


Hardware description language (HDL)

A hardware description language (HDL) is a computer-based language that describes the hardware of digital systems in a textual form.

It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behaviour of logic circuits.

It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behaviour of a digital system.

Design entry creates an HDL-based description of the functionality that is to be implemented in hardware. Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioural model.



In the public domain, there are two standard HDLs that are supported by the IEEE: VHDL and Verilog.

VHDL is a Department of Defense–mandated language. (The V in VHDL stands for the first letter in VHSIC, an acronym for very high-speed integrated circuit.)

Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard.

History


Verilog HDL was invented by Phil Moorby and Prabhu Goel around 1984. It served as a proprietary hardware modeling language owned by Gateway Design Automation Inc

In 1990, Gateway Design Automation Inc was acquired by Cadence Design System who recognised the value of Verilog, and realized that if Verilog remained as a closed language, the pressure of standardization would eventually drive people to shift to VHDL.

Submitted to IEEE and became IEEE standard 1364-1995, commonly referred as Verilog-95.

IEEE standard 1364-2001, signed variables (in 2's complement) became supported.

As of 2009, SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009).



The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development.

Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. Syntactic differences include: required bit-widths for variable declarations, demarcation of procedural blocks (Verilog uses begin/end instead of curly braces {}), and many other minor differences.

Verilog requires that variables be given a definite size. In C these sizes are assumed from the 'type' of the variable (for instance an integer type may be 8 bits).

Logic simulation

Logic simulation displays the behaviour of a digital system through the use of a computer.

A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals.

The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a test bench.

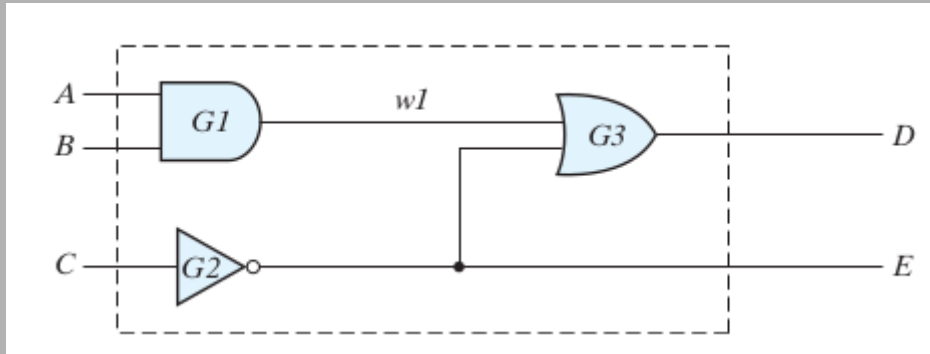
Logic synthesis

Logic synthesis is the process of deriving a list of physical components and their interconnections (called a netlist) from the model of a digital system described in an HDL. The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list.

Timing verification confirms that the fabricated, integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit. Propagation delays ultimately limit the speed at which a circuit can operate.


Timing verification checks each signal path to verify that it is not compromised by propagation delay. This step is done after logic synthesis specifies the actual devices that will compose a circuit.

Combinational Logic Modelled with Primitives



```
module Simple_Circuit (A, B, C, D, E);  
  output      D, E;  
  input       A, B, C;  
  wire        w1;  
  
  and         G1 (w1, A, B); // Optional gate instance name  
  not         G2 (E, C);  
  or          G3 (D, w1, E);  
endmodule
```

The port list of a module is the interface between the module and its environment. In this example, the ports are the inputs and outputs of the circuit. The logic values of the inputs to a circuit are determined by the environment; the logic values of the outputs are determined within the circuit and result from the action of the inputs on the circuit. The port list is enclosed in parentheses, and commas are used to separate elements of the list.



