Catalog

5.1 Hardware Description Language (HDL)

In electronics, a hardware description language or HDL is any language from a class of computer languages and/or programming languages for formal description of digital logic and electronic circuits. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.

HDLs are standard text-based expressions of the spatial and temporal structure and behaviour of electronic systems. In contrast to a software programming language, HDL syntax and semantics include explicit notations for expressing time and concurrency, which are the primary attributes of hardware. Languages whose only characteristic is to express circuit connectivity between hierarchies of blocks are properly classified as netlist languages used on electric computer-aided design (CAD).

HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

Design using HDL

The vast majority of modern digital circuit design revolves around an HDL description of the desired circuit, device, or subsystem.

Most designs begin as a written set of requirements or a high-level architectural diagram. The process of writing the HDL description is highly dependent on the designer's background and the circuit's nature. The HDL is merely the 'capture language'—often begin with a high-level algorithmic description such as MATLAB or a C++ mathematical model. Control and decision structures are often prototyped in flowchart applications, or entered in a state-diagram editor. Designers even use scripting languages (such as Perl) to automatically generate repetitive circuit structures in the HDL language. Advanced text editors (such as Emacs) offer editor templates for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

As the design's implementation is fleshed out, the HDL code invariably must undergo code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers enforce standardized code a guideline, identifying ambiguous code constructs before they can cause misinterpretation by downstream synthesis, and check for common logical coding errors, such as dangling ports or shorted outputs.In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, this netlist is passed off to the back-end stage. Depending on the physical technology (FPGA, ASIC gate-array, ASIC standard-cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL-description. Finally, a silicon chip is manufactured in a fab.

HDL and programming languages

A HDL is analogous to a software programming language, but with major differences. Programming languages are inherently procedural (single-threaded), with limited syntactical and semantic support to handle concurrency. HDLs, on the other hand, can model multiple parallel processes (such as flipflops, adders, etc.) that automatically execute independently of one another. Any change to the process's input automatically triggers an update in the simulator's process stack. Both programming languages and HDLs are processed by a compiler (usually called a synthesizer in the HDL case), but with different goals. For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms: a "simulation" netlist with gate-delay information, a "handoff" netlist for post-synthesis place and route, or a generic industry-standard EDIF format (for subsequent conversion to a JEDEC-format file).

On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor. As HDLs and programming languages borrow concepts and features from each other, the boundary between them is becoming less distinct. However, pure HDLs are unsuitable for general purpose software application development, just as general-purpose programming languages are undesirable for modeling hardware. Yet as electronic systems grow increasingly complex, and reconfigurable systems become

increasingly mainstream, there is growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming. SystemC is an example of such—embedded system hardware can be modeled as non-detailed architectural blocks (blackboxes with modeled signal inputs and output drivers). The target application is written in C/C++, and natively compiled for the host-development system (as opposed to targeting the embedded CPU, which requires host-simulation of the embedded CPU). The high level of abstraction of SystemC models is well suited to early architecture exploration, as architectural modifications can be easily evaluated with little concern for signal-level implementation issues.

In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level of the design. Companies such as Cadence, Synopsys and Agility Design Solutions are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs. Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, and in the Impulse C tools from Impulse Accelerated Technologies. Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry. Languages such as SystemVerilog, SystemVHDL, and Handel-C seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive versus making FPGAs more accessible to existing software engineers. Thus SystemVerilog is more quickly and widely adopted than SystemC. There is more information on C to HDL and Flow to HDL in their respective articles.

5.2 VHDL Operators

Highest precedence first,

left to right within same precedence group, use parenthesis to control order.

Unary operators take an operand on the right.

"result same" means the result is the same as the right operand. Binary operators take an operand on the left and right.

"result same" means the result is the same as the left operand.

