

UNIT – V

PRINCIPLES OF HDL

CONTENTS

- 1. Introduction to VHDL**
 - 1.1 VHDL Application
- 2. VHDL Program Structure**
 - 2.1 Entity Block
 - 2.2 Architecture Block
- 3. VHDL Operators**
- 4. Packages**
 - 4.1 Package Declaration
 - 4.2 Package Body
 - 4.3 Important Packages
- 5. Data Types**
- 6. Process**
- 7. Sequential Statements**
 - 7.1 Wait Statement
 - 7.2 Assertion Statement
 - 7.3 Report Statement
 - 7.4 Signal Assignment Statement
 - 7.5 Variable Assignment Statement
 - 7.6 Procedure Call Statement
 - 7.7 If Statement
 - 7.8 Case Statement
 - 7.9 Loop Statement
 - 7.10 Next Statement
 - 7.11 Exit Statement
 - 7.12 Return Statement
 - 7.13 Null Statement
- 8. Concurrent Statements**
 - 8.1 Block Statement

- 8.2 Generate Statement
- 9. Component**
 - 9.1 Component declaration
 - 9.2 Component Instantiation and interconnections
- 10. Functions**
- 11. Procedures**
- 12. Simulation**
- 13. Automated Testbench Generation**
 - 13.1 Off-line configuration
 - 13.2 On-line configuration
 - 13.3 Adaptive configuration
 - 13.4 Testbench Example
- 14. Introduction to Verilog**
- 15. VHDL/Verilog Comparison**

1.An Introduction to VHDL

VHDL is an acronym for **V**ery high speed integrated circuit (**VHSIC**) **H**ardware **D**escription **L**anguage which is a programming language that describes a logic circuit by function, data flow behavior, and/or structure. This hardware description is used to configure a programmable logic device (**PLD**), such as a field programmable gate array (**FPGA**), with a custom logic design.

The general format of a VHDL program is built around the concept of **BLOCKS** which are the basic building units of a VHDL design. Within these design blocks a logic circuit of function can be easily described.

A VHDL design begins with an **ENTITY** block that describes the interface for the design. The interface defines the input and output logic signals of the circuit being designed. The **ARCHITECTURE** block describes the internal operation of the design. Within these blocks are numerous other functional blocks used to build the design elements of the logic circuit being created.

After the design is created, it can be simulated and synthesized to check its logical operation. **SIMULATION** is a bare bones type of test to see if the basic logic works according to design and concept. **SYNTHESIS** allows timing factors and other influences of actual field programmable gate array (**FPGA**) devices to effect the simulation thereby doing a more thorough type of check before the design is committed to the **FPGA** or similar device.

1.1 VHDL Application

VHDL is used mainly for the development of Application Specific Integrated Circuits (ASICs). Tools for the automatic transformation of VHDL code into a gate-level net list were developed already at an early point of time. This transformation is called synthesis and is an integral part of current design flows.

For the use with Field Programmable Gate Arrays (FPGAs) several problems exist. In the first step, Boolean equations are derived from the VHDL description, no matter, whether an ASIC or a FPGA is the target technology. But now, this Boolean code has to be

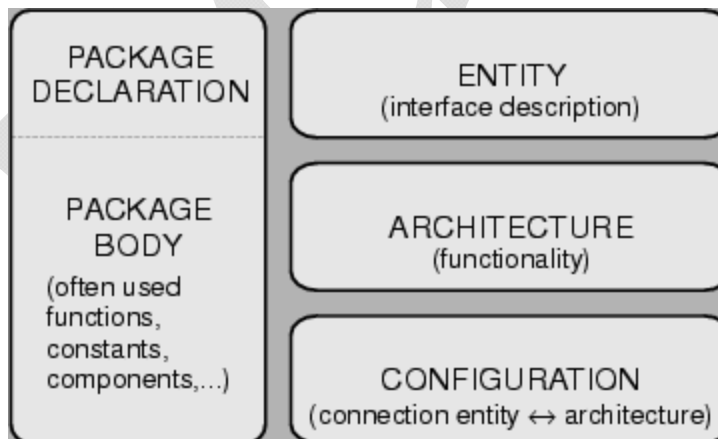
partitioned into the configurable logic blocks (CLB) of the FPGA. This is more difficult than the mapping onto an ASIC library. Another big problem is the routing of the CLBs as the available resources for interconnections are the bottleneck of current FPGAs.

While synthesis tools cope pretty well with complex designs, they obtain usually only suboptimal results. Therefore, VHDL is hardly used for the design of low complexity Programmable Logic Devices(PLDs).

VHDL can be applied to model system behavior independently from the target technology. This is either useful to provide standard solutions, e.g. for micro controllers, error correction (de-)coders, etc, or behavioral models of microprocessors and RAM devices are used to simulate a new device in its target environment.

An ongoing field of research is the hardware/software co design. The most interesting question is which part of the system should be implemented in software and which part in hardware. The decisive constraints are the costs and the resulting performance.

2 VHDL Program Structure



```
entity entity-name is  
    [port(interface-signal-declaration);]  
end [entity] [entity-name];
```

```
architecture architecture-name of entity-name is  
    [declarations]  
    begin  
        architecture body  
end [architecture] [architecture-name];
```

2.1 ENTITY BLOCK

An **entity block** is the beginning building block of a VHDL design. Each design has only one entity block which describes the interface signals into and out of the design unit. The syntax for an entity declaration is:

```
entity entity_name is  
    port (signal_name,signal_name : mode type;  
          signal_name,signal_name : mode type);  
end entity_name;
```

An entity block starts with the reserve word **entity** followed by the entity_name. Names and identifiers can contain letters, numbers, and the under score character, but must begin with an alpha character. Next is the reserved word **is** and then the **port** declarations. The indenting shown in the entity block syntax is used for documentation purposes only and is not required since VHDL is insensitive to white spaces.

A single **PORT** declaration is used to declare the interface signals for the entity and to assign **MODE** and data **TYPE** to them. If more than one signal of the same type is declared, each identifier name is separated by a **comma**. Identifiers are followed by a **colon** (:), mode and data type selections.

In general, there are five types of modes, but only three are frequently used. These three will be addressed here. They are **in**, **out**, and **inout** setting the signal flow direction for the ports as input, output, or bidirectional. Signal declarations of different mode or type are listed individually and separated by **semicolons** (;). The last signal declaration in a port statement and the port statement itself are terminated by a semicolon on the outside of the port's closing parenthesis.

The entity declaration is completed by using an **end** operator and the entity name. Optionally, you can also use an **end entity** statement. In VHDL, all statements are terminated by a semicolon.

Here is an example of an entity declaration for a set/reset (**SR**) latch:

```
entity latch is  
    port (s,r : in std_logic;  
        q,nq : out std_logic);  
end latch;
```

The set/reset latch has input control bits **s** and **r** which are defined as single input bits and output bits **q** and **nq**. Notice that the declaration does not define the operation yet, just the interfacing input and output logic signals of the design. A design circuit's operation will be defined in the architecture block.

We can define a **literal constant** to be used within an entity with the **generic** declaration, which is placed before the port declaration within the entity block. Generic literals can be used in port and other declarations. This makes it easier to modify or update designs. For instance if you declare a number of `bit_vector` bus signals, each eight bits in length, and at some future time you want to change them all to 16-bits, you would have to change each of the `bit_vector` range. However, by using a generic to define the range value, all you have to do is change the generic's value and the change will be reflected in each of the `bit_vectors` defined by that generic. The syntax to define a generic is:

```
generic (name : type := value);
```

The reserved word **generic** defines the declaration statement. This is followed by an identifier name for the generic and a colon. Next is the data type and a literal assignment value for the identifier. **:=** is the assignment operator that allows a literal value to be assigned to the generic identifier name. This operator is used for other assignment functions as we will see later.

For example, here is the code to define a bus width size using a generic literal.

entity my_processor is generic (busWidth : integer := 7);

Presently, busWidth has the literal value of 7. This makes the documentation more descriptive for a vector type in a port declaration:

**port(data_bus : in std_logic_vector (busWidth downto 0);
q-out : out std_logic_vector (busWidth downto 0));**

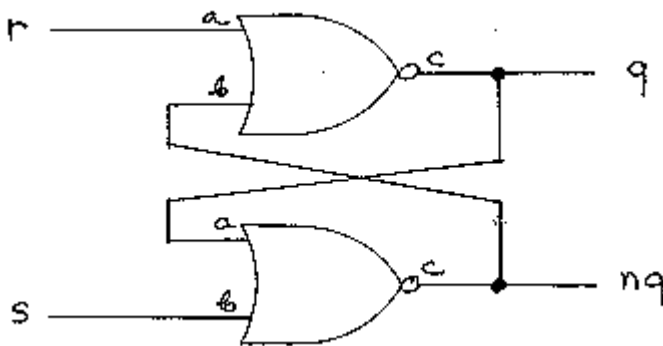
In this example, data_bus and q_out have a width of eight (8) bits (**7 down to 0**). When the design is updated to a larger bus size of sixteen (16) bits, the only change is to the literal assignment in the generic declaration from **7 to 15**.

2.2 ARCHITECTURE BLOCK

The **architecture block** defines how the entity operates. This may be described in many ways, two of which are most prevalent: **STRUCTURE** and **DATA FLOW or BEHAVIOR** formats. The **BEHAVIOR** approach describes the actual logic behavior of the circuit. This is generally in the form of a Boolean expression or process. The **STRUCTURE** approach defines how the entity is structured - what logic devices make up the circuit or design. The general syntax for the architecture block is:

```
architecture arch_name of entity_name is  
  declarations;  
  begin  
    statements defining operation;  
end arch_name;
```

example, we will use the set/reset NOR latch of figure 1. In VHDL code listings, -- (double dash) indicates a comment line used for documentation and ignored by the compiler.



```

library ieee;
use ieee.std_logic_1164.all;
    -- entity block
entity latch is
    -- interface signal declarations
    port (s,r : in std_logic;
        q,nq : out std_logic);
end latch;
    -- architecture block
architecture flipflop of latch is
begin
    -- assignment statements
    q <= r nor nq;
    nq <= s nor q;
end flipflop;

```

The first two lines imports the IEEE standard logic library **std_logic_1164** which contains predefined logic functions and data types such as **std_logic** and **std_logic_vector**. The use statement determines which portions of a library file to use. In this example we are selecting all of the items in the 1164 library. The next block is the entity block which declares the latch's interface inputs, **r** and **s** and outputs **q** and **nq**. This is followed by the architecture block which begins by identifying itself with the name **flipflop** as a description of entity **latch**.

Within the architecture block's body (designated by the **begin** reserved word) are two assignment statements. Signal assignment statements follow the general syntax of:

signal_identifier_name <= expression;

The **<=** symbol is the assignment operator for assigning a value to a signal. This differs from the **:=** assignment operator used to assign an initial literal value to generic identifier used earlier.

In our latch example, the state of the signal **q** is assigned the logic result of the **nor** function using input signals **r** and **nq**. The **nor** operator is defined in the IEEE **std_logic_1164** library as a standard VHDL function to perform the **nor** logic operation. Through the use of

Boolean expressions, the operation of the NOR latch's behavior is described and translated by a VHDL compiler into the hardware function appearing in figure 1.

3. VHDL Operators

Predefined VHDL operators can be grouped into seven classes:

1. binary logical operators: **and or nand nor xor xnor**

and logical and result is boolean,
nand logical complement of and result is boolean,
nor logical complement of or result is boolean,
xor logical exclusive or result is boolean,
xnor logical complement of exclusive or result is boolean,

2. relational operators:

= 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

3 shift operators:

sll shift left logical,
srl shift right logical,
sla shift left arithmetic,
sra shift right arithmetic,
rol rotate left,
rор rotate right,

4. adding operators:

+ addition, numeric + numeric, result numeric
- subtraction, numeric - numeric, result numeric
& concatenation, array or element & array or element,
result array

5. unary sign operators:

+ unary plus, + numeric, result numeric
- unary minus, - numeric, result numeric

6. multiplying operators:

* multiplication, numeric * numeric, result numeric
/ division, numeric / numeric, result numeric
mod modulo, integer **mod** integer, result integer
rem remainder, integer **rem** integer, result integer

7. miscellaneous operators:

abs absolute value, **abs** numeric, result numeric
not complement, **not** logic or boolean, result same
****** exponentiation, numeric ****** integer, result numeric

HIGHEST PRECEDENCE

Highest precedence first, left to right within same precedence Group, use parenthesis to control order. Unary operators take an operand on the right. Binary operators take an operand on the left and right.