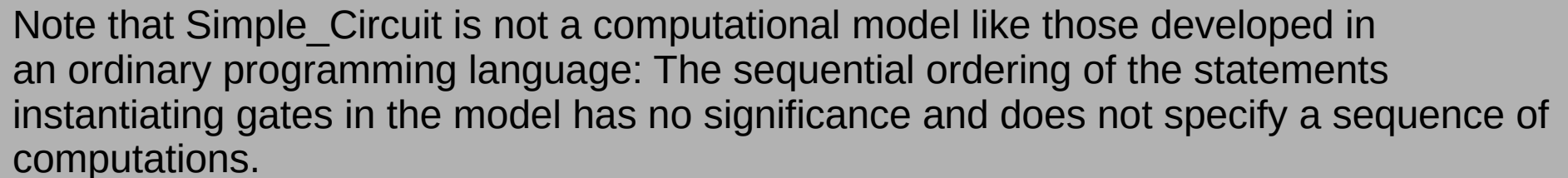
The structure of the circuit is specified by a list of (predefined) primitive gates, each identified by a descriptive keyword (and, not, or).

The elements of the list are referred to as instantiations of a gate, each of which is referred to as a gate instance .

Each gate instantiation consists of an optional name (such as G1, G2 , etc.) followed by the gate output and inputs separated by commas and enclosed within parentheses. The output of a primitive gate is always listed first, followed by the inputs. For example, the OR gate of the schematic is represented by the or primitive, is named G3 , and has output D and inputs w1 and E.

Note : The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.

The module description ends with the keyword endmodule. Each statement must be terminated with a semicolon, but there is no semicolon after endmodule.



Note that Simple_Circuit is not a computational model like those developed in an ordinary programming language: The sequential ordering of the statements instantiating gates in the model has no significance and does not specify a sequence of computations.

A Verilog model is a descriptive model. Simple_Circuit describes what primitives form a circuit and how they are connected.

The input–output behaviour of the circuit is implicitly specified by the description because the behaviour of each logic gate is defined.

Structural Hardware Modeling

Here we model circuits by specifying the internal structure of the block.

It is common to directly use the most primitive building-blocks available, such as, logic gates, or larger blocks which have already been defined elsewhere.


Verilog provides AND, NOT, OR, XOR, NAND, XNOR, NOR gates, among others. Verilog allows each logic gate to have any valid number of inputs.

For instance, a single NOR gate can have four inputs, computing

$$z = \overline{(x_1 \oplus x_2 \oplus x_3 \oplus x_4)}.$$


In Verilog we can request an XNOR function to be applied to its inputs, as if we were "calling a function",

```
xnor xnor1(z,x1,x2,x3,x4);
```



There are different points of view regarding how an exclusive-OR gate with more than two inputs should behave. Most often such an XOR gate behaves like a cascade of 2-input gates and performs an odd-parity function. This is what verilog does.

However, some people interpret the meaning of exclusive-OR more literally and say that the output should be a 1 if and only if exactly one of the inputs is a 1.



Usually the output signals in the predefined gates appear first in the argument list of the gate and the input signals appear afterwards.

The identifier `xnor1` is associated with this particular instance of an XNOR gate.

Identifiers are not mandatory for gate instances. Any other gate can be similarly requested by mentioning its name and then listing the identifiers that correspond to its output and input signals.

For instance the following

```
or or1(z,x1,x2,x3,x4);  
and and1(x1,a,b);  
and and2(x2,a,c);  
not not1(na,a);  
not not2(ne,e);  
and and3(x3,na,d);  
and and4(x4,ne,d);
```

represents in "structural" form the logic network which computes

$$z = a \cdot b + a \cdot c + \bar{a} \cdot d + \bar{e} \cdot d.$$

The variable names acting as arguments in the logic gate functions, implicitly show the internal connections among the various gates themselves. These variables are of the "wire" type.

The development of a logic circuit from scratch, starting from primitive gates is called structural or "gate-level" modeling.

Using Numeric Constants in Verilog

In Verilog constants are specified in the traditional form of a series of digits with or without a sign, but also in the following form

`<size> <base format> <number>`

where `<size>` contains decimal digits that specify the size of the constant in the number of bits.

The `<size>` is optional.

The `<base format>` is the single character ' followed by one of the following characters b, d, o and h, which stand for binary, decimal, octal and hexadecimal, respectively.

The `<number>` part contains digits which are legal for the `<base format>`.