** exponentiation, numeric ** integer, result numeric

abs absolute value, abs numeric, result numeric

not complement, not logic or boolean, result same

* multiplication, numeric * numeric, result numeric

/ division, numeric / numeric, result numeric

mod modulo, integer mod integer, result integer

rem remainder, integer rem integer, result integer

+ unary plus, + numeric, result numeric

- unary minus, - numeric, result numeric

+ addition, numeric + numeric, result numeric

- subtraction, numeric - numeric, result numeric

& concatenation, array or element & array or element,

result array

sll shift left logical, logical array sll integer, result same srl

shift right logical, logical array srl integer, result same sla

shift left arithmetic, logical array sla integer, result same sra

shift right arithmetic, logical array sra integer, result same rol

rotate left, logical array rol integer, result same

ror rotate right, logical array ror integer, result same

= test for equality, result is boolean

/= test for inequality, result is boolean

< test for less than, result is boolean

<= test for less than or equal, result is boolean

> test for greater than, result is boolean

>= test for greater than or equal, result is boolean

and logical and, logical array or boolean, result is same

or logical or, logical array or boolean, result is same

nand logical complement of and, logical array or boolean, result is same

nor logical complement of or, logical array or boolean, result is same

xor logical exclusive or, logical array or boolean, result is same

xnor logical complement of exclusive or, logical array or boolean, result is same

5.3 Packages

A package is a collection of logically related declarations. They typically are used to contain sets of type subtypes, constants declarations, global signal declarations, and global variables (VHDL'93). The packages may contain component declarations, attribute declarations and specifications and subprograms. The subprograms declared in packages provide information hiding; they can be called from outside the package but the body of the subprograms remains hidden and protected from users.

Normally a package is defined in two parts:

- a package declaration, which defines the visible contents of a package, and
- a package body, which provides the implementation details of subprograms and the actual values of deferred constants.

The body part may be omitted if there are no subprogram implementations or deferred constants.


```
package simple is
constant cdiff: integer; -- deferred constant
subtype word is bit_vector(15 downto 0);
subtype address is bit_vector (24 downto 0);
type mem is array (0 to 31) of word;
function address2int (val : address) return integer;
function increment_word(val : word) return word;
end simple;
```

In the example above, the bodies of the two functions address2int and increment_word are not specified, so a package body is needed.

Note the deferred constant cdiff is initialized in the package body; its value may be changed by re-analyzing only the package body.

The package body can be written as:

```
package body simple is
constant cdiff: integer:= 500; -- deferred constant initialized
address_to_int(value : address) return integer is
-- the definition of the function
end address2int;
function increment_word( val : word) return word is
-- the definition of the function
end increment_word;
end simple;
```


Note that the subtype declarations are not repeated, they are visible from the package declarations block.

use clause

Items declared within a package become accessible by direct selection:

```
variable pc: simple.address; -- simple is the package name
```

or by the use clause:

```
use work.simple.address; -- work is the name of working library where the package is compiled
variable pc: address;     -- direct visibility
```

If all of the declared names in a package are to be used in this way, you can indicate it through special suffix all , for example:

use work.simple.all ;

variable pc: address;

variable ram: mem;

Standard packages

There are two predefined packages provided with VHDL:

- standard package including various data and type definitions (e.g. bit_vector, string, ..)
- textio package including basic read/write procedures (e.g. read, readline, write, writeline, ..)

package standard is

type severity_level is (note, warning, error, failure);

type boolean is (false, true);

type bit is (`0', `1')

type character is (

NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FSP, GSP, RSP, USP, ` `, `!', `"', `#', `$', `%', `&', `', `\ `, `*', `+', `,', `-', `.', `/', `0', `1', `2', `3', `4', `5', `6', `7', `8', `9', `:', `;', `<`, `=', `>', `?', `@', `A', `B', `C', `D', `E', `F', `G', `H', `I', `J', `K', `L', `M', `N', `O', `P', `Q', `R', `S', `T', `U', `V', `W', `X', `Y', `Z', `[`, `\\', `]', `^', `_', ``', `a', `b', `c', `d', `e', `f', `g', `h', `i', `j', `k', `l', `m', `n', `o', `p', `q', `r', `s', `t', `u', `v', `w', `x', `y', `z', `{`, `|', `}', `~', DEL);

