Verilog

-brief-

Verilog is a hardware description language (HDL) used to model electronic systems. The language (sometimes called *Verilog HDL*) supports the design, verification, and implementation of analog, digital, and mixed-signal circuits at various levels of abstraction.

The designers of Verilog wanted a language with syntax similar to the C programming language so that it would be familiar to engineers and readily accepted. The language has a pre-processor like C, and the major control keywords such as "if", "while", etc are similar. The formatting mechanism in the printing routines and language operators (and their precedence) are also similar.

The language differs in some fundamental ways. Verilog uses Begin/End instead of curly braces to define a block of code. The definition of constants in verilog require a bit width along with their base, consequently these differ. Verilog 95 and 2001 don't have structures, pointers, or recursive subroutines either. (However, System Verilog now includes these capibilities) Finally, the concept of time —so important to a HDL— won't be found in C.

The language differs from a conventional programming language in that the execution of statements is not strictly linear. A Verilog design consists of a hierarchy of modules. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behaviour of the module by defining the relationships between the ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all concurrent statements and all begin/end blocks in the design are executed in parallel. A module can also contain one or more instances of another module to define sub-behavior.

A subset of statements in the language is synthesizable. If the modules in a design contain only synthesizable statements, software can be used to transform or synthesize the design into a netlist that describes the basic components and connections to be implemented in hardware. The netlist may then be transformed into, for example, a form describing the standard cells of an integrated circuit (e.g. an ASIC) or a bitstream for a programmable logic device (e.g. a FPGA).

Contents

History

Beginning

Verilog was invented by Phil Moorby at Automated Integrated Design Systems (later renamed to Gateway Design Automation) in 1985 as a hardware modeling language. Gateway Design Automation was later purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL simulator logic simulators.

Verilog-95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog 2001

Verilog 2005

Verilog 2005, IEEE Standard 1364-2005, focus mostly on minor corrections, as any language improvement was done as a separate project, known as SystemVerilog.

The latest versions of the language include support for analog and mixed signal modelling. These are referred to as Verilog-AMS.

SystemVerilog

The advent of High Level Verification languages such as OpenVera, and Verisity's E language encouraged the development of Superlog by Co-Design Automation Inc. Co-Design Automation Inc was later purchased by Synopsys. The foundations of Superlog and Vera have been donated to Accellera. It has been transformed and updated to SystemVerilog which became the IEEE standard P1800-2005. SystemVerilog is fully alligned with Verilog-2005.

Example

The easiest Hello world example looks like this:

```
module main;
    initial
    begin
        $display("Hello world!");
        $finish;
    end
endmodule
```

A simple example of two flip-flops follows:

```
module toplevel(clock,reset);
 input clock;
 input reset;

 reg flop1;
 reg flop2;

 always @ (posedge reset or posedge clock)
 if (reset)
   begin
     flop1 <= 0;
     flop2 <= 1;
```

```
   end
 else
   begin
     flop1 <= flop2;
     flop2 <= flop1;
   end
endmodule
```

The "<=" operator in verilog is another aspect of its being a hardware description language as opposed to a normal procedural language. This is known as a "non-blocking" assignment. When the simulation runs, all of the signals assigned with a "<=" operator have their assignment scheduled to occur after all statements occurring during the same point in time have executed. After all the statements have been executed for one event, the scheduled assignments are performed. This makes it easier to code behaviours that happen simultaneously.

In the above example, flop1 is assigned flop2, and flop2 is assigned flop1. These statements are executed during the same time event. Since the assignments are coded with the "<=" non-blocking operator, the assignments are scheduled to occur at the end of the event. Until then, all reads to flop1 and flop2 will use the values they had at the beginning of the time event.

This means that the order of the assignments are irrelevant and will produce the same result. flop1 and flop2 will swap values every clock.

The other choice for assignment is an "=" operator and this is known as a blocking assignment. When the "=" operator is used, things occur in the sequence they occur much like a procedural language.

In the above example, if the statements had used the "=" blocking operator instead, then the order of the statements would affect the behaviour. If the same code were used but changed to "=" operators, the reset would set flop2 to a 1, and flop1 to a 0. A clock event would set flop1 to flop2 (a 1) and this assignment would happen immediately. The next statement would assign flop2 to flop1, which is now a 1. Rather than swap values every clock, flop1 and flop2 would both become 1 and remain that way.