Here are some examples of the use of VHDL operators

```
entity Operator_1 is end; architecture Behave of Operator_1 is  
begin process  
variable b : BOOLEAN; variable bt : BIT := '1'; variable i : INTEGER;  
variable pi : REAL := 3.14; variable epsilon : REAL := 0.01;  
variable bv4 : BIT_VECTOR (3 downto 0) := "0001";  
variable bv8 : BIT_VECTOR (0 to 7);  
begin  
b := "0000" < bv4; -- b is TRUE, "0000" treated as BIT_VECTOR.  
b := 'f' > 'g'; -- b is FALSE, 'dictionary' comparison.  
bt := '0' and bt; -- bt is '0', analyzer knows '0' is BIT.  
bv4 := not bv4; -- bv4 is now "1110".  
i := 1 + 2; -- Addition, must be compatible types.
```

```

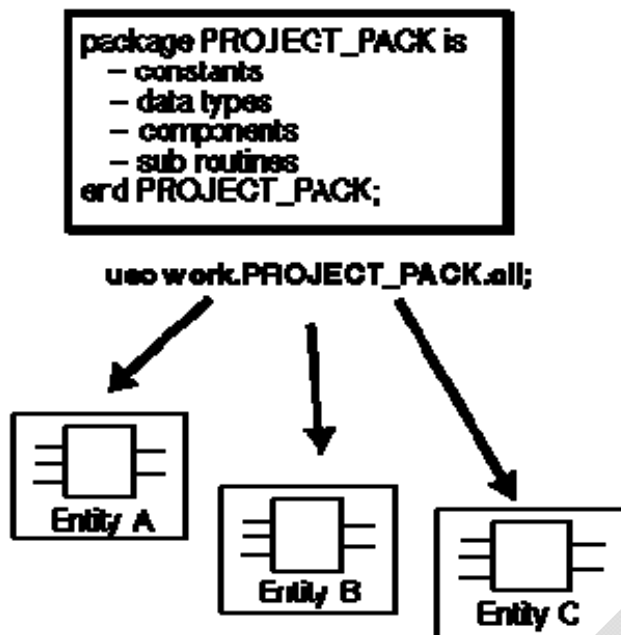
i := 2 ** 3;      -- Exponentiation, exponent must be integer.
i := 7/3;        -- Division, L/R rounded towards zero, i=2.
i := 12 rem 7;   -- Remainder, i=5. In general:
                  -- L rem R = L-((L/R)*R).
i := 12 mod 7;   -- modulus, i=5. In general:
                  -- L mod R = L-(R*N) for an integer N.
-- shift := sll | srl | sla | sra | rol | ror (VHDL-93 only)
bv4 := "1001" srl 2; -- Shift right logical, now bv4="0100".
-- Logical shift fills with T'LEFT.
bv4 := "1001" sra 2; -- Shift right arithmetic, now bv4="0111".
-- Arithmetic shift fills with element at end being vacated.
bv4 := "1001" ror 2; -- Rotate right, now bv4="0110".
-- Rotate wraps around.
-- Integer argument to any shift operator may be negative or zero.
if (pi*2.718)/2.718 = 3.14 then wait; end if; -- This is unreliable.
if (abs(((pi*2.718)/2.718)-3.14)<epsilon) then wait; end if; -- Better.
bv8 := bv8(1 to 7) & bv8(0); -- Concatenation, a left rotation.
wait; end process;
end;

```

4. Packages

A package is used as a collection of often used data types, components, functions, and so on. Once these objects are declared and defined in a package, they can be used by different VHDL design units. In particular, the definition of global information and important shared parameters in complex designs or within a project team is recommended to be done in packages.

It is possible to split a package into a declaration part and the so-called body. The advantage of this splitting is that after changing definitions in the package body only this part has to be recompiled and the rest of the design can be left untouched. Therefore, a lot of time consumed by compiling can be saved.



4.1 Package Declaration

As the name implies, a package declaration includes all globally used declarations of types, components, procedures and functions. A possible package declaration is presented by means of an example:

Example:

```

package MY_PACK is
type SPEED is (STOP, SLOW, MEDIUM, FAST);
component HA
port (I1, I2 : in bit; S, C : out bit);
end component;
constant DELAY_TIME : time;
function INT2BIT_VEC (INT_VALUE : integer)
return bit_vector;
end MY_PACK;

```

The name of this package is MY_PACK. It consists of different declarations, such as a type SPEED, a component HA, and so on. Attention should be paid to the declaration of the constant DELAY_TIME and the function INT2BIT_VEC which are declared but are not defined. Their definitions will be done in the package body but it would be possible to define the constant DELAY_TIME in the package declaration part as well. The definition of functions must be done in a package body.

4.2 Package Body

In the package body the definition of functions and procedures that were *only* declared in the package declaration must be specified. Constants which were declared only must get a value assigned to them in the package body.

The body of the package MY_PACK could be defined as:

Example:

```
package body MY_PACK is
  constant DELAY_TIME : time := 1.25 ns;
  function INT2BIT_VEC (INT_VALUE : integer)
    return bit_vector is
  begin
    -- sequential behavioral description (omitted here)
  end INT2BIT_VEC;
end MY_PACK;
```

The binding between the package declaration and the body is established by using the same name. In the above example it is the package name MY_PACK.

4.3 Important Packages

There are four important packages often used in VHDL descriptions.

STANDARD:

The package STANDARD is usually integrated directly in the simulation or synthesis program and, therefore, it does not exist as a VHDL description. It contains all basic types: boolean, bit, bit_vector, character, integer, and the like. Additional logical, comparison and arithmetic operators are defined for these types within the package.

The package STANDARD is a part of the STD library. Thus, it does not have to be explicitly included by the use statement.

TEXTIO:

The package TEXTIO contains procedures and functions which are needed to read from and write to text files.

This package is also a part of the library STD. It is *not* included in every VHDL description by default. Therefore, if required, it has to be included by the statement `use STD.TEXTIO.all;`

STD LOGIC 1164:

The `STD_LOGIC_1164` package has been developed and standardized by the IEEE. It introduces a special type called `std_ulogic` which has nine different logic values. The reason for this enhancement is that the type `bit` is not suitable for the precise modeling of digital circuits due to the missing values, such as uninitialized or high impedance.

The type `std_ulogic` consists of the following elements:

Declaration:

```
type std_ulogic is (  
    'U', -- uninitialized  
    'X', -- forcing unknown  
    '0', -- forcing 0  
    '1', -- forcing 1  
    'Z', -- high impedance  
    'W', -- weak unknown  
    'L', -- weak 0  
    'H', -- weak 1  
    '-' ); -- "don't care"
```

Besides this type used for modeling single wires other types are declared in the `STD_LOGIC_1164` package. Frequently used in descriptions of bus systems are the types `std_ulogic_vector` and `std_logic_vector`.

The use of the types `std_ulogic` and `std_logic` is strongly recommended. The package `STD_LOGIC_1164`, if it is available on the system installation, is usually kept in the logical library `IEEE`. It could be referenced with the two statements:

Syntax:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

STD LOGIC ARITH or NUMERIC STD:

Two additional packages, `STD_LOGIC_ARITH` (provided by SYNOPSYS) and `NUMERIC_STD` (provided by the IEEE), represent an additional part for the `STD_LOGIC_1164` package. They contain basic

arithmetic functions to enable calculations and comparisons based on the types `std_ulogic_vector` and `std_logic_vector`. These types represent buses - a bunch of signal lines - whose state can be interpreted as a binary or as a two's complement number. Therefore, it is necessary to specify which number representation is valid for a given bus system. This can be done by a conversion into the data types unsigned and signed. The appropriate conversion functions are also defined in these packages.

Example:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
architecture DETAILED of EXAMPLE is
    signal A, B : std_logic_vector (7 downto 0);
    signal SUM : std_logic_vector (8 downto 0);
    signal SUM_S : signed (8 downto 0);
    signal PROD : std_logic_vector (15 downto 0);
    signal PROD_S : signed (15 downto 0);
begin
    -- extension by one digit, conversion into a two's
    -- complement number and calculation of the sum:
    SUM_S <= signed(A(7) & A) + signed(B(7) & B);
    -- conversion to 9 bit std_logic_vector:
    SUM <= conv_std_logic_vector(SUM_S, 9);
    -- calculation of the product:
    PROD_S <= signed(A) * signed(B);
    -- conversion to 16 bit std_logic_vector:
    PROD <= conv_std_logic_vector(PROD_S, 16);
end DETAILED;
```

In the above example the sum and the product of the two buses A and B are calculated. Because the width of the resulted sum is the same as those of the operands, the width of A and B has to be extended by one bit in order to avoid an overflow. Since both A and B are two's complement numbers their MSB's have to be doubled. This is achieved by the concatenations `A(7) & A` and `B(7) & B`. After converting signals A and B with the `signed(...)` and adding, the result is assigned to a temporary signal `SUM_S`. This signal is then converted back to a 9 bit wide bus of the type `std_logic_vector` with the function `conv_std_logic_vector(SUM_S, 9)`. For the multiplication, the width of the result is 16 bit, which is equal to the sum of the widths of the

operands A and B. The appropriate information is required in the conversion of PROD_S to PROD.

5. Data Types

VHDL is a very **strongly** typed language. It does not allow a lot of intermixing of data types. The idea here is that since you are describing a piece of hardware, you need to keep things like signals and numbers separate. We shall start by looking at the different types of data that can be used with VHDL which include bits, buses, boolean, strings, real and integer number types, physical, and user defined enumerated types.

Defining Signals

There are two data types used for defining interfacing and interconnecting signals - **bits** and **bit_vectors**. The bit type defines a single binary bit type of signal like **RESET** or **ENABLE**. It is used anytime you need to define a single control or data line. For multiple bus signals, such as data or address buses, an array called a bit vector is used. Bit vectors require a range of bits to be defined and has the syntax: bit vector (range)

The range for a bit vector is defined from the least significant bit (**LSB**) to the most significant bit (**MSB**) and can be set to go from one to the other in ascending or descending order by using: **LSB to MSB** or **MSB downto LSB**. Here are some examples of bit vector forms:

```
addressbus(0 to 7);  
databus(15 downto 0);
```

The first defines an 8-bit address bus from addressbus(0) to addressbus(7). The second, a data bus from databus(15) downto databus(0).

The Boolean Type

The Boolean type has only two values: **TRUE (1)** and **FALSE (0)** and is usually used to hold the results of a comparison or the basis for conditional statement results.

Numerical Types

Number types that are usable in VHDL code are **INTEGERS** and **REALS**. Integers are signed numbers and reals are used for floating point

values. The range of values for both number types is somewhat dependent on the software application being used.

Subtyping

VHDL provides a method to create a version of an existing type with a specified range of values by using the **SUBTYPE** declaration. A typical example of the use and syntax of this operation is:

subtype SHORTINT is integer range 0 to 255; which creates an integer type, **SHORTINT** with a specified range of values from 0 to 255. This is NOT a new or enumerated (user) type which we shall describe next, but rather a modified existing type.

Enumerated or User Data Type

An enumerated data type provides a means for creating and defining user types. They are declared using the **TYPE** operator with syntax of:

TYPE type_name (type values);

Once the data type has been declared, then it can be used in a variable declaration (discussed later). For example, here is a declaration for a data type called **MONTHS**:

TYPE MONTHS (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);

A variable declared to be of type **MONTHS** can have anyone of the twelve values indicated in the parenthesis.

Other Data Types

VHDL specifications include additional data types that are used in the behavioral description of a circuit design. These types are:

- A. Arrays are single or multidimensional enumerated array types and the `std_logic_vector` type.
- B. An access type acts like a pointer type and has limited use.
- C. A file type is used to access a file.

D. A physical type is used to specify finite quantities such as time, voltage, etc. This type includes units of measure such as milliseconds (ms) and volts.

E. Time units used with the physical type are:

- primary unit is fs.....femtosecond
- ps = 1,000 fs.....picosecond
- ns = 1,000 ps.....nanosecond
- us = 1,000 ns.....microsecond
- ms = 1,000 us.....millisecond
- sec = 1,000 ms.....second
- min = 60 sec.....minute
- hour = 60 min.....hour

F. The line type is an ASCII string of characters.

G. A record contains a collection of multiple data types.

This has been a summary of the data types used by VHDL. As we progress, we will see how most are implemented in VHDL design code.

Example

architecture RTL of TRAFFIC_LIGHT is

```
type T_STATE is
  ( INIT,RED,REDYELLOW,GREEN,YELLOW );
signal STATE, NEXT_STATE : T_STATE;
```

```
signal COUNTER: integer;
constant END_RED   : integer := 10000;
constant END_GREEN : integer := 20000;
```

begin

