

Verilog HDL

Hardware Description Language (HDL)

- Basic idea is a programming language to describe hardware
 - Initial purpose was to allow abstract design and simulation
 - Design could be verified then implemented in hardware
 - Now Synthesis tools allow direct implementation from HDL code.
 - Large improvement in designer productivity
-

HDL

- HDL allows write-run-debug cycle for hardware development.
 - Similar to programming software
 - Much, much faster than design-implement-debug
 - Combined with modern Field Programmable Gate Array chips large complex circuits (100000s of gates) can be implemented.
-

HDLs

- There are many different HDLs
 - Verilog HDL
 - ABEL
 - VHDL
-

Verilog HDL

- Verilog HDL is most common
 - Easier to use in many ways = better for teaching
 - C - like syntax
 - History
 - Developed as proprietry language in 1985
 - Opened as public domain spec in 1990
 - Due to losing market share to VHDL
 - Became IEEE standard in 1995
-

Abstraction Levels

- Verilog is both a behavioral and a structural language.
 - Internals of each module can be defined at four levels of abstraction, depending on the needs of the design.
 - The module behaves identically with the external environment irrespective of the level of abstraction at which the module is described.
 - The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment.
 - The levels are defined below.
-

Abstraction Levels

- Behavioral or algorithmic level
 - Dataflow level
 - Gate level
 - Switch level
-

Behavioral or algorithmic level

- This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.
-

Dataflow level

- At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.
-

Gate level

- The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.
-

Switch level

- This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.
-

Register Transfer Level

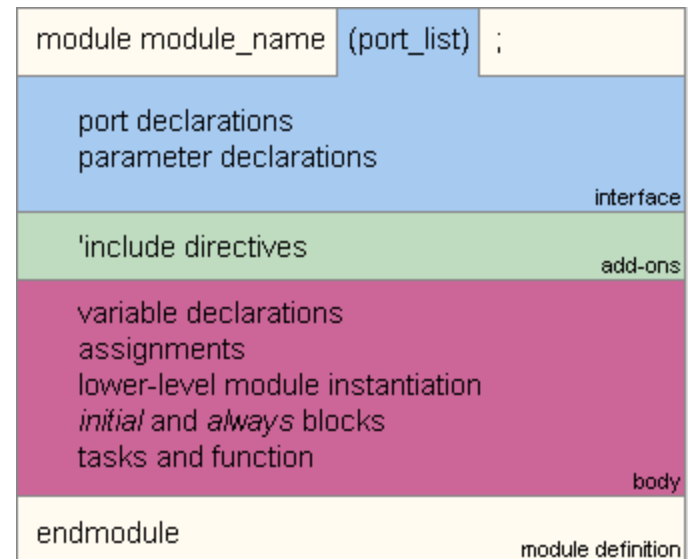
- Verilog allows the designer to mix and match all four levels of abstractions in a design.
 - In the digital design community, the term register transfer level (RTL) is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.
-

Verilog HDL

- Verilog constructs are use defined *keywords*
 - Examples: and, or, wire, input output
- One important construct is the *module*
 - Modules have inputs and outputs
 - Modules can be built up of Verilog primitives or of user defined submodules.

Definition of Module

- Interface: port and parameter declaration
- Body: Internal part of module
- Add-ons (optional)



Some points to remember

- The name of Module
 - Comments in Verilog
 - One line comment (`//`)
 - Block Comment (`/*.....*/`)
 - Description of Module (optional but suggested)
-

Module Structure

module *module_name* (*port_list*);

port declarations

*data type
declarations*

circuit functionality

timing specifications

endmodule

Module Port

■ Port List:

- A listing of the port names
- Example:

```
module rca_4bit (sum, c_out , a, b);
```

■ Port Types:

- input --> input port
- output --> output port
- inout --> bidirectional port

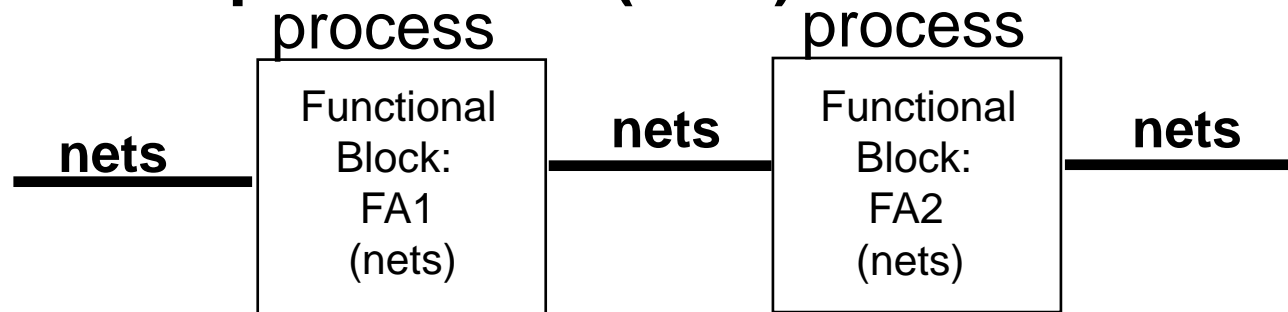
■ Port Declarations:

- `<port_type> <port_name>;`
- Example:

```
input [3:0] a, b;  
input overflow;  
output [3:0] out;
```

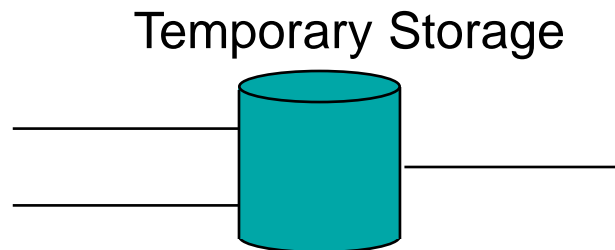
Data Types(1)

- **Net Data Type - represent physical interconnect between processes. (wire)**



- **Register Data Type - represent variable to store data temporarily. (reg)**

- It does not represent a physical (hardware) register.

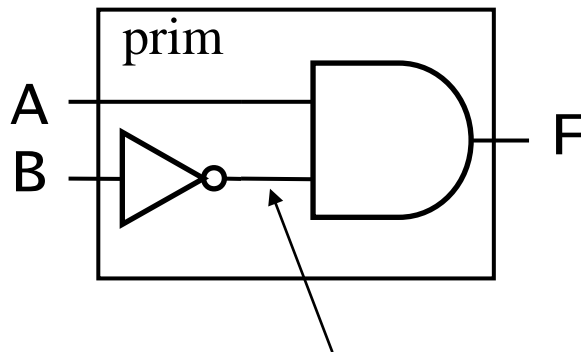


Data Type(2)

- **wires and registers can take one of {1, 0, x, z} : 4-valued logic**
 - x : either uninitialized variable or a conflict (ex: two signals shorted?)
 - z : high impedance or a floating value (used for tri-state buses)
- **Bus Declarations:**
 - `<data_type> [MSB : LSB] <signal name> ;`
 - `<data_type> [LSB : MSB] <signal name> ;`
- **Examples:**
 - `wire <signal name> ;`
 - `wire [15:0] mult_out, adder_out;`
 - `reg <signal name> ;`
 - `reg [7 : 0] out ;`
- **Constants**
 - radix : B,b (binary), O,o (octal), D,d (decimal), H,h (hexadecimal)
 - examples : 15, 4'd15 (decimal 15)
 'h15 (decimal 21, hex 15)
 5'b10011 (decimal 19, binary 10011)
 12'h01F (decimal 31, hex 01F)