An example counter circuit follows:

```
module Div20x (rst, clk, cet, cep, count,tc);
// TITLE 'Divide-by-20 Counter with enables'
// enable CEP is a clock enable only
// enable CET is a clock enable and
// enables the TC output
// a counter using the Verilog language

parameter size = 5;
parameter length = 20;

input rst; // These inputs/outputs represent
input clk; // connections to the module.
```

```
input cet;
input cep;

output [size-1:0] count;
output tc;

reg [size-1:0] count; // Signals assigned
                      // within an always
                      // (or initial)block
                      // must be of type reg

wire tc; // Other signals are of type wire

// The always statement below is a parallel
// execution statement that
// executes any time the signals
// rst or clk transition from low to high

always @ (posedge clk or posedge rst)
  if (rst) // This causes reset of the cntr
    count <= 5'b0;
  else
  if (cet && cep) // Enables both  true
    begin
      if (count == length-1)
        count <= 5'b0;
      else
        count <= count + 5'b1; // 5'b1 is 5 bits
    end                        // wide and equal
                               // to the value 1.

// the value of tc is continuously assigned
// the value of the expression
assign tc = (cet && (count == length-1));

endmodule
```

An example of delays:

```
...
reg a, b, c, d;
wire e;
...
always @(b or e)
 begin
   a = b & e;
   b = a | b;
   #5 c = b;
   d = #6 c ^ e;
 end
```

The always clause above illustrates the other type of method of use, i.e. the always clause executes any time any of the entities in the list change, i.e. the b or e change. When one of these changes, immediately a and b are assigned new values. After a delay of 5 time

units, c is assigned the value of b and the value of c ^ e is tucked away in an invisible store. Then after 6 more time units, d is assigned the value that was tucked away.

Signals that are driven from within a process (an initial or always block) must be of type reg. Signals that are driven from outside a process must be of type wire. The keyword reg does not necessarily infer a hardware register.

Definition of Constants

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<*Width in bits*>'<*base letter*><*number*>

Examples:

- 12'h123 - Hexidecimal 123 (using 12 bits)
- 20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)
- 4'b1010 - Binary 1010 (using 4 bits)
- 6'o77 - Octal 77 (using 6 bits)

Synthesizeable constructs

As mentioned previously, there are several basic templates that can be used to represent hardware.

```
// Mux examples - Three ways to do the same thing.

// The first example uses continuous assignment
wire out ;
assign out = sel ? a : b;

// the second example uses a procedure
// to accomplish the same thing.

reg out;
always @(a or b or sel)
  out = sel ? a: b;

// Finally - you can use if/else in a
// procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

The next interesting structure is a transparent latch; it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and store it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

```
// Transparent latch example

reg out;
always @(gate or din)
 if(gate)
   out = din; // Pass through state
   // Note that the else isn't required here. The variable
   // out will follow the value of din while gate is high.
   // When gate goes low, latch_out will remain constant.
```

An edge-triggered latch is the next significant template; in verilog, the D-latch is the simplest, and it can be modeled as:

```
reg q;
always @(posedge clk)
  q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic rule of thumb is to use <= when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-latch is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
    q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```
reg q;
always @(posedge clk or posedge reset or posedge set)
 if(reset)
   q <= 0;
 else if(set)
   q <= 1;
 else
   q <= d;
```

The final basic variant is one that implements a D-latch with a mux feeding its input. The mux has a d-input and feedback from the flop itself. This allows a gated load function.

```
// Basic structure with feedback path illustrated
always @(posedge clk)
  if(gate)
    q <= d;
  else
    q <= q;

// The more common structure ASSUMES the feedback is present
// This is a safe assumption since this is how the
// hardware compiler will interpret it. This structure
// looks much like a Latch. The differences are the
// '''@(posedge clk)''' and the non-blocking '''<='''
//
always @(posedge clk)
  if(gate)
    q <= din; // the mux is "implied"
```

Looking at the original counter example you can see a combination of the basic asynchronous reset flop and Gated input flop used. The register variable **count** is set to zero on the rising edge or **rst**. When **rst** is 0, the variable **count** will load new data when **cet && cep** is true.

Initial and Always

There are two separate ways of declaring a verilog process. These are the **always** and the **initial** keywords. The **initial** keyword indicates a process that should begin running at time 0 in the simulation. The **always** keyword will run its process depending on the accompanying clause (almost always..)