```
LOGIC : process (STATE, COUNTER)
```

```
begin
```

```
  NEXT_STATE <= STATE;
```

```
  case STATE is
```

```
    when RED           =>
```

```
      if COUNTER = END_RED then
```

```
        NEXT_STATE <= REDYELLOW ;
```

```
      end if;
```

```
    when REDYELLOW   => -- statements
```

```

        when GREEN           => -- statements
    when YELLOW           => -- statements
        when INIT           => -- statements
    end case;
end process LOGIC;
end RTL;

```

6. PROCESS

Statements within architecture blocks, to this point, are executed **concurrently** - that is at the same time. Also, there is no way to synchronize their execution with clocking or any other kind of signals. To incorporate sequential statement execution and some manner of synchronization, we need to use a **PROCESS** block whose general syntax form is:

```

process_name : process (sensitivity list)
    variable variable_names : variable_type;
    begin
        statements;
    end process;

```

EXAMPLE:

```

entity AND_OR_XOR is
    port (A,B : in bit;
          Z_OR, Z_AND, Z_XOR : out bit);
end AND_OR_XOR;

```

```

architecture RTL of AND_OR_XOR is
begin

```

```

    A_O_X: process (A, B) ← sensitivity list
    begin
        Z_OR <= A or B;
        Z_AND <= A and B;
        Z_XOR <= A xor B;
    end process A_O_X ;

```

```

end RTL;

```

Process statements are placed in the **architecture block** of our design. The `process_name` and variable declarations are optional. Process names are

handy if your design contains more than one process. **Variable** declarations are used to define a variable local to and used by the process. Variable declarations are added in the declaration area preceding the body of the process block. In contrast to a signal, variable declarations define memory locations, identified by variable identifier names, used to store results of expressions. Signals, by their nature, cannot be used to perform arithmetic manipulations such as incrementing or decrementing their value while variables can be operated on mathematically. The variable assignment operator is := which is the same one used for assigning initial literal values. The syntax for a variable assignment is:

variable_identifier := expression;

To evaluate expressions used in a variable declaration or process block, we must become familiar with the operators used by VHDL. Many of them are not strangers to anyone who has any kind of programming experience. In order of their precedence, they are:

Highest

() - parenthesis

** - exponential

abs - absolute unsigned magnitude numbers

not - inversion

* - multiplication

/ - division

mod - modulo or quotient from division

rem - remainder result of division

+ - identity

- - negation

+ - addition

- - subtraction

& - concatenation

sll - shift left logical

srl - shift right logical

sla - shift left arithmetic

sra - shift right arithmetic

rol - rotate left

ror - rotate right

= - equality

/= - not equal

< - less than

> - greater than

<= - less than or equal

>= - greater than or equal

LOWEST

and - logic and

or - logic or

nand - logic nand

nor - logic nor

xor - logic exclusive or

xnor - logic exclusive nor

Now an example of a variable assignment:

cnt := cnt + 1;

As with any other language, the expression on the right is evaluated first. In this case one is added to the variable `cnt`. The results are then stored back into the `cnt` variable indicated on the left side of the assignment statement. This one simply increments `cnt` by 1. To set this variable statement into a process block, the code would look like:

```
count : process(x)  
  variable cnt : integer := -1;  
  begin  
    cnt := cnt + 1;  
  end process;
```

The first line of the process syntax is its declaration and contains an optional parameter list, known as the **sensitivity list**. A process executes once at the beginning of a simulation and any time that an event occurs on an item in the sensitivity list. An **EVENT** is any change of state of a signal. A change of state on signal `x` will cause this process to execute once.

The next line is a variable declaration that is similar to a port (signal) declaration. Since it is a variable and not a port, there is no mode selection. Also, variables can be assigned an initial value using an assignment operator as shown in the example. We want `cnt` to start at 0, but since the process executes once upon starting simulation (without an event occurring on `x`), we need to initialize `cnt` to -1. The initial execution of the process due to the start of a simulation will set `cnt` to 0 by incrementing it once. After that, each time an event occurs on `x`, `cnt` will be incremented once, thus keeping track of how many times `x` changes state. The statements to be executed by the process body follow the **begin** reserved word. Finally, the process declaration is completed using an end process statement.

Declarations within the process block and preceding the process body are executed *only once* - when simulation is initiated. Thereafter, when the process is run due to an event on one of the signals on the sensitivity list, *only the body* of the process is executed. This prevents variables from being re-initialized each time the process is run.

All statements in a process execute sequentially. Here are a couple of examples of process statements with an analysis of each:

```
process ( Y )  
variable X, Z : std_logic;  
begin
```

```
X := Y;  
Z := not X;  
end process;
```

This is a fairly easy appearing example, but let's take some time exploring what happens to make sure you fully grasp the difference between concurrent and sequential operation. **Y** is included in the sensitivity list, so it must have been declared in the design before the process statement. Variables **X** and **Z** are declared in the process block forcing these variables to be local to the process and not accessible outside of it.

To follow what happens when the process is executed, let's assume some initial values for our three variables:

- Y = 1
- X = 1
- Z = 0

Initial values for variables can be set in the variable declaration statements using the: = assignment operator in this manner:

```
variable X : std_logic := 1;  
variable Z : std_logic := 0;
```

Of course, signal **Y** would have to be initialized before the process statement to give it a beginning state. In this case, you would probably use an assignment statement:

```
Y <= '1';
```

Since **Y** has been defined as an interface signal in an entity, the <= assignment operator is required here. Assigning a literal logic state, 1 or 0, to a signal requires a **single quote** around the 1 or 0. This causes the software to convert the ASCII 1 or 0 to a logic state and assign it to the signal. Assigning a string of logic bit literals to a vector requires double quotes so that the ASCII string can be converted to logic states for each bit of the vector. Numerical literals will not use the quotes around it.

The sample states were not selected as randomly as you might think. I chose them to illustrate the point of **sequential** operation within the process. When **Y** changes to a 0 through some outside influence, an event occurs and the process is initiated. If the statements within the process were executed

concurrently, they would use the *initial* values to produce results for all outputs. The change in **Y** from 1 to 0 causes **X** to change to a 0 because of the statement **X := Y**; Because **X** had a value of 1 initially, this value is used for the second statement in concurrent execution. This forces **Z** to become 1 from the statement **Z := not X**;

However, the statements in the process are executed **sequentially** rather than concurrently. What actually occurs in the process is **X** becomes 0 when **Y** changes to 0 as it did for a concurrent execution. However, this time, **Z** would become 1 since the second statement in a sequential execution would use the new value of **X** instead of **X**'s initial value.

Now to a more practical example use of a process which will also include a method to prevent statements within the process body from executing when simulation is first begun and an event has not yet occurred:

```
library ieee;
use ieee.std_logic_1164.all;
entity DFF is
-- Signals are initialized to 0 by default.
-- To make QN a 1, it has to be initialized
port ( D, CLK : in std_logic;
       Q : out std_logic;
       QN : out std_logic := '1');
end DFF;
architecture data_flip of DFF is
begin
  process ( CLK )
  begin
    if (CLK = '1' and CLK'event ) then
      Q <= D after 10ns;
      QN <= not D after 10ns;
    end if;
  end process;
end data_flip;
```

There is a lot going on in this short design, so let's examine it carefully. The only wrinkle in the entity block is covered by the **comment lines** which are always preceded by a **double dash** (--). Identifiers of all kinds are usually initialized by most compilers when they are declared, to 0 or null. To set **QN** to the opposite state of **Q** initially, we had to assign it an

initially value of 1 by using: = '1' following its port declaration. The rest of the entity block is straight forward.

In the architecture block we did not require any local variables or signals so none are declared. The process block contains one signal in the sensitivity list, **CLK**. The only statement in the process body is an **if..then..else** statement. The if..then..else statement which is explored in more detail later, has a standard format of:

```
if condition then  
statements;  
else  
statements;  
end if;
```

The **else block** is optional and is used when there are statements to be executed when the conditional test returns a **false result**. The **then** statements are executed when the condition rings **true**.

The **if..then..else** statement in the example has two conditions and both have to be met to execute the statements within the then block. The first condition requires the state of **CLK** to be *high*. The and operator in the condition field forces a second condition to also be true. This condition is **CLK'event** which says that an *event* must have occurred on **CLK** to be true. What this format really accesses is a property of the process object called an event. If the event occurred, **CLK'event** returns **true**. If no event occurred, it returns a **false** value. The inclusion of this condition eliminates the execution of the statements within the if block when simulation first begins since the lack of a **CLK** event causes **CLK'event** to be **false**. The only time the if condition will be satisfied is when an event on **CLK** occurred. Additionally, **CLK** has to be high, so this combination causes the then statements to be executed only on a *positive transition (edge)* of the **CLK** signal.

By now, you should notice some significant difference in declaring and initializing integers and signals. The: = operator is used to assign initial values in a variable statement. Notice that for signals, **single quotes** are required around the initial value ('0') while none are used for an integer (0). This is because signal values are logic states and integer values are numerical. Numerical values do not require quotes.

Also notice the difference when integer variables are assigned a value from an expression compared to a signal assignment. In a previous example we used **Y <= A and B;** to assign to **Y** the results of **A** and **B**. In this most recent example, we did an arithmetic operation on an integer value and assigned the results to it: **cnt := cnt + 1;** It is very easy to use the incorrect assignment symbol (**:= or <=**) since they look so similar.

7. VHDL Sequential Statements

There are several statements that may only be used in the body of a process. These statements are called *sequential statements* because they are executed sequentially. That is, one after the other as they appear in the design from the top of the process body to the bottom. In this section we will examine some of these statements

Sequential Statements

- wait statement
- assertion statement
- report statement
- signal assignment statement
- variable assignment statement
- procedure call statement
- if statement
- case statement
- loop statement
- next statement
- exit statement
- return statement
- null statement

7.1 wait statement

Cause execution of sequential statements to wait.

[label:] **wait** [sensitivity clause] [condition clause] ;

wait for 10 ns;

-- timeout clause, specific time delay.

wait until clk='1'; -- condition clause, Boolean condition
wait until A>B **and** S1 **or** S2; -- condition clause, Boolean condition
wait on sig1, sig2; -- sensitivity clause, any event on any
-- signal terminates wait

```

entity FF is
  port (D, CLK : in bit;
        Q : out bit);
end FF;

```

```

architecture BEH_1 of FF is
begin
  process
  begin
    wait on CLK;
    if (CLK = '1') then
      Q <= D;
    end if;
  end process;
end BEH_1;

```

7.2 assertion statement

Used for internal consistency check or error message generation.

[label:] **assert** boolean_condition [**report** string] [**severity** name] ;

```

assert a=(b or c);
assert j<i report "internal error, tell someone";
assert clk='1' report "clock not up" severity WARNING;

```

Predefined severity names are: NOTE, WARNING, ERROR, FAILURE . Default severity for assert is ERROR.

7.3 Report statement

Used to output messages.

[label:] **report** string [**severity** name] ;

```
report "finished pass1"; -- default severity name is NOTE
report "Inconsistent data." severity FAILURE;
```

7.4 signal assignment statement

The signal assignment statement is typically considered a concurrent statement rather than a sequential statement. It can be used as a sequential statement but has the side effect of obeying the general rules for when the target actually gets updated.

In particular, a signal can not be declared within a process or subprogram but must be declared in some other appropriate scope. Thus the target is updated in the scope where the target is declared when the sequential code reaches its end or encounters a 'wait' or other event that triggers the update.

```
[ label: ] target <= [ delay_mechanism ] waveform ;
```

delay_mechanism

transport

reject time_expression

inertial

waveform

waveform_element [, waveform_element]

unaffected

waveform_element

value_expression [**after** time_expression]

null [**after** time_expression]

```
sig1 <= sig2;
```

```
Sig <= Sa and Sb or Sc nand Sd nor Se xor Sf xnor Sg;
```

```
sig1 <= sig2 after 10 ns;
```

```
clk <= '1' , '0' after TimePeriod/2 ;
```

```
sig3 <= transport sig4 after 3 ns;
```

```
sig4 <= reject 2 ns sig5 after 3 ns; -- increasing time order
```

```
sig6 <= inertial '1' after 2 ns, '0' after 3 ns , '1' after 7 ns;
```

Note: omitting [**after** time expression] is equivalent to **after** 0 fs;

7.5 Variable assignment statement

Variables can only be defined in a process and they are only accessible within this process. Variables and signals show a fundamentally different behavior. In a process, the last signal assignment to a signal is carried out when the process execution is suspended. Value assignments to variables, however, are carried out immediately. To distinguish between a signal and a variable assignment different symbols are used: ' \leq ' indicates a signal assignment and ' $:=$ ' indicates a variable assignment.

[label:] target := expression ;

```
architecture RTL of XYZ is
  signal A, B, C : integer range 0 to 7;
  signal Y, Z :   integer range 0 to 15;
begin
  process (A, B, C)
    variable M, N : integer range 0 to 7;
  begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
  end process;
end RTL;
```

7.6 Procedure call statement

Call a procedure.

[label:] procedure-name [(actual parameters)] ;

do_it; -- no actual parameters

compute(stuff, A=>a, B=>c+d); -- positional association first,

-- then named association of
-- formal parameters to actual parameters

7.7 if statement

The if condition must evaluate to a boolean value ('true' or 'false'). After the first if condition, any number of elsif conditions may follow. Overlaps may occur within different conditions. An else branch, which combines all cases that have not been covered before, can optionally be inserted last. The if statement is terminated with 'end if'.

The first if condition has top priority: if this condition is fulfilled, the corresponding statements will be carried out and the rest of the 'if - end if' block will be skipped.