Some examples:

`549` // decimal number

`'h08FF` // 16-bit hexadecimal number

`4'b11` // 4-bit binary number 11 (0011)

`3'b10x` // 3-bit binary number with least significant bit unknown/don't care

`12'o1345` // 12-bit octal number

`5'd3` // 5-bit decimal number 3 (00011)

`-4'b11` // 4-bit two's complement negation of binary number 11 (1101)

Four-valued Logic

Verilog uses a form of four-valued logic rather than the familiar two-valued Boolean logic.

The four values are

0 (zero, false),

1 (one, true),

x which if seen as output means that Verilog does not know how to compute the value for this output (either because it is driven by incompatible signals simultaneously or because no input is driving it)

z high-impedance value.

x and z combine with 0 and 1 in interesting ways. For example, $x \text{ AND } 1 = x$, but $x \text{ AND } 0 = 0$.

Icarus Verilog for Windows

Icarus Verilog is a free compiler implementation for the IEEE-1364-2005 Verilog hardware description language. Icarus is maintained by Stephen Williams and it is released under the **GNU GPL license**.

<http://bleyer.org/icarus/>

Verilog Virtual Processor


vvp is the run time engine that executes the default compiled form generated by Icarus Verilog. The output from the iverilog command is not by itself executable on any platform. Instead, the vvp program is invoked to execute the generated output file.

VCD is an acronym for Value Change Dump. It notes the changes in signal values for a simulation in an external file so that waveform viewers can view the results. This dump format is defined by the Verilog standard, and so is most ubiquitous.

GTKWave is a fully featured **GTK+** based wave viewer. GTK+, or the GIMP Toolkit, is a multi-platform toolkit for creating graphical user interfaces.

```
iverilog sim.v testsim.v -o sim  
vvp sim
```

```
gtkwave sim.vcd
```

You should use a text editor with syntax highlighting to write your verilog programs.
gedit is good.

Do not use word processors which may inset special character for quote symbols etc.

Timescale specifies the time unit and time precision of a module that follow it. The simulation time and delay values are measured using time unit. The precision factor is needed to measure the degree of accuracy of the time unit, in other words how delay values are rounded before being used in simulation.

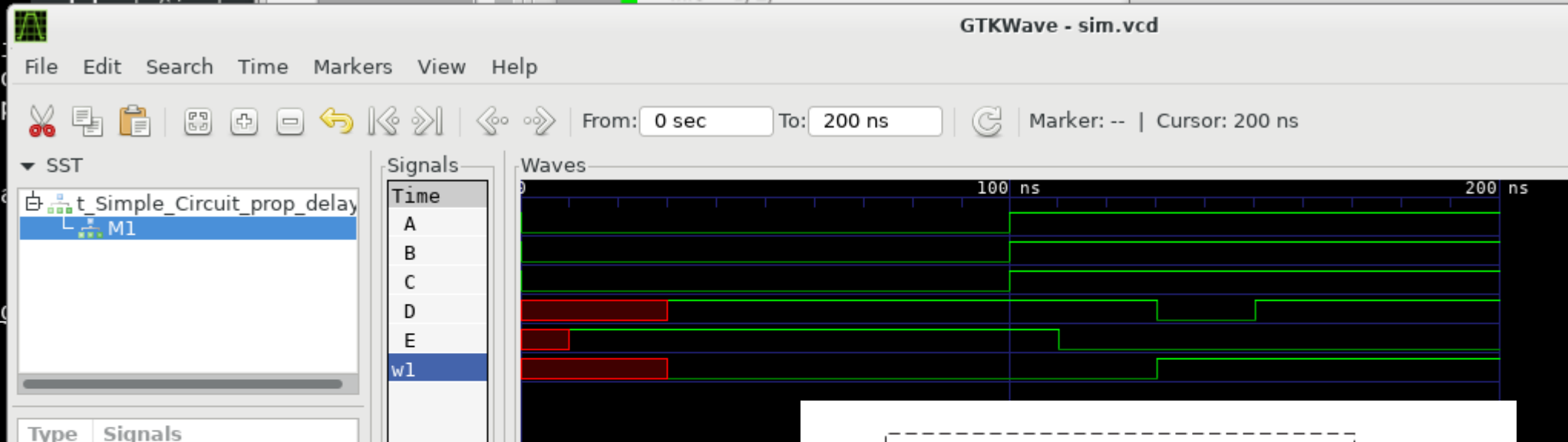
```
// Verilog model of simple circuit with propagation delay
```

```
`timescale 1ns/100ps  
`default_nettype none
```

```
module Simple_Circuit_prop_delay (A, B, C, D, E);  
    output D, E;  
    input A, B, C;  
    wire w1;  
    and #(30) G1 (w1, A, B);  
    not #(10) G2 (E, C);  
    or #(20) G3 (D, w1, E);  
endmodule
```

```
// Test bench for Simple_Circuit_prop_delay  
`default_nettype none
```