```
//Examples:
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end

always @(a or b) // Anytime a or b CHANGE, run the process
  if (a)
    c = b;
  else
    d = ~b;


always @(posedge a)// Run whenever reg a has a low to high change
  a <= b;
```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always** keyword without the **@()** sensitivity list. It is possible to use always as shown below:

```
always
 begin // Always begins executing at time 0 and NEVER stops
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end // Keeps executing - so continue back at the top of the begin
```

The **always** keyword acts similar to the "C" construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

The example below is functionally identical to the **always** example above.

```
initial forever // Start at time 0 and repeat the begin/end forever
 begin
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end
```

Fork/Join

The **fork/join** pair are used by Verilog to create parallel processes. All statements (or blocks) between a fork/join pair begin execution simultaneously upon execution flow hitting the **fork**. Execution continues after the **join** upon completion of the longest running statement or block between the **fork** and **join**.

```
initial
 fork
   $write("A"); // Print Char A
   $write("B"); // Print Char B
   begin
     #1; // Wait 1 time unit
     $write("C");// Print Char C
   end
 join

The way the above is written, it is possible to have either the
sequences "ABC" or "BAC" print out. The order of simulation between the
first $write and the second $write depends on the simulator
implementation. This illustrates one of the biggest issues with
Verilog. You can have race conditions where the language execution
order doesn't guarantee the results.
```

Race Conditions

The order of execution isn't always guaranteed within verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```
initial
  a = 0;


initial
  b = a;

initial
  begin
    #1;
    $display("Value a=%b Value of b=%b",a,b);
  end
```

What will be printed out for the values of a and b? Well - it could be 0 and 0, or perhaps 0 and X! This all depends on the order of execution of the initial blocks. If the simulators scheduler works from the top of the file to the bottom, then you would get 0 and 0. If it begins from the bottom of the module and works up, then b will receive the initial value of a at the beginning of the simulation BEFORE it has been initialized to 0 (the value of any variable not set explicitily is set to X.) This is the way you can experience a race condition in a simulation. So be careful! Note that the 3rd initial block will execute as you expect because of the #1 there. That is a different point on the time wheel beyond time 0, consequently both of the earlier initial blocks have completed execution.

System tasks

System tasks are available to handle simple I/O, and various design measurement functions. You'll note that all the system tasks start with a **$**. This section presents a short list of the most often used tasks. It is by no means a comprehensive list.

- $display - Print a line followed by an automatic newline.
- $write - Print a line without the newline.
- $readmemh - Read hex file content into a memory array.
- $readmemb - Read binary file content into a memory array.
- $monitor - Print out all the listed variables when any change value.
- $time - Value of current simulation time.
- $dumpfile - Declare the VCD (Value Change Dump) format output file name.
- $dumpvars - Turn on and dump the variables.
- $random - Return a random value.

Program Language Interface (PLI)

Program Language Interface provides a programmer with transferring control from Verilog to a program function written in C language as well as provides task function named tf_putlongp(), tf_getlongp which are used to write and read the argument of the current verilog task or function respectively.

This capability enables Verilog to cooperate with other programs written in C language such as test harness, Instruction Set Simulator of microcontroller, debugger, etc.

References

- Johan Sandstrom. "Comparing Verilog to VHDL Syntactically and Semantically", *Integrated System Design, EE Times*, October 1995. — Sandstrom presents a table relating Verilog constructs to VHDL constructs.

See also

- VHDL
- SystemC
- SystemVerilog
- Property Specification Language

External links

Free Tools

- Icarus Verilog – The most popular open-source simulation and synthesis tool for Linux.
- LogicSim – A Windows-based Verilog simulator and debugger.
- Verilator – An open-source Verilog compiler.
- Veriwell – A Verilog-1995 simulator, which has recently been converted to open-source.
- Notepad++ – A Windows-based Verilog editor.

Further Information and Help

- Asic-World – Extensive free online tutorial with many examples.
- Comp.Lang.Verilog – Newsgroup for discussion of Verilog.
- Verilog F.A.Q. – Alternate Frequently Asked Questions
- A top-down approach to IC design contains useful information about using Verilog for chip design.
- Verilog.net – Premiere list of Verilog resources on the Internet.

Standards development

- IEEE Std 1364-2001 – The official standard for Verilog 2001 (not free).
- IEEE P1364 – Working group for Verilog (inactive).
- IEEE P1800 – Working group for SystemVerilog (replaces above).
- Verilog syntax – A description of the syntax in Backus-Naur form. This predates the IEEE-1364 standard.
- Verilog 2001 syntax – A heavily linked BNF syntax for Verilog 2001 (generated by EBNF tools).