```
if CONDITION then  
  -- sequential statements  
end if;
```

```
if CONDITION then  
  -- sequential statements  
else  
  -- sequential statements  
end if;
```

```
if CONDITION then  
  -- sequential statements  
elsif CONDITION then  
  -- sequential statements  
  . . .  
else  
  -- sequential statements  
end if;
```

```
entity IF_STATEMENT is  
  port (A, B, C, X : in bit_vector (3 downto 0);  
        Z           : out bit_vector (3 downto 0);  
  end IF_STATEMENT;
```

```

architecture EXAMPLE1 of IF_STATEMENT is
begin
  process (A, B, C, X)
  begin
    Z <= A;
    if (X = "1111") then
      Z <= B;
    elsif (X > "1000") then
      Z <= C;
    end if;
  end process;
end EXAMPLE1

```

7.8 case statement

While the priority of each branch is set by means of the query's order in the IF case, all branches are equal in priority when using a CASE statement. Therefore it is obvious that there must not be any overlaps. On the other hand, all possible values of the CASE EXPRESSION must be covered. For covering all remaining, i.e. not yet covered, cases, the keyword ' others ' may be used.

The type of the EXPRESSION in the head of the CASE statement has to match the type of the query values. Single values of EXPRESSION can be grouped together with the '|' symbol, if the consecutive action is the same. Value ranges allow to cover even more choice options with relatively simple VHDL code.

Ranges can be defined for data types with a fixed order, only, e.g. user defined enumerated types or integer values. This way, it can be decided whether one value is less than, equal to or greater than another value. For ARRAY types (e.g. a BIT_VECTOR) there is no such order, i.e. the range "0000" TO "0100" is undefined and therefore not admissible.

case EXPRESSION is

when VALUE_1 =>

-- sequential statements

when VALUE_2 | VALUE_3 =>

-- sequential statements

when VALUE_4 to VALUE_N =>

```

-- sequential statements

when others =>
-- sequential statements

end case ;

entity CASE_STATEMENT is
  port (A, B, C, X : in integer range 0 to 15;
        Z          : out integer range 0 to 15;
  end CASE_STATEMENT;

architecture EXAMPLE of CASE_STATEMENT is
begin
  process (A, B, C, X)
  begin
    case X is
      when 0 =>
        Z <= A;
      when 7 | 9 =>
        Z <= B;
      when 1 to 5 =>
        Z <= C;
      when others =>
        Z <= 0;
    end case;
  end process;
end EXAMPLE;

```

7.9 loop statement

The loop label is optional. By defining the range the direction as well as the possible values of the loop variable are fixed. The loop variable is only accessible within the loop.

For synthesis the loop range has to be locally static and must not depend on signal or variable values. While loops are not generally synthesizable.

Three kinds of iteration statements.

```
[ label: ] loop  
    sequence-of-statements -- use exit statement to get out  
end loop [ label ] ;
```

```
[ label: ] for variable in range loop  
    sequence-of-statements  
end loop [ label ] ;
```

```
[ label: ] while condition loop  
    sequence-of-statements  
end loop [ label ] ;
```

```
loop  
    input_something;  
    exit when end_file;  
end loop;
```

example

```
entity CONV_INT is  
    port (VECTOR: in  bit_vector(7 downto 0);  
          RESULT: out integer);  
end CONV_INT;
```

```
architecture A of CONV_INT is  
begin  
    process(VECTOR)  
        variable TMP: integer;
```

```
begin  
    TMP := 0;
```

```
    for I in 7 downto 0 loop  
        if (VECTOR(I)='1') then  
            TMP := TMP + 2**I;  
        end if;  
    end loop;
```

```
RESULT <= TMP;
```

```
end process;  
end A;
```

architecture C of CONV_INT is

```
begin  
  process(VECTOR)  
    variable TMP: integer;  
    variable I : integer;  
  begin  
    TMP := 0;  
    I := VECTOR' high;  
    while (I >= VECTOR' low) loop  
      if (VECTOR(I)='1') then  
        TMP := TMP + 2**I;  
      end if;  
      I := I - 1;  
    end loop;  
    RESULT <= TMP;  
  end process;  
end C;
```

7.10 next statement

A statement that may be used in a loop to cause the next iteration.

```
[ label: ] next [ label2 ] [ when condition ] ;
```

```
next;
```

```
next outer_loop;
```

```
next when A>B;
```

```
next this_loop when C=D or done; -- done is a Boolean variable
```

7.11 exit statement

A statement that may be used in a loop to immediately exit the loop.

```
[ label: ] exit [ label2 ] [ when condition ] ;
```

```
exit;
```

```
exit outer_loop;
```

```
exit when A>B;
```

exit this_loop **when** C=D **or** done; -- done is a Boolean variable

7.12 return statement

Required statement in a function, optional in a procedure.

[label:] **return** [expression] ;

return; -- from somewhere in a procedure
return a+b; -- returned value in a function

7.13 null statement

Used when a statement is needed but there is nothing to do.

[label:] **null** ;

null;

8. VHDL Concurrent Statements

All statements within architectures are executed concurrently. While it is possible to use VHDL processes as the only concurrent statement, the necessary overhead (process, begin, end, sensitivity list) lets designer look for alternatives when the sequential behavior of processes is not needed.

The signal assignment statement was the first VHDL statement to be introduced. The signal on the left side of the assignment operator '<=' receives a new value whenever a signal on the right side changes. The new value stems from another signal in the simplest case (i.e. when an intermediate signal is necessary to match different port modes) or can be calculated from a number of signals.

Concurrent Statements

- block statement
- process statement
- concurrent procedure call statement
- concurrent assertion statement
- concurrent signal assignment statement

- conditional signal assignment statement
- selected signal assignment statement
- component instantiation statement
- generate statement

8.1 block statement

The sub modules in an architecture body can be described as blocks. A block is a unit of module structure, with its own interface, connected to other blocks or ports by signals. A block is specified using the syntax:

```

block_statement ::=
block_label :
block [ ( guard_expression ) ]
block_header
block_declarative_part
begin
block_statement_part
end block [ block_label ] ;
block_header ::=
[ generic_clause
[ generic_map_aspect ; ] ]
[ port_clause
[ port_map_aspect ; ] ]
generic_map_aspect ::= generic map ( generic_association_list )
port_map_aspect ::= port map ( port_association_list )
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }

```

The guard expression is not covered in this booklet, and may be omitted. The block header defines the interface to the block in much the same way as an entity header defines the interface to an entity. The generic association list specifies values for the generic constants, evaluated in the context of the enclosing block or architecture body. The port map association list specifies which actual signals or ports from the enclosing block or architecture body are connected to the block's ports. Note that a block statement part may also contain block statements, so a design can be composed of a hierarchy of blocks, with behavioral descriptions at the bottom level of the hierarchy.

The control unit block has ports `clk`, `bus control` and `bus ready`, which are connected to the processor entity ports. It also has an output port for controlling the data path, which is connected to a signal declared in the architecture. That signal is also connected to a control port on the data

path block. The address and data ports of the data path block are connected to the corresponding entity ports. The advantage of this modular decomposition is that each of the blocks can then be developed independently, with the only effects on other blocks being well defined through their interfaces.

As an example, suppose we want to describe a structural architecture of the processor entity .

```
architecture block_structure of processor is  
type data_path_control is ... ;  
signal internal_control : data_path_control;  
begin  
  control_unit : block  
  port (clk : in bit;  
        bus_control : out proc_control;  
        bus_ready : in bit;  
        control : out data_path_control);  
  port map (clk => clock,  
            bus_control => control, bus_ready => ready;  
            control => internal_control);  
  declarations for control_unit  
    begin  
      statements for control_unit  
    end block control_unit;  
  data_path : block  
  port (address : out integer;  
        data : inout word_32;  
        control : in data_path_control);  
  port map (address => address, data => data,  
            control => internal_control);  
  declarations for data_path  
    begin  
      statements for data_path  
    end block data_path;  
end block_structure;
```

8.2 GENERATE

VHDL has an additional concurrent statement which can be used in architecture bodies to describe regular structures, such as arrays of blocks,

component instances or processes. The syntax is:

```
generate_statement ::=
```

```
generate_label :  
generation_scheme generate  
{ concurrent_statement }  
end generate [ generate_label ] ;
```

```
generation_scheme ::=  
for generate_parameter_specification  
if condition
```

The for generation scheme describes structures which have a repeating pattern. The if generation scheme is usually used to handle exception cases within the structure, such as occur at the boundaries. This is best illustrated by example.

Suppose we want to describe the structure of an adder

```
for i in 0 to width-1 generate  
ls_bit : if i = 0 generate  
ls_cell : half_adder port map (a(0), b(0), sum(0), c_in(1));  
end generate lsbit;  
middle_bit : if i > 0 and i < width-1 generate  
middle_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), c_in(i+1));  
end generate middle_bit;  
ms_bit : if i = width-1 generate  
ms_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), carry);  
end generate ms_bit;  
end generate adder;
```

Adder constructed out of full-adder cells, with the exception of the least significant bit, which consists of a half-adder.

The outer generate statement iterates with *i* taking on values from 0 to Width-1. For the least significant bit (*i*=0), an instance of a half adder component is generated. The input bits are connected to the least significant bits of *a* and *b*, the output bit is connected to the least significant bit of *sum*, and the carry bit is connected to the carry in of the next stage. For intermediate bits, an instance of a full adder component is generated with inputs and outputs connected similarly to the first stage. For the most

significant bit ($i=\text{width}-1$), an instance of the half adder is also generated, but its carry output bit is connected to the signal carry.

9. COMPONENT

A structural way of modeling describes a circuit in terms of components and its interconnection. Each component is supposed to be defined earlier (e.g. in package) and can be described as structural, a behavioral or dataflow model. At the lowest hierarchy each component is described as a behavioral model, using the basic logic operators defined in VHDL. In general structural modeling is very good to describe complex digital systems, though a set of components in a *hierarchical* fashion.

A structural description can best be compared to a schematic block diagram that can be described by the components and the interconnections. VHDL provides a formal way to do this by

- Declare a list of components being used
- Declare signals which define the nets that interconnect components
- Label multiple instances of the same component so that each instance is uniquely defined.

The components and signals are declared within the architecture body,

```
architecture architecture_name of NAME_OF_ENTITY is  
  -- Declarations  
    component declarations  
    signal declarations  
  begin  
    -- Statements  
    component instantiation and connections  
    :  
  end architecture_name;
```

9.1 Component declaration

Before components can be instantiated they need to be declared in the architecture declaration section or in the package declaration. The component declaration consists of the component name and the interface (ports). The syntax is as follows:

```

component component_name [is]
    [port (port_signal_names: mode type;
           port_signal_names: mode type;
           :
           port_signal_names: mode type);]
end component [component_name];

```

The component name refers to either the name of an entity defined in a library or an entity explicitly defined in the VHDL file (see example of the four bit adder).

The list of interface ports gives the name, mode and type of each port, similarly as is done in the entity declaration.

A few examples of component declaration follow:

```

component OR2
    port (in1, in2: in std_logic;
           out1: out std_logic);
end component;

```

```

component PROC
    port (CLK, RST, RW, STP: in std_logic;
           ADDRBUS: out std_logic_vector (31 downto 0);
           DATA: inout integer range 0 to 1024);

```

```

component FULLADDER
    port(a, b, c: in std_logic;
           sum, carry: out std_logic);
end component;

```

As mentioned earlier, the component declaration has to be done either in the architecture body or in the package declaration. If the component is declared in a package, one does not have to declare it again in the architecture body as long as one uses the **library** and **use** clause.

9.2 Component Instantiation and interconnections

The component instantiation statement references a component that can be

- Previously defined at the current level of the hierarchy or
- Defined in a technology library (vendor's library).

The syntax for the components instantiation is as follows,