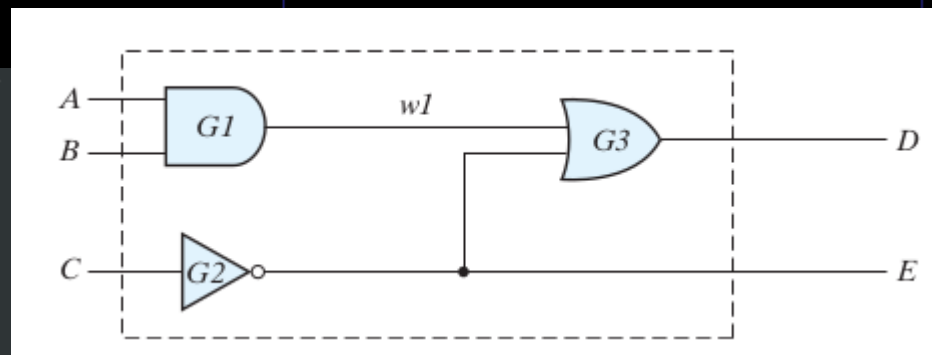
```
module t_Simple_Circuit_prop_delay;  
    wire D, E;  
    reg A, B, C;  
    Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required  
  
    initial  
        begin  
            $dumpfile("sim.vcd");  
            $dumpvars(0, t_Simple_Circuit_prop_delay);  
            A = 1'b0; B = 1'b0; C = 1'b0;  
            #100 A = 1'b1; B = 1'b1; C = 1'b1;  
        end  
  
    initial #200 $finish;  
endmodule
```



```
// Verilog model of simple circuit with propagation delay
```

```
`timescale 1ns/100ps
`default_nettype none
```

```
module Simple_Circuit_prop_delay (A, B, C, D, E);
    output D, E;
    input A, B, C;
    wire w1;
    and #(30) G1 (w1, A, B);
    not #(10) G2 (E, C);
    or #(20) G3 (D, w1, E);
endmodule
```



```
// Test bench for Simple_Circuit_prop_delay
`default_nettype none
```

```
module t_Simple_Circuit_prop_delay;
    wire D, E;
    reg A, B, C;
    Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

    initial
        begin
            $dumpfile("sim.vcd");
            $dumpvars(0, t_Simple_Circuit_prop_delay);
            A = 1'b0; B = 1'b0; C = 1'b0;
            #100 A = 1'b1; B = 1'b1; C = 1'b1;

        end


    initial #200 $finish;
endmodule
```

Beware of Implicit Declaration

If an undeclared identifier is used as a connection to an instance of a module, interface, program, or primitive, then an implicit net is inferred, and no error or warning is reported.

```
module bad_adder (input wire a, b, ci,  
                  output wire sum, co);  
    wire n1, n2, n3;  
    xor g1 (n1, a, b);  
    xor g2 (sum, n1, ci); // GOTCHA!  
    and g3 (n2, a, b, c); // GOTCHA!  
    and g4 (n3, n1, ci);  
    or g5 (co, n2, n3);  
endmodule
```

Use ``default_nettype none` of compile with `-Wimplicit` or `-Wall`



Net types: (wire) Physical connection between structural elements.

Value assigned by a continuous assignment or a gate output.

Register type: (reg, integer, time, real, real time) represents abstract data storage element. Assigned values only within an always statement or an initial statement.

The main difference between wire and reg is wire cannot hold (store) a value when there no connection between a and b.

If there is no connection in a and b, the wire loses its value.

But reg can hold a value even if there is no connection.