type time is range implementation_defined units fs;

ps = 1000 fs;

ns = 1000 ps;

us = 1000 ns;

ms = 1000 us;

```
sec = 1000 ms;

min = 60 sec;

hr = 60 min;

end units ;

function now return time; -- returns the current simulation time

subtype natural is integer range 0 to integer'high;

subtype positive is integer range 1 to integer'high;

type bit_vector is array (natural range <>) of bit;

type string is array (positive range <>) of character;

end standard;

package textio is

type line is access string; -- a line is a pointer to a string value

type text is file of string; -- a file of variable-length ASCII records

type side is (right, left); -- for justifying output data within fields

subtype width is natural; -- for specifying widths of output fields

-- standard text files

file input: text is in "STD_INPUT";

file utput: text is out "STD_OUTPUT";

-- input procedures for standard types

procedure readline(f:in text; l:out line);

procedure read(:inout line; value:out le_type;good: out boolean);
```

```
procedure read(:inout line; value:out le_type);

-- for le_type in: bit, bit_vector, boolean, character, integer, real, string, time

procedure writeline(f:out text; l:in line);

procedure write(l:inout line; value:in le_type;justified:in side:=right; field:in width:=0);

-- for le_type in: bit, bit_vector, boolean, character, integer, string

procedure write(l:inout line; value:in real;justified:in side:=right; field:in width:=0; digits: in natural:=0);

procedure write(l:inout line; value:in time;justified:in side:=right; field:in width:=0; unit: in time:=ns);

-- file position predicates

function endline (l:in line) return boolean;

function endfile (f:in text) return boolean;

end textio;
```


Libraries

All VHDL compilers store the compiled designs in design libraries. The standard package is already compiled into std library and used implicitly by VHDL compiler. Other VHDL files must be explicitly compiled into design libraries before the use by simulator.

Library named work is used when no explicit user designed library is provided.

```
use work.all;

entity using_work is

...
```

For comprehensive designs the compiled vhdl modules should be placed in design libraries other than work. The library name should reflect the nature of the compiled components (e.g. CMOS_logic).

```
library CMOS_logic;
```

use work.CMOS_logic.package_gates.all;

entity CMOS_gates is

...

One of the well known libraries is IEEE standard logic library. If the compiled vhdl module requires standard logic , IEEE library must be included into the compilation process.

library IEEE;

use IEEE.std_logic_1164.all;

entity using_std_logic is

...

The synthesis tools use standard numeric packages. These numeric packages contain the functions which allow to apply the numeric operators to bit vectors. Depending on the use of simple bit vectors and standard logic vectors two numeric packages are provided:

- numeric_bit package based on VHDL type bit and
- numeric_std package based on the subtype std_ulogic

library IEEE;

use IEEE.std_logic_1164.all;

use IEEE.numeric_std.all;

...

5.4 VHDL-Entity Declaration.

Introduction:

The following section explains the Entity declaration in details.

Entity Declaration:

The entity' declaration specifies the name of the entity being modeled and lists the set of interface ports.

Ports are signals through which the entity communicates with the other models in its external environment.

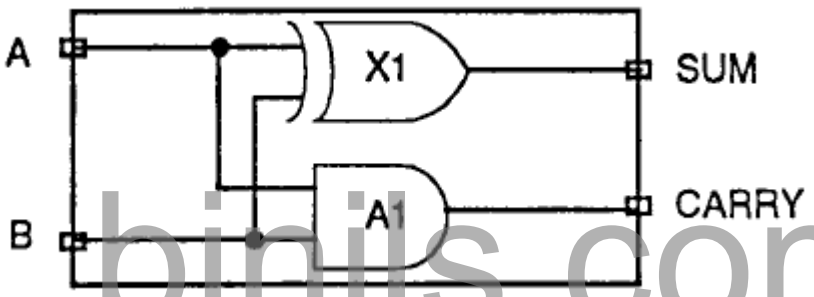


Figure 5.4.1 A half-adder circuits

entity HALF_ADDER is

port (A, B: in BIT; SUM, CARRY: **out** BIT);

end HALF_ADDER;

The entity, called HALF_ADDER, has two input ports, A and B (the mode in specifies input port), and two output ports, SUM and CARRY (the mode out specifies output port).