```
instance_name : component name
  port map (port1=>signal1, port2=> signal2,... port3=>signaln);
```

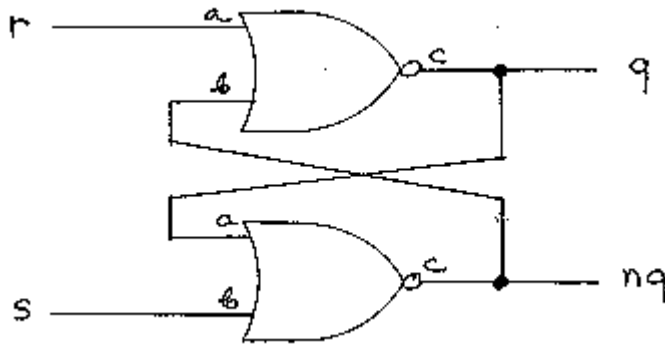
The instance name or label can be any legal identifier and is the name of this particular instance. The component name is the name of the component declared earlier using the component declaration statement. The port name is the name of the port and signal is the name of the signal to which the specific port is connected. The above port map associates the ports to the signals through named association. An alternative method is the positional association shown below,

```
port map (signal1, signal2,...signaln);
```

in which the first port in the component declaration corresponds to the first signal, the second port to the second signal, etc. The signal position must be in the same order as the declared component's ports. One can mix named and positional associations as long as one puts all positional associations before the named ones. The following examples illustrates this,

```
component NAND2
  port (in1, in2: in std_logic;
        out1: out std_logic);
end component;
signal int1, int2, int3: std_logic;
architecture struct of EXAMPLE is
  U1: NAND2 port map (A,B,int1);
  U2: NAND2 port map (in2=>C, in2=>D, out1=>int2);
  U3: NAND3 port map (in1=>int1, int2, Z);
  ....
```

For an example, we will use the following NOR latch which is similar to one in figure .



And here is the entity and architecture blocks of the structure design:

```

library IEEE;
use IEEE.std_logic_1164.all;

-- NOR gate entity design

entity nor_gate is
  port (a,b : in std_logic;
              c : out std_logic);
end nor_gate;
architecture my_nor of nor_gate is
begin
  c <= a nor b;
end my_nor;

-- begin latch design

entity latch is
  port (s,r : in std_logic;
              q,nq : out std_logic);
end latch;
architecture flipflop of latch is

-- NOR gate component declaration

  component nor_gate
    port (a,b : in std_logic; c out std_logic);
  end component;
begin

```

-- instantiation of two NOR gates

```
n1 : nor_gate  
  port map (r, nq, q);  
n2 : nor_gate  
  port map (s, q, nq);  
end flipflop;
```

A number of concepts are illustrated by this example, so let's explore each of them. First, we have the component declaration placed in the architecture block preceding the architecture body. It starts with the reserve word **component** followed by the component's name, in this case - nor gate. As with an entity, signals for the component are declared using the **port** function. In this example we have two inputs, **a** and **b** and one output, **c**. An end component statement completes the declaration. This creates a component **OBJECT**. Notice that the nor gate component and its corresponding entity declarations are identical except for the component and entity reserved words. The importance of this, is that we will declare two instances of this object, each of which will **inherit the properties** of the nor gate component object.

The first instance is **n1** and notice that its declaration is placed in the body of the architecture block. This means that the architecture wants to use this instance for the latch. In order to *inherit* nor_gate properties, the signals used by the instance must be **MAPPED** from the nor_gate object. This is accomplished using the **port map** statement. The signals mapped must be in the same order as the component object. In this case **r** and **nq** inherit the input function from **a** and **b** of the nor_gate object. **q** gets the output function from **c** in the nor_gate component.

In this simple example, there are no extra interconnecting signals. All connections are established using signals of the NOR gate components. Here is a little different design that incorporates interconnecting signals between components.

10. FUNCTIONS

A **function** in VHDL is similar to functions in most upper level languages. It is a subprogram that accepts input parameters and returns a single result. The syntax for a function declaration is:

```
function function_name ( formal parameters )  
    return return_type is  
    variable declarations;  
    begin  
        statements;  
    return return_variable_name;  
end function_name;
```

Here is an example of a function that computes the parity on a data array of unspecified length:

```
function parity ( word : std_logic_vector )  
    return std_logic is  
    variable tmp : std_logic;  
    begin  
    for i in word'range loop  
        tmp := tmp xor word(i);  
    end loop;  
    return tmp;  
end parity;
```

When the function is called, a specific variable of `std_logic_vector` type is passed to it. A `tmp` variable of `std_logic` type is declared and, by default, is initialized to zero. The for loop exclusive ORs `tmp` with each bit of the word array. The loop repeats from 1 to the number of bits in the array (`word'length`). The final value of `tmp` (the even parity state of the word passed in) is passed out of the function via the return statement.

Function Call

Functions are called using a single assignment statement that has the general form of:

```
variable <= function_name ( actual parameters );
```

The actual parameter(s) passed to the function must be of the same type and length (if a specified array size was made) as the formal parameter(s) in the function declaration. The returned value of the function is stored into the variable on the left side of the expression.

Function call example:

```
even_parity <= parity( data_bus_in );
```

This function call accepts the vector array `data_bus_in` and calculates its parity, returning the even parity result to the variable `even_parity`.

12. PROCEDURES

A **procedure** like a function is a subprogram that must first be declared and then called. Unlike the function, procedures can pass out numerous results through its parameter list. Because of this, parameter declarations must include an in or out mode declaration as well as a data type indication. The syntax for declaring a procedure is:

```
procedure procedure_name (formal parameter : mode type,  
formal parameter : mode type ) is  
variable declaration;  
begin  
statements;  
end procedure_name;
```

Since there are possible multiple results, procedures are not called using an assignment statement like a function. Instead they are called using this format:

```
procedure_name (actual parameter list);
```

The **parameter list** contains the names of the actual parameters to be passed in and out of the procedure and while they do not have to have the same identifier names as those in the declaration, they must follow the exact order as well as having the same mode and type as the formal parameters in the procedure declaration. The following procedure returns two results **Z** and **ZCOMP** based on the values of words **A** and **B** and the type of operation to be performed on them.

```

type op_code is ( ADD, SUB, MUL, DIV, LT, LE, EQ );
.
.
procedure ARITH_UNIT ( A, B : in integer; op : in op_code;
Z : out integer; ZCOMP : out boolean ) is
begin
    case op is
when ADD => Z := A + B;
when SUB => Z := A - B;
when MUL => Z := A * B;
when DIV => Z := A / B;
when LT => ZCOMP := A < B;
when LE => ZCOMP := A <= B;
when EQ => ZCOMP := A = B;
when others => Z := Z;
    end case;
end ARITH_UNIT;

```

Somewhere, preceding the procedure declaration is an enumerated user type declaration creating a new type `op_code` with the values indicated in the parenthesis. The procedure `ARITH_UNIT` is declared with three input parameters passed into the procedure - `A` and `B` which is integer types and `op` which is an `op_code` type. Two output parameters, integer `Z` and Boolean `ZCOMP` contain the results passed out by the procedure. In the body of the procedure, `op` is checked for a value using a case statement and dependent on that result, one of the when assignments are performed. An example statement that calls this procedure is:

```

ARITH_UNIT (word1, word2, operation, result, comp_check );

```

Word1 and word2 are passed in as parameters `A` and `B`. operation enters as parameter `op`. The results of the procedure are passed out through parameters `Z` and `ZCOMP` to variables `result` and `comp_check`.

13. Simulation

After the successful analysis of VHDL models, their simulation could be performed to verify the correct functionality. For this purpose, the elements in the lowest hierarchy level must be available as behavioral

descriptions. Starting point of the simulation is the analyzed configuration declaration of a test bench or the top-level module.

Before the actual simulation takes place, the following two steps are executed (without the interaction between the circuit developer and the simulation tool):

1. Elaboration phase: The most important part of this step is assembling the hierarchy. This is where all entity-architecture pairs are built as specified by the configurations. This is similar to the activities taking place during linking in other programming languages like C. Furthermore, memory is allocated for signals, variables and constants, and their values are initialized as specified.
2. Initialization phase: All processes are executed once until they get suspended by the first encountered wait statements, or after one complete pass in case of an existing *sensitivity list*. Signals are assigned their starting values and the simulation time is set to zero.

The simulation is usually done by stimulating the input signals of the unit under test (UUT) with the appropriate waveforms. This is easily achieved by the so-called test bench, a special entity which resides on top of the complete unit under test. The test bench generates the stimuli waveforms for the input signals of the unit under test by either a behavioral description or by reading them from a file. It is also possible to have the output signals from the UUT read, checked for correctness or written to a file by the test bench. Figure illustrates the test bench concept.

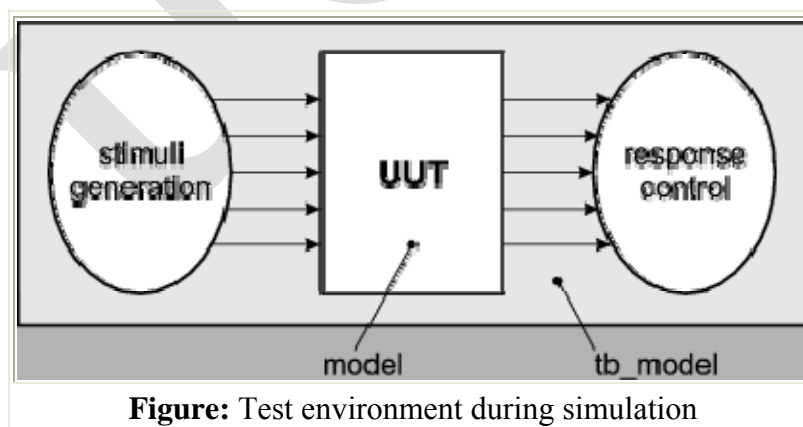


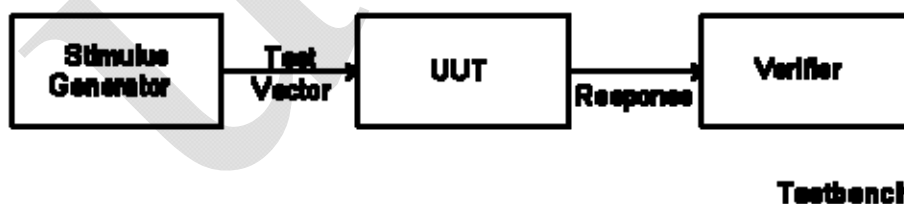
Figure: Test environment during simulation

13. Automated Testbench Generation

With the ever-increasing size and complexity of electronic designs, mixed hardware and software components, and tight schedules, the design verification becomes more and more of a challenge. To help designers overcome this challenge, Active-HDL provides tools that automate test bench creation as well as support for test benches that generate randomized, constrained stimulus. This page presents simple to-use, point-and-click tools as well as advanced C-based methodologies.

The **HDL test bench** is a VHDL or Verilog program that describes simulation input using standard HDL language procedures. Simply speaking, the **test bench** is a top-level hierarchical model which instantiates the Unit Under Test (UUT) and drives it with a set of test vectors and compares the generated results with expected responses. A typical VHDL or Verilog test bench is composed of three main elements:

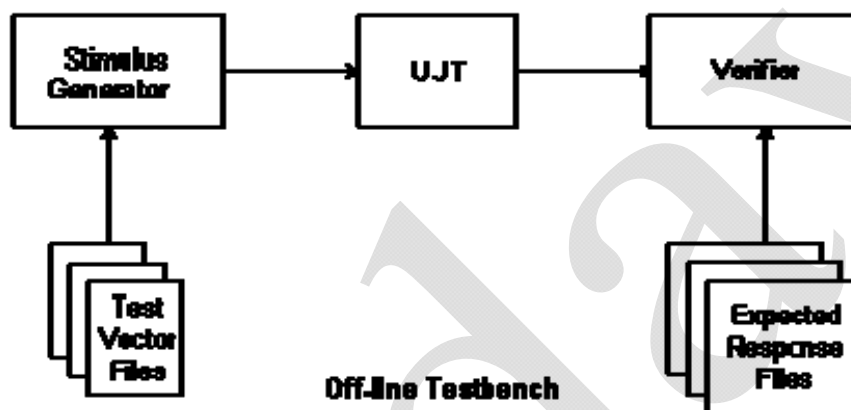
- **Stimulus Generator**; driving the UUT with certain signal conditions (correct and incorrect transactions, minimum and maximum delays, fault conditions, etc.)
- **Unit Under Test (UUT)**, representing the model undergoing verification.
- **Verifier**; automatically checking and reporting any errors encountered during the simulation run. It also compares model responses with the expected results.



To verify and add to report's fault conditions in the most efficient way, **Stimulus Generator** and **Verifier** uses textual files that store test vectors and simulation results. With that basis, we can distinguish three test bench configurations.

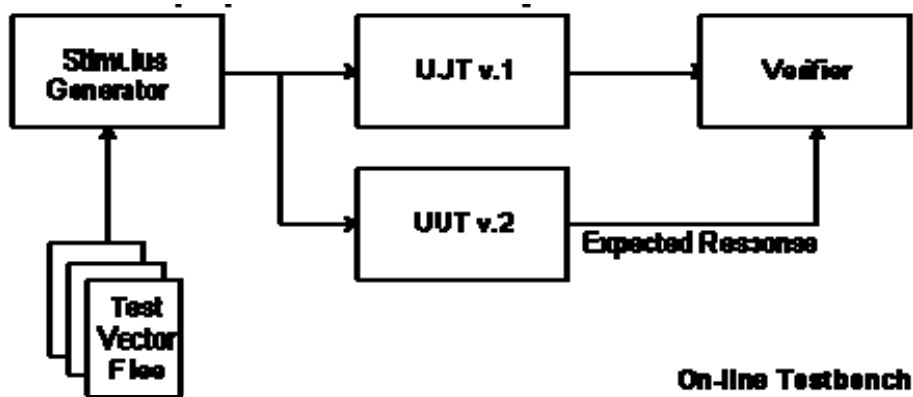
13.1 Off-line Configuration

In this configuration the **Stimulus Generator** and **Verifier** take advantage of a file access, and read all needed information (test vectors, expected results) from the previously saved files. The **Stimulus Generator** reads all input signals from a file and provides clock processes. The **Verifier** on the other hand compares the UUT responses with the expected results and reports any faulty behavior. VHDL and Verilog constructs allow logging all responses in a file for documentation purposes and future verification. For example, you can use the same test bench (with the same test vectors) to verify any changes in a model at different stages of development (logical, pre-layout, P&R) by exchanging the instantiated component.



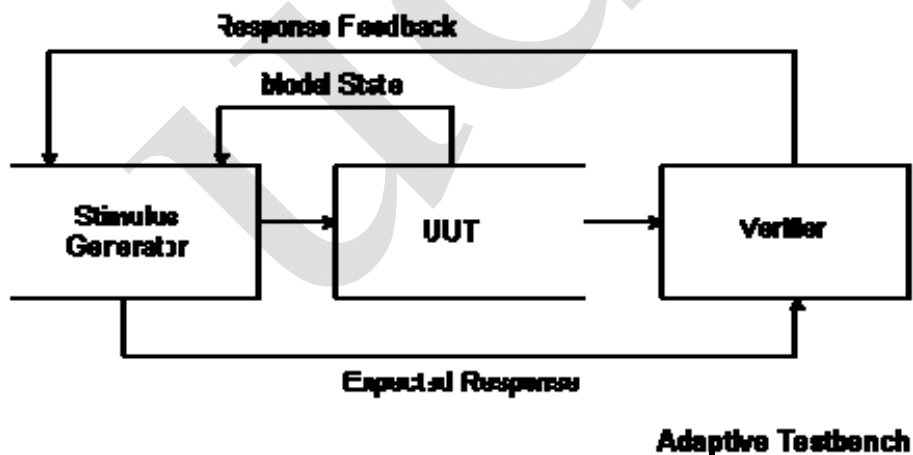
13.2 On-line Configuration

Test benches also allow you to interactively verify any model changes at different stages of design development. The **Stimulus Generator** provides the same input signals to each tested model. Thus the response of all models are simultaneously generated and without any user interaction such as exchanging the components. The **Verifier** operation is much simpler than in the off-line configuration, because it only gathers the simulation results from each model and compares them, detecting any differences and deciding whether to continue the simulation or not.



13.3 Adaptive Configuration

Unlike on-line and off-line configurations, the adaptive test bench configuration requires a good knowledge of VHDL or Verilog constructs and capabilities. Here, the stimulus generator uses high-level abstraction techniques to adapt test vectors to the changing conditions and responses of a tested model. As a result test vectors are generated in response to feedback from the UUT and **Verifier**. For example: effective testing models containing such communication protocols as UART or PCI bus. For more information about writing your own simulation test benches, please refer to the VHDL and Verilog literature.



13.4 VHDL Testbench Example:

```
architecture TESTBENCH_ARCH of test_bench is
file RESULTS: TEXT open WRITE_MODE is "results.txt";
```

```

component GENERATOR is
  port (A    : in STD_LOGIC;
        B    : in STD_LOGIC;
        CLOCK : in STD_LOGIC;
        RESET : in STD_LOGIC;
        S    : in STD_LOGIC;
        Y    : out STD_LOGIC);
end component;

shared variable end_sim : BOOLEAN := false;
signal RESET, Y      : STD_LOGIC;
signal A,B,S,CLOCK  : STD_LOGIC;
-- declaration of procedure WRITE_RESULTS skipped

begin
  -- <>
  UUT: GENERATOR
    port map ( A => A,
              B => B,
              CLOCK => CLOCK,
              RESET => RESET,
              S => S,
              Y => Y      );

  CLK_IN: process
  begin
    if end_sim = false then
      CLOCK <= '0';
      wait for 10 ns;
      CLOCK <= '1';
      wait for 10 ns;
    else
      wait;
    end if;
  end process;

  S_IN: process
  begin
    if end_sim = false then
      S <= '0';
      wait for 250 ns;
    end if;
  end process;

```

```
S <= '1';  
wait for 250 ns;  
else  
wait;  
end if;  
end process;
```

```
A_IN: process  
begin  
if end_sim = false then  
A <= '0';  
wait for 500 ns;  
A <= '1';  
wait for 500 ns;  
else  
wait;  
end if;  
end process;
```

```
B_IN: process  
variable time0 : TIME := 0 us;  
begin  
while time0 < 20 us loop  
time0 := time0 + 1000 ns;  
B <= '0';  
wait for 1000 ns;  
B <= '1';  
wait for 1000 ns;  
end loop;  
end_sim := true;  
wait;  
end process;
```

```
RESET_IN: process  
begin  
RESET <= '1';  
wait for 100 ns;  
RESET <= '0';  
wait;  
end process;
```

```
WRITE_TO_FILE: WRITE_RESULTS(a,b,clock,reset,s,y);  
  
end TESTBENCH_ARCH;
```

14. VHDL / VERILOG COMPARISON

Introduction

There are now two industry standard hardware description languages, VHDL and Verilog. The complexity of ASIC and FPGA designs has meant an increase in the number of specialist design consultants with specific tools and with their own libraries of macro and mega cells written in either VHDL or Verilog. As a result, it is important that designers know both VHDL and Verilog and that EDA tools vendors provide tools that provide an environment allowing both languages to be used in unison. For example, a designer might have a model of a PCI bus interface written in VHDL, but wants to use it in a design with macros written in Verilog.

Background

VHDL (Very high speed integrated circuit Hardware Description Language) became IEEE standard 1076 in 1987. It was updated in 1993 and is known today as "IEEE standard 1076 1993". The Verilog hardware description language has been used far longer than VHDL and has been used extensively since it was launched by Gateway in 1983. Cadence bought Gateway in 1989 and opened Verilog to the public domain in 1990. It became IEEE standard 1364 in December 1995.

There are two aspects to modeling hardware that any hardware description language facilitates; true abstract behavior and hardware structure. This means modeled hardware behavior is not prejudiced by structural or design aspects of hardware intent and that hardware structure is capable of being modeled irrespective of the design's behavior.

VHDL/Verilog compared & contrasted

This section compares and contrasts individual aspects of the two languages; they are listed in alphabetical order.

Capability

Hardware structure can be modeled equally effectively in both VHDL and Verilog. When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI. The choice of which to use is not therefore based solely on technical capability but on:

- personal preferences
- EDA tool availability
- commercial, business and marketing issues

The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction IN FIG.

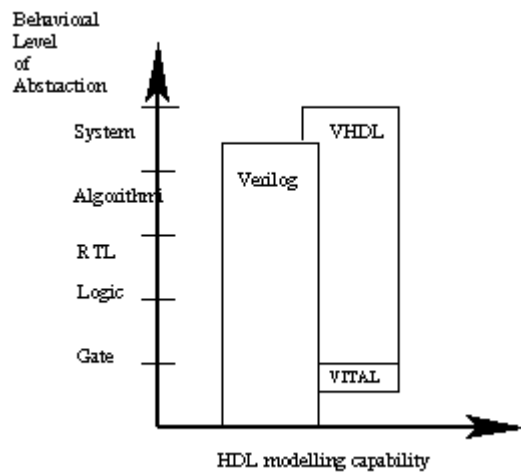


Fig. HDL modeling capability

Compilation

VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in its own system file in which case separate compilation should not be an issue.

Verilog. The Verilog language is still rooted in its native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

Data types

VHDL. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog. Compared to VHDL, Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects that are signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of its simplicity.

Design reusability

VHDL. Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.

Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the ``include` compiler directive.

Easiest to Learn

Starting with zero knowledge of either language, Verilog is probably the easiest to grasp and understand. This assumes the Verilog compiler directive language for simulation and the PLI language is not included. If these languages are included they can be looked upon as two additional languages that need to be learned. VHDL may seem less intuitive at first for two primary reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially those with large hierarchical structures.

Forward and back annotation

A spin-off from Verilog is the Standard Delay Format (SDF). This is a general purpose format used to define the timing delays in a circuit. The format provides a bidirectional link between, chip layout tools, and either synthesis or simulation tools, in order to provide more accurate timing representations. The SDF format is now an industry standard in it's own right.

High level constructs

VHDL. There are more constructs and features for high-level modeling in VHDL than there are in Verilog. Abstract data types can be used along with the following statements:

- * package statements for model reuse,
- * configuration statements for configuring design structure,
- * generate statements for replicating structure,
- * generic statements for generic models that can be individually characterized, for example, bit width.

All these language statements are useful in synthesizable models.

Verilog. Except for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modeling statements in Verilog.

Language Extensions

The use of language extensions will make a model non standard and most likely not portable across other design tools. However, sometimes they are necessary in order to achieve the desired results.

VHDL. Has an attribute called 'foreign that allows architectures and subprograms to be modeled in another language.

Verilog. The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools. For example, a designer, or more likely, a Verilog tool vendor, can specify user defined tasks or functions in the C programming language, and then call them from the Verilog source description. Use of such tasks or functions

make a Verilog model nonstandard and so may not be usable by other Verilog tools. Their use is not recommended.

Libraries

VHDL. A library is a store for compiled entities, architectures, packages and configurations. Useful for managing multiple design projects.

Verilog. There is no concept of a library in Verilog. This is due to its origins as an interpretive language.

Low Level Constructs

VHDL. Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified using the after clause. Separate constructs defined under the VITAL language must be used to define the cell primitives of ASIC and FPGA libraries.

Verilog. The Verilog language was originally developed with gate level modeling in mind, and so has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries. Examples include User Defined Primitives (UDP), truth tables and the specify block for specifying timing delays across a module.

Managing large designs

VHDL. Configuration, generate, generic and package statements all help manage large design structures.

Verilog. There are no statements in Verilog that help manage large designs.

Operators

The majority of operators are the same between the two languages. Verilog does have very useful unary reduction operators that are not in VHDL. A loop statement can be used in VHDL to perform the same operation as a Verilog unary reduction operator. VHDL has the mod operator that is not found in Verilog.

Parameterizable models

VHDL. A specific bit width model can be instantiated from a generic n-bit model using the generic statement. The generic model will not synthesize until it is instantiated and the value of the generic given.

Verilog. A specific width model can be instantiated from a generic n-bit model using overloaded parameter values. The generic model must have a default parameter value defined. This means two things. In the absence of an overloaded value being specified, it will still synthesize, but will use the specified default parameter value. Also, it does not need to be instantiated with an overloaded parameter value specified, before it will synthesize.

Procedures and tasks

VHDL allows concurrent procedure calls; Verilog does not allow concurrent task calls.

Readability

This is more a matter of coding style and experience than language feature. VHDL is a concise and verbose language; its roots are based on Ada. Verilog is more like C because its constructs are based approximately 50% on C and 50% on Ada. For this reason an existing C programmer may prefer Verilog over VHDL. Although an existing programmer of both C and Ada may find the mix of constructs somewhat confusing at first. Whatever HDL is used, when writing or reading an HDL model to be synthesized it is important to think about hardware intent.

Structural replication

VHDL. The *generate* statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately.

Verilog. There is no equivalent to the *generate* statement in Verilog.

Test harnesses

Designers typically spend about 50% of their time writing synthesizable models and the other 50% writing a test harness to verify the synthesizable models. Test harnesses are not restricted to the synthesizable subset and so are free to use the full potential of the language. VHDL has generic and configuration statements that are useful in test harnesses that are not found in Verilog.

Verboseness

VHDL. Because VHDL is a very strongly typed language models must be coded precisely with defined and matching data types. This may be considered an advantage or disadvantage. However, it does mean models are often more verbose, and the code often longer, than its Verilog equivalent.

Verilog. Signals representing objects of different bits widths may be assigned to each other. The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits, and is independent of whether it is the assigned signal or not. Unused bits will be automatically optimized away during the synthesis process. This has the advantage of not needing to model quite as explicitly as in VHDL, but does mean unintended modeling errors will not be identified by an analyzer.

Designing RTL hardware models in VHDL and Verilog

It is a requirement of the problem to synthesize the generic model with 8-bit bus signals. This is easily achieved in Verilog model by setting the default parameter value Width to 8. This means it does not need to be separately instantiated before it can be synthesized and have the correct bit width. This is not the case in VHDL, which uses a generic. The value of the generic is only specified when the model is instantiated. Although the VHDL model will be instantiated in the test harness, the test harness is not synthesized. Therefore, in order to synthesize an 8-bit GCD circuit a separate synthesizable model must be used to instantiate the RTL level model which specifies the generic, Width, to be 8. The simulation test harness does not need to use this extra model as it too, will specify the generic, Width, to be 8.

VHDL RTL model

