnor	c = ~^b;

Operator	Name, Description	Examples
==	logical equality, result may be unknown if x or z in the input	if (a == b)
===	logical equality including x and z	
!=	logical inequality, result may be unknown if x or z in the input	
!==	logical inequality including x and z	
>	relational greater than	
>>,<<	shift right or left by a number of positions	a = shiftvalue >> 2;
>=	relational greater than or equal	
<	relational less than	
<<<,>>>	Signed shifts, shift right or left by a number of positions with a signed left argument	
<=	relational less than or equal	if (a <= b)

+,-,*,/	Arithmetic Operators
	Note: synthesizers may only support divide by constant power of two
		c=a+b;
		c=b/4; //right shift by 2

**	power	c=a**b

Operator	Name, description	Examples
<=	non blocking assignment statement, schedules assignment for future and allows next statement to execute	b <= b + 2;
=	blocking assignment statement, waits until assignment time before allowing next statement to execute	a = a + 2;

Operator	Name	Examples

{ , }	Concatenation:
	concatenation of one, two, or more operands
		{4'b1111,4'b0000}
		{2'b11,2'b11,2'b00,2'b00}
		
		Both produce
		8'b11110000

{n{x}}	Replication: Allows **fixed** number of replications (n must be a constant)	assume **a=16'hFFFF;** then **2{a}** produces **32'hFFFFFFFF** **{16{1'b0},a}** produces **32'h0000FFFF** **8{2'b10}** Produces **16'b1010101010101010**

Some additional Behavioral Data Types: integer, real, and time

- The types in integer and real are convenient data types to use for counting in behavioral code blocks. These data types act like their counter parts in other programming languages. If you eventually plan to synthesize your behavioral code then you would probably want to avoid using these data types because they often synthesize large circuits.

- The data type time can hold a special simulator value called simulation time which is extracted from the system function $time. The time information can be used to help you debug your simulations.

```
integer i, y;
real a;
real b = 3.5;
real c = 4;
time simulationTime;
initial begin
  y = 4;
  i = 5 + y;
  c = c + 3.5;
  a = 5.3e4;
  simulationTime = $time;
  $display("integer y = %d, i = %f \n", y, i);
  $display("reals   c = %f, a = %e, b= %g \n", c, a, b);
  $display("time    simulationTime = %t \n", simulationTime);
end
```

Verilog Design Flow

- Create RTL Design Models and Behavioral Test Bench Code
- Functionally Simulate your Register-Transfer-Level Design
- Convert RTL-level files to a Gate-level model with a Synthesizer
- Perform Gate Level simulations with FPGA or ASIC libraries
- Optional Step: Gate-level simulation with SDF timing information (with results from place and route)