Verilog Wires

- Wire is an internal connection between primitives



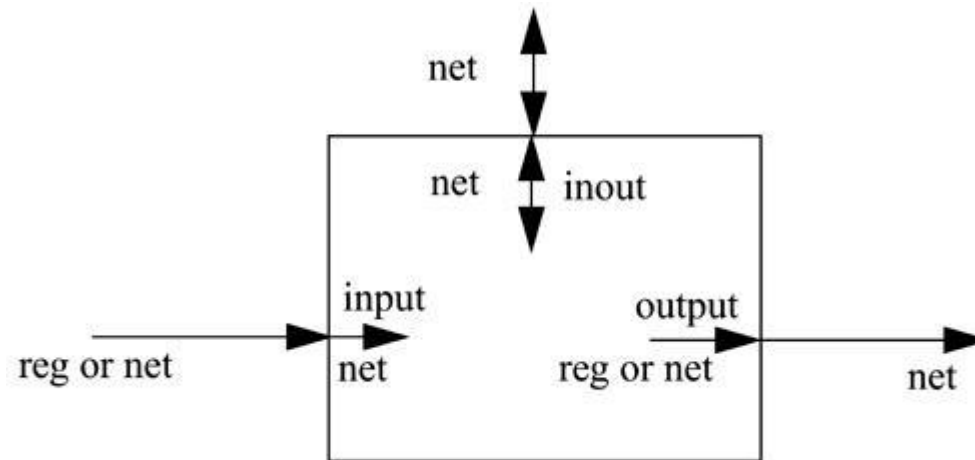
```
module prim(A,B,F);  
    input A, B;  
    output F;  
    wire w;  
    not (w,B);  
    and (F,A,w);  
endmodule
```

wire

Port Connection Rules

- One can visualize a port as consisting of two units, one unit that is internal to the module and another that is external to the module.
 - The internal and external units are connected.
 - There are rules governing port connections when modules are instantiated within other modules.
 - The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure.
-

Port Connection Rules



Verilog Operator Types

Operator	Argument	Result
Arithmetic	Pair of operands	word
Bitwise	Pair of operands	word
Reduction	Single operand	bit
Logical	Pair of operands	bit
Relational	Pair of operands	bit
Shift	Single operand	word
Conditional	Three operands expression	

Arithmetic Operator's

Operator	Operation	Example
+	Arithmetic +	$C = A + B$
-	Arithmetic -	$C = A - B$
*	Arithmetic *	$C = A * B$
/	Arithmetic /	$C = A /$
%	Modulus	$C = A \% B$

Operator	Argument	Result
Arithmetic	Pair of operands	word

Bitwise Operator's

Operator	Operation	Example
~	Bitwise Negation	$C = \sim A$
&	Bitwise And	$C = A \& B$
	Bitwise Or	$C = A B$
^	Bitwise Exclusive-Or	$C = A \wedge B$
$\sim \wedge$ or $\wedge \sim$	Bitwise Equivalence	$C = A \sim \wedge B$

Operator	Argument		Result
Bitwise	Pair of operands	word	

Reduction Operator's

Operator	Operation	Example
&	Reduction And	$C = \&(A)$
$\sim\&$	Reduction Nand	$C = \sim\&(A)$
	Or	$C = (A)$
$\sim $	Reduction Nor	$C = \sim (A)$
\wedge	Reduction Exclusive-Or	$C = \wedge(A)$
$\sim\wedge$ or $\wedge\sim$	Reduction Exclusive-Nor	$C = \sim\wedge(A)$

Operator	Argument	Result
Reduction	Single operand	bit

Logical Operator's

Operator	Operation	Example
!	Logical negation	if(!A)
&&	Logical and	if((A) && (B))
	Logical or	if((A) (B))
==	Logical equality	if (A == B)
!=	Logical inequality	if (A != B)

Operator	Argument	Result
Logical	Pair of operands	bit

Relational Operator's

Operator	Operation	Example
>	Greater Than	if (A > B)
<	Less Than	if (A < B)
>=	Greater Than or Equal if (A >= B)	
<=	Less Than or Equal	if (A <= B)

Operator	Argument	Result
Relational	Pair of operands	bit

Other Operator's

<<	Left Shift	$A = A \ll 1$
>>	Right Shift	$A = A \gg 3$
?:	Conditional	$Y = A ? C : D;$
or	Event Or	always @ (clk or sel)
{}, {{ }}	Concatenation	{c_out, sum}
		$\{4\{a\}\} = \{a, a, a, a\}$

Operator	Argument	Result
Shift	Single operand word	
Conditional	Three operands	expression

Language Rules

- Case Sensitive `My_input` vs. `my_input`
- Space Free
- Upper and Lower case letters
- Digits
- Underscore
- Dollar Sign
- Variable cannot begin with Number or \$
- Reserved Keywords
- Vectors indicate more than one wire `sum[3:0]`

Assignments

- Blocking
B = A;
C = B;
D = C; //A=B=C=D
- Non-Blocking
B <= A;
C <= B;
D <= C; //shift

Signal Levels

Level	Description
0	Logic value 0
1	Logic value 1
z	Tri-state (high impedance)
x	Unknown value

Nested Modules

```
module halfadder (sum, c_out, a, b);  
input a, b;  
output sum, c_out;  
xor (sum, a, b);  
and (c_out, a, b);  
endmodule
```

Nested Modules

```
module fulladder (sum, c_out, a, b, c_in);  
input a, b, c_in;  
output sum, c_out;  
wire w1, w2, w3;  
halfadder HA1(w1, w2, a, b);  
halfadder HA2(sum, w3, c_in, w1);  
or (c_out, w2, w3);  
endmodule
```

Circuit to code

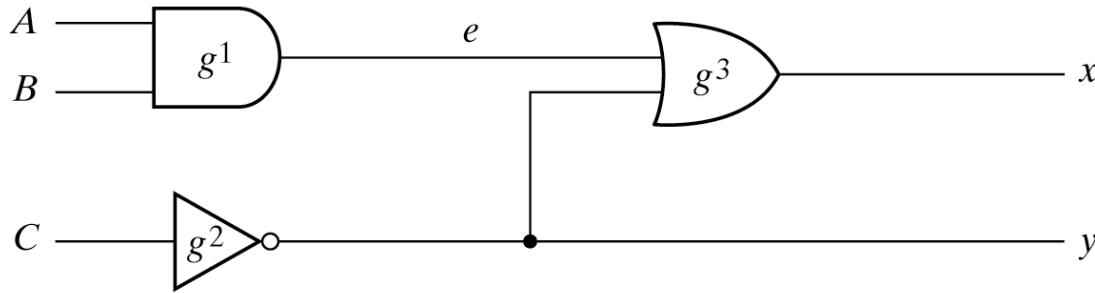


Fig. 3-37 Circuit to Demonstrate HDL

```
module  smpl_circuit (A,B,C,x,y) ;  
    input  A,B,C;  
    output x,y;  
    wire  e;  
    and   g1 (e,A,B) ;  
    not   g2 (y, C) ;  
    or    g3 (x,e,y) ;  
endmodule
```

Input signals

- In order to simulate a circuit the input signals need to be known so as to generate an output signal.
 - The input signals are often called the circuit *stimulus*.
 - An HDL module is written to provide the circuit stimulus. This is known as a *testbench*.
-

Testbench

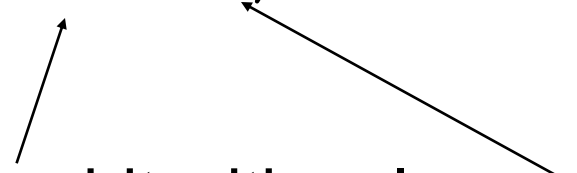
- The *testbench* module includes the module to be tested.
 - There are no input or output ports for the testbench.
 - The inputs to the test circuit are defined with **reg** and the outputs with **wire**.
 - The input values are specified with the keyword **initial**
 - A sequence of values can be specified between **begin** and **end**.
-

Signal Notation

- In Verilog signals are generalised to support multi-bit values (e.g. for buses)

- The notation

`A = 1'b0;`

A diagram showing the Verilog notation '1'b0'. Two arrows originate from below the text. One arrow points to the '1' character, and the other points to the '0' character. This highlights the bit width and the bit value respectively.

- means signal A is one bit with value zero.

- The end of the simulation is specified with **`$finish.`**

Verilog Module

- Description of internal structure/function
 - Implicit semantic of time associated with each data object/signal
 - Implementation is hidden to outside world
- Communicate with outside through ports
 - Port list is optional
- Achieve hardware encapsulation

```
module Add_half ( sum, c_out, a, b );  
  input a, b;  
  output      sum, c_out;  
  wire c_out_bar;  
  
  xor (sum, a, b);  
  nand (c_out_bar, a, b);  
  not (c_out, c_out_bar);  
endmodule
```