```
library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;
entity GCD is
generic (Width: natural);
port (Clock,Reset,Load: in std_logic;
      A,B: in unsigned(Width-1 downto 0);
      Done: out std_logic;
      Y: out unsigned(Width-1 downto 0));
end entity GCD;
architecture RTL of GCD is
  signal A_New,A_Hold,B_Hold: unsigned(Width-1 downto 0);
  signal A_lessthan_B: std_logic;
begin
-----
-- Load 2 input registers and ensure B_Hold < A_Hold
-----
```

```

LOAD_SWAP: process (Clock)
begin
  if rising_edge(Clock) then
    if (Reset = '0') then
      A_Hold <= (others => '0');
      B_Hold <= (others => '0');
    elsif (Load = '1') then
      A_Hold <= A;
      B_Hold <= B;
    else if (A_lessthan_B = '1') then
      A_Hold <= B_Hold;
      B_Hold <= A_New;
    else A_Hold <= A_New;
    end if;
  end if;
end process LOAD_SWAP;
SUBTRACT_TEST: process (A_Hold, B_Hold)
begin
  -----
  -- Subtract B_Hold from A_Hold if A_Hold >= B_Hold
  -----
  if (A_Hold >= B_Hold) then
    A_lessthan_B <= '0';
    A_New <= A_Hold - B_Hold;
  else
    A_lessthan_B <= '1';
    A_New <= A_Hold;
  end if;
  -----
  -- Greatest common divisor found if B_Hold = 0
  -----
  if (B_Hold = (others => '0')) then
    Done <= '1';
    Y <= A_Hold;
  else
    Done <= '0';
    Y <= (others => '0');
  end if;
end process SUBTRACT_TEST;
end architecture RTL;

```

Verilog RTL model

```

module GCD (Clock, Reset, Load, A, B, Done, Y);
parameter Width = 8;
input Clock, Reset, Load;
input [Width-1:0] A, B;
output Done;
output [Width-1:0] Y;
reg A_lessthan_B, Done;
reg [Width-1:0] A_New, A_Hold, B_Hold, Y;
//-----
// Load 2 input registers and ensure B_Hold < A_Hold
//-----
always @(posedge Clock)
begin: LOAD_SWAP
    if (Reset) begin
        A_Hold = 0;
        B_Hold = 0;
    end
    else if (Load) begin
        A_Hold = A;
        B_Hold = B;
    end
    else if (A_lessthan_B) begin
        A_Hold = B_Hold;
        B_Hold = A_New;
    end
    else
        A_Hold = A_New;
end
always @(A_Hold or B_Hold)
begin: SUBTRACT_TEST
//-----
// Subtract B_Hold from A_Hold if A_Hold >= B_Hold
//-----
    if (A_Hold >= B_Hold) begin
        A_lessthan_B = 0;
        A_New = A_Hold - B_Hold;
    end
    else begin
        A_lessthan_B = 1;
        A_New = A_Hold;
    end
end

```

```
//-----  
// Greatest common divisor found if B_Hold = 0  
//-----  
if (B_Hold == 0) begin  
    Done = 1;  
    Y = A_Hold;  
end  
else begin  
    Done = 0;  
    Y = 0;  
end  
end  
endmodule
```

Conclusions

The reasons for the importance of being able to model hardware in both VHDL and Verilog has been discussed. VHDL and Verilog has been extensively compared and contrasted in a neutral manner. A tutorial has been posed as a problem and solution to demonstrate some language differences and indicated that hardware modeled in one language can also be modeled in the other. Room did not allow test harness models to be included in this tutorial paper, but is shown in the book "HDL Chip Design" [1]. The choice of HDL is shown not to be based on technical capability, but on: personal preferences, EDA tool availability and commercial, business and marketing issues.

15. Introduction to Verilog

Years ago, as integrated circuits grew in complexity, a better method for designing them was needed. Schematic capture tools had been developed which allowed an engineer to draw schematics on a computer screen to represent a circuit. This worked well because graphic representations are always useful to understand small but complex functions. But as chip density increased, schematic capture of these circuits became unwieldy and difficult to use for the design of large circuits. The transistor densities of Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) grew to the point where a better tool was needed. That tool is a Hardware Description Language (HDL). As more engineers design complex systems on a chip, they have no option but to use HDLs. The major advantages of HDLs are:

1. Ability to handle large, complex designs
2. Different levels of abstraction
3. Top down design
4. Reusability
5. Concurrency
6. Timing
7. Optimization
8. Standards
9. Documentation

What are HDLs?

Hardware Description Languages use statements, like programming language statements, in order to define, simulate, synthesize, and layout hardware. The two main HDLs are Verilog and VHDL. There are other, limited capability, languages such as ABEL, CUPL, and PALASM that are tailored specifically for designing PALs and CPLDs. They are not robust enough to cover the complexity required for most FPGAs and ASICs. However, both Verilog and VHDL can be used to design anything from the most complex ASIC to the least complex PAL. This paper will use examples in Verilog, and will focus on that language, but the general attributes of that language can be ascribed also to VHDL.

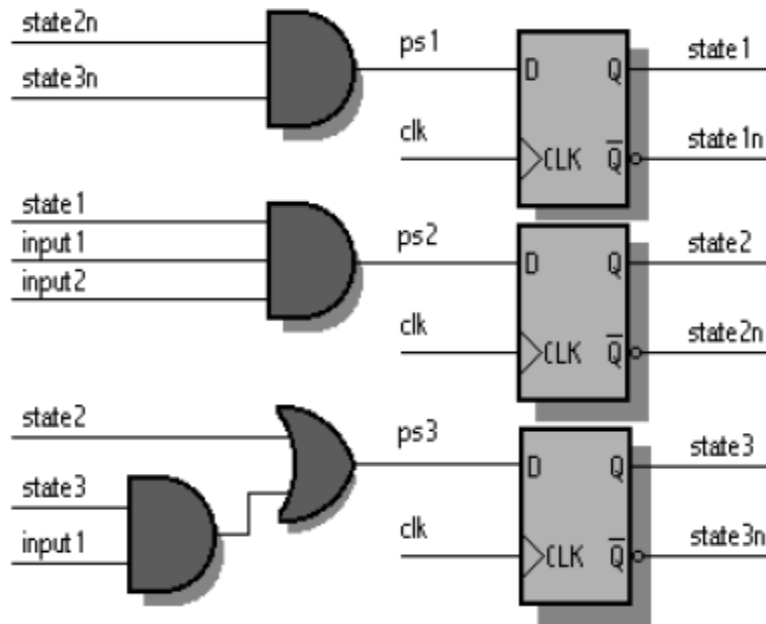


Figure 1. State machine schematic

A schematic for a simple state machine is shown in Figure 1. The equivalent Verilog code is shown in Example 1. One important difference to note is that schematic capture is limited to just the design aspect that involves physically selecting and connecting devices. HDLs, on the other hand, are used for all stages of the design as is explained further below, where the advantages of HDLs are enumerated.

// This module is used to implement the memory control state machine.

```

module state_machine(clk, input1, input2, state3);
/* INPUTS */
input clk; // system clock
input input1, input2; // inputs from the adder
/* OUTPUTS */
output state3; // can be used as the write signal
/* DECLARATIONS */
wire ps1; // input to state1 flip-flop
wire ps2; // input to state2 flip-flop
wire ps3; // input to state3 flip-flop
reg state1, state2, state3; // state bits
assign ps1 = ~state2 & ~state3;

```



```

assign ps2 = state1 & input1 & input2;
assign ps3 = state2 | (state3 & input1);
initial begin // initialize the state machine
state1 = 0;
state2 = 0;
state3 = 0;
end
always @(posedge clk) begin // clock in the new state on the
state1 <= #3 ps1; // rising edge of clk
state2 <= #3 ps2;
state3 <= #3 ps3;
end
endmodule // state_machine

```

Example 1. State machine

Different levels of abstraction

Algorithmic Level	Behavioral Models
Architectural Level	
Register Transfer Level (RTL)	Structural Models
Gate Level	
Switch Level	

Table 1. Levels of Abstraction

A hardware description language can be used to design at any level of abstraction from high-level architectural models to low-level switch models. These levels, from least amount of detail to most amount of detail, are given in Table 1. The top two levels use what are called Behavioral Models, while the lower three levels use what are called Structural Models.

Behavioral Models

Behavioral models consist of code that represents the behavior of the hardware without respect to its actual implementation. Behavioral models don't include timing numbers. Buses don't need to be broken down into their individual signals. Adders can simply add two or more numbers without specifying registers or gates or transistors. Behavioral models can be classified further as Algorithmic or Architectural.

Algorithmic models

Algorithms are step-by-step methods of solving problems and producing results. No hardware implementation is implied in an algorithmic model. An algorithmic model is not concerned with clock signals, reset signals, or any kind of signals. It does not deal with flip-flops, gates, or transistors. There is no need to specify how a numeric quantity is represented in hardware, or what kind of memory device is used to store it.

For example, suppose we are designing an Arithmetic Logic Unit (ALU). This ALU takes two numeric inputs *operand_a* and *operand_b*, and a control input called *operation*. The ALU performs one of four operations, addition, subtraction, multiplication, or division, depending on the value of the *operation* input. The result is called *result_c*. Our algorithmic model could look something like this:

```
if operation = 0, then result_c = operand_a + operand_b
if operation = 1, then result_c = operand_a - operand_b
if operation = 2, then result_c = operand_a * operand_b
if operation = 3, then result_c = operand_a / operand_b
```

Architectural models

Architectural models describe hardware on a very high level, using functional blocks like memory, control logic, CPU, FIFO, etc. These blocks may represent PC boards, ASICs, FPGAs, or other major hardware components of the system. An architectural model deals with hardware issues like how to represent signed integers, how much memory is required to store intermediate values, etc. An architectural description may involve clock signals, but usually not to the extent that it describes everything that happens in the hardware on every clock cycle. That is the responsibility of a Register Transfer Level (RTL) description. An architectural description is useful for determining the major functions of a design. An architectural model of our ALU might look like this:

```
main_block:
declare operand_a as a 16-bit bus
declare operand_b as a 16-bit bus
declare result_c as a 16-bit bus
declare mem as a 3 by 16 memory
wait for the rising edge of the clock signal
store operand_a in mem[0]
store operand_b in mem[1]
if operation = 0, then use adder_block
if operation = 1, then use subtractor_block
if operation = 2, then use multiplier_block
```

if *operation* = 3, then use *divider_block*
wait for five more rising edges of the *clock* signal
read *result_c* from *mem[2]*

In this example, the architecture of the hardware now consists of six sections: *main_block*, *adder_block*, *subtractor_block*, *multiplier_block*, *divider_block*, and *memory*. Only the *main_block* section is shown in the code above. A diagram of this architecture is shown in Figure 2.

We have divided the hardware into smaller building blocks, but each block still has a high level of functionality. We are concerned now with how to represent the numbers. We are concerned about clock signals, but not what the hardware is doing on every clock cycle.

Structural Models

Structural models consist of code that represents specific pieces of hardware. Structural models can be further classified as register transfer level (RTL), gate level, or switch level models.

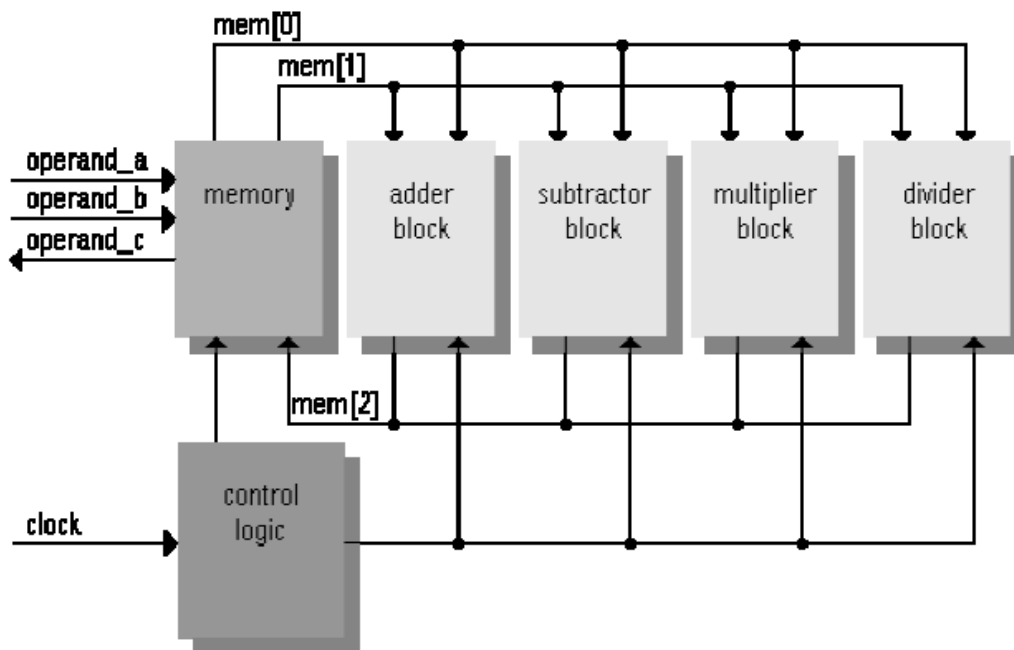


Figure 2. Architectural block diagram

Register transfer level (RTL) models

Register Transfer Level descriptions, commonly called RTL, specify hardware on a register level. In other words, RTL level code specifies what happens on each clock edge. Actual gates descriptions are avoided, although RTL code may use Boolean functions that can be implemented in gates.

High level descriptions of hardware functions can be used, as long as these functions have a behavior that is associated with clock cycles. State machines are good examples of RTL descriptions. The functionality of a state machine can be complex, but the RTL state machine is defined by what occurs on each clock cycle. An example of some sequential logic is shown in Figure 3. The schematic is on the left while an RTL description is on the right.

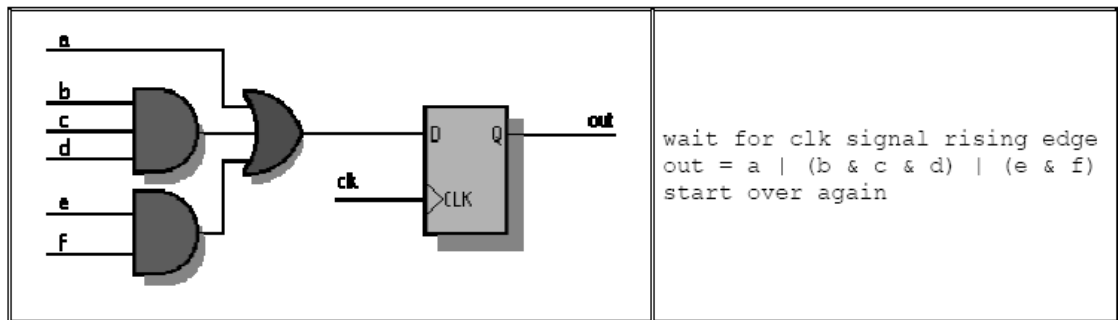


Figure 3. RTL Description of a Sequential Circuit

Gate level models

Gate level modeling consists of code that specifies very simple functions such as NAND and NOR gates. The same sequential circuit shown in Figure 3 above would be described on a gate level using primitive NAND, NOR, and DFF gates as shown in Figure 4 below. `nand3(b,c,d,x)`
`nand2(e,f,y)`
`nor3(a,x,y,z)`
`dff(z,clk,out)`

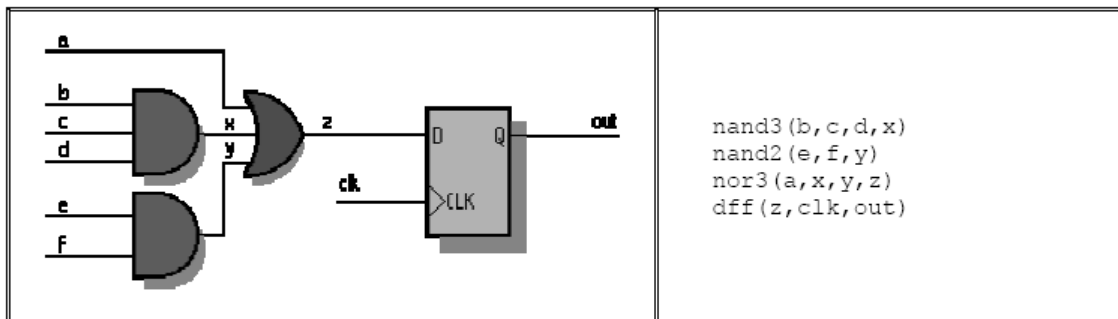


Figure 4. Gate Level Description of a Sequential Circuit

Switch level models

Finally, the lowest level description is that of switch-level models, which specifies the actual transistor switches that are combined to make gates. ASIC vendors and other semiconductor manufacturers may use this level of modeling to simulate their circuits.

The advantage to HDLs is that all of these different levels of modeling can be done with the same language. This makes all the stages of design very convenient to implement. You don't need to learn different tools. You can easily simulate the design at a behavioral level, and then substitute various behavioral code modules with structural code modules. For system simulation, this allows you to analyze your entire project using the same set of tools. First, the algorithms can be tested and optimized. Next, the behavioral models can be used to partition the hardware into boards, ASIC, and FPGAs. The RTL code can then be written and substituted for behavioral blocks one at a time to easily test the functionality of each block. From that, the design can be synthesized into gate and switch level blocks that can be re simulated with timing numbers to get actual performance measurements. Finally this low level code can be used to generate a net list for layout. All stages of the design have been performed using the same basic tool.

Top-Down Design

Behavioral level modeling is particularly useful for doing top-down design. Top-down design is a methodology where the highest level functions and algorithms are described first, followed by more and more detailed designs that more directly relate to actual hardware.

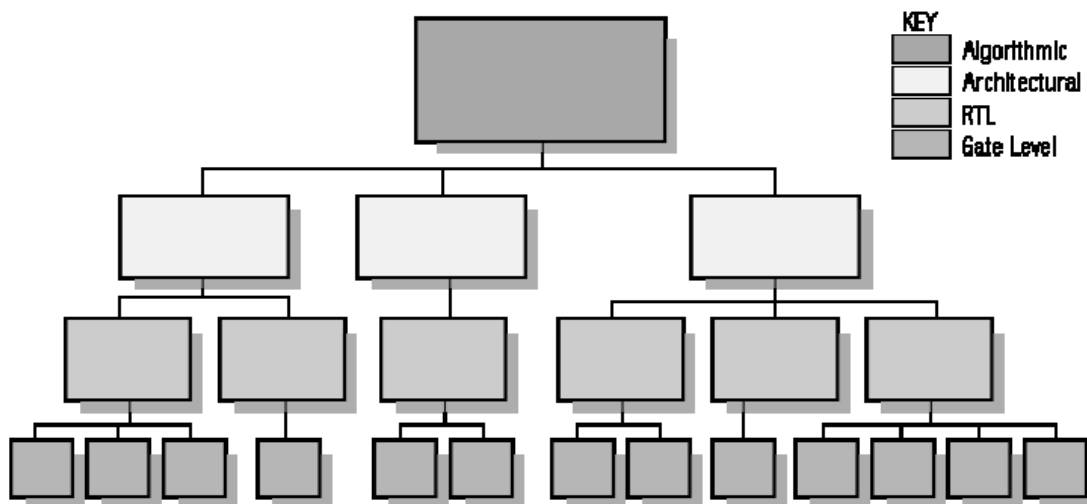


Figure 5. Top Down Design

Top-down design is the design method whereby high level functions are defined first, and the lower level implementation details are filled in later. A hardware design can be viewed as a hierarchical tree as shown in Figure 5. The top-level block represents the entire design, represented by

algorithms. Each lower level block represents major functions of the design. Intermediate level blocks contain more detailed functionality. The middle levels consist of architectural and RTL descriptions. These RTL descriptions are eventually synthesized into the bottom row of gate level descriptions. This bottom level contains only gates and macro functions, which are vendor-supplied basic functions.

Top-down design is the preferred methodology for chip design for several reasons. First, chips often incorporate a large number of gates and a very high level of functionality. Attempting the design at a low level, without understanding the entire architecture, would be overwhelming. Scheduling the design would be a huge task. Top-down methodology forces the design to be partitioned into simpler pieces. This allows for better scheduling and resource allocation. For example, a junior engineer may be given a small piece of the chip to design while a senior engineer may be given several large pieces. Each engineer can complete his or her piece without needing to wait for the other team members to complete their pieces. If one piece gets delayed, the rest of the chip can still be completed on schedule. Even if only one engineer is responsible for the design, this partitioning makes a very complex task into a number of very simple tasks. Also important is the fact that simulation is very much simplified using this design methodology. Simulation is an extremely important consideration in ASIC design since an ASIC cannot be blue-wired after production. For this reason, simulation must be done extensively before the ASIC is sent for fabrication. Using a top-down approach, first the behavioral description is simulated. This is done to determine that the overall function of the design is correct. Then the design is broken into smaller pieces of RTL code. Each RTL function block can be simulated independently, simply by substituting the RTL blocks into the equivalent behavioral block and rerunning the simulation. The results should be consistent. If not, you've found a bug. Because a simulation of the full RTL or gate level description can be very time consuming, mixing and matching RTL and behavioral descriptions greatly speeds up simulation time and, thus greatly speeds up bug detection and correction.

This methodology also allows flexibility in the design. Sections can be removed and replaced with higher-performance or optimized designs without affecting other sections of the chip. This allows you to easily try out different designs by simply plugging in new hardware descriptions and simulating.

Reusability

Reusability is a big advantage of HDLs. Code written in one HDL can be used in any system that supports that HDL. Schematics, on the other hand, are only useful in a particular schematic capture software tool. Even using the same tool, portability can be difficult if a module does not physically fit into the new design. A behavioral level HDL model, and most RTL level models can be easily used over and over again on multiple designs.

Concurrency

Concurrency is an advantage that HDLs offer over normal software languages which can also be used to simulate hardware. With a normal software language, statements are executed sequentially. With an HDL, on the other hand, provisions have been added to support concurrent execution of statements. This is an absolute necessity since in a hardware design, many events occur simultaneously. For example, in a synchronous design, all flip-flops on a particular clock line must be evaluated simultaneously. While normal software languages can be used to model simultaneous events, it is up to the programmer to add the mechanism for handling this. With an HDL, the mechanism is built into the language.

Timing

With schematic capture tools, when it comes time to simulate a design, the timing numbers are embedded in the netlist that is generated from the schematic. These timing numbers are based on parameters supplied by the vendors whose chips are being used. The user has some limited ability to change these numbers and some of the parameters.

With HDLs, the timing numbers are explicitly stated in the code as shown in Example 2. Nothing is hidden from the user, and the user has complete control over these numbers. This makes it much easier to control and optimize the timing of your design.

```
module state_machine(sysclk, input1, input2, state3)
```

```
...
```

```
// Output delays are specified with the # symbol
```

```
assign ps1 = #10 ~state1 & ~state2;
```

```
assign ps2 = #5 state1 & input1 & input2;
```

```
assign ps3 = #12 state2 | (state3 & input1);
```

```
initial begin // initialize the state machine
```

```
#13; // wait 13 time units before initializing
```

```
state1 = 0;
```

```
state2 = 0;
```

```
state3 = 0;
end
always @(posedge sysclk) begin
state1 <= #3 ps1; // output delay = 3 time units
state2 <= #3 ps2;
state3 <= #3 ps3;
end
endmodule
```

Example 2. Code with explicit timing

Optimization

HDLs are particularly useful for optimization. The most