BIT is a predefined type of the language; it is an enumeration type containing the character literals '0' and '1'.

The port types for this entity have been specified to be of type BIT, which means that the ports can take the values, '0' or '1'.

5.5 VHDL Behavioural Modelling Style

The behavioral modeling describes how the circuit should behave.

For these reasons, behavioral modeling is considered highest abstraction level as compared to data-flow or structural models.
The VHDL synthesizer tool decides the actual circuit implementation.

The VHDL behavioral model is widely used in test bench design, since the test bench design doesn't care about the hardware realization

Introduction to signal assignment

In VHDL-models, stimuli and responses occur through **signals**.

A signal can be declared only and exclusively in sections of concurrent code namely the architecture declaration section between the keyword **is** and **begin**.

A signal can be assigned:

to **ports**, which it is connected to

into **processes** which elaborates it

The simplest of these processes is the **signal assignment statement**.

Signal assignment statement

It will be shown that, according to the situation in which it is used, a signal assignment statement can be used in both concurrent and sequential code sections.

Its syntax is:

T <= S;

Meaning that the target signal T takes the value of source signal S.

This statement is executed whenever signal "T" change its value.

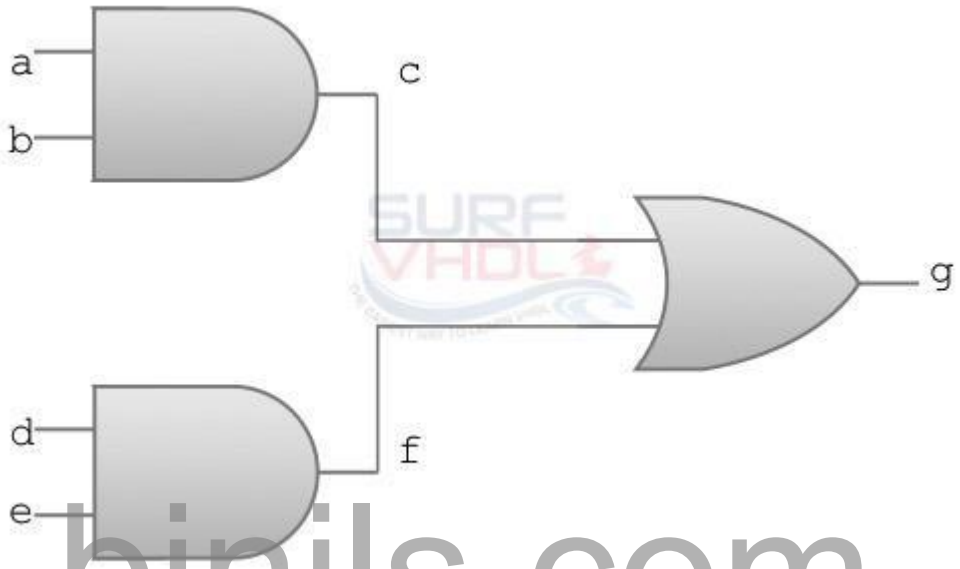
Signal "S" is in the sensitivity list of this statement.

VHDL Behavioral Modeling Style Example

In the structural modeling style page is reported the example of and_or entity structural architecture implementation.
The following VHDL code implements the same functionality using a behavioral approach:

```
entity and_or is
port(
a : in std_logic;
b : in std_logic;
d : in std_logic;
e : in std_logic;
g : out std_logic);
```

```
end and_or;
architecture and_or_a of and_or is
-- declarative part: empty
begin
g <= (a and b) or (d and e);
end and_or_a;
```



5.6 VHDL Structural Modelling Style

The Structural Modelling is very similar to the schematic entry, in this case implemented as text instead of graphically.

As digital designs become more complex, it becomes less likely that we can use only one of the three-implementation styles seen before. The result is the use of the hybrid VHDL model.

The term structural modelling is the terminology that VHDL uses for the modular design:

if you are designing a complex project, you should split in two or more simple design in order to easy handle the complexity.

The benefits of modular design in VHDL are similar to the benefits that modular design or object-oriented design provides for higher-level computer languages.

Modular designs allow you to pack low-level functionality into modules.

This approach allows a design reuse without the need to reinvent and re-test the wheel every time.

Next step is the hierarchical approach where you can extend beyond the structural coding modeling.

In this case in the TOP Level design, are instantiated a

- design1
    - *sub-design 1*
    - *sub-design 2*
- design2
    - *sub-design 3*
    - *sub-design 4*

Structural architecture declaration

In order to instance a component inside a design, you shall:

- Declare the components in the declarative part of the architecture
- Instance the component in the architecture statement section

The declaration section of the architecture is included between the keyword “is” and “begin”

architecture *architecture_name* of *entity_name* is

-- ARCHITECTURE declarative part

begin

-- ARCHITECTURE statement part

end *architecture_name*;

The statement section or concurrent section of the architecture is included between the keyword “begin” and “end”.

The component declaration to be inserted in the declarative section of the architecture can be summarized as follow:

- *Copy* the entity declaration relative to the component
- *Replace* the keyword ”entity” with the keyword ”component”
- *Delete* keyword ”is”. If you use VHDL 1993 standard or above you can leave the keyword ”is”
- *Replace* the entity name after the keyword “end” with the keyword “component”.

Component Instantiation

- *Copy* component declaration in the architecture concurrent area
- *Replace* the keyword “component” with the instance name and add the keyword “map” after the generic and port clause.

In order to map generic and port

- *Replace* the comma with the two symbols equal greater than

- *Replace* the semicolon with colon.

An example will clarify better than thousand explanations.

Structural architecture declaration example

In this example we want to realize the structural implementation the entity *and_or* in figure below

There are 4 input ports ***a, b, d, e*** and one output port ***g***. The instance ***u1*** and ***u2*** of the two **AND gate** are connected to the ***u3*** using two wire named ***c*** and ***f.*** Figure above shows the entity declaration for **AND2** and **OR2** component.

In this moment, we don't care about the architecture of **AND2** and **OR2** entity. It could be structural with other component instantiation or behavioral. Let's see the architecture of the ***and_or*** entity in order to understand better the structural instantiation.


```
entity and_or is
port(
  a : in std_logic;
  b : in std_logic;
  d : in std_logic;
  e : in std_logic;
  g : out std_logic);
end and_or;
architecture and_or_a of and_or is
component and2 -- and2 component declaration
```

```
port(
  a : in std_logic;
  b : in std_logic;
  c : out std_logic);
end component;
component or2 -- or2 component declaration
port(
  a : in std_logic;
  b : in std_logic;
  c : out std_logic);
end component;
signal c : std_logic; -- wire used to connect
signal f : std_logic; -- the component
begin
-- and2 component instance
u1: and2
port map(
  a => a,
  b => b,
  c => c);
-- and2 component instance
```

```
u2: and2
port map(
  a => d,
  b => e,
  c => f);
-- or2 component instance
u3: or2
port map(
  a => c,
  b => f,
  c => g);
end and_or_a;
```


In the architecture declarative section we found:

- component declaration **AND2** and **OR2**
- signal declaration **c** and **f**

In the statement section, we found the **component instantiation**. Now pay attention to the **port mapping**.

- The port **a, b** and **c** mapped in the instance **u1** are respectively the entity input port **a** and **b,** the internal wire signal **c**.
- The port **a, b** and **c** mapped in the instance **u2** are respectively the entity input port **d** and **e,** the internal wire signal **f.**

The instance **u3** has the port **a** mapped on signal **c**, input port **b** on signal **f** and output port **c** on entity output port **g**.

As general rule,

the input/output port of an entity are signal inside the architecture.

The **input** port can be **only read**, the **output** port can be only **written**.

If you try to read from an output port or try to write to an input port the simulator rises an error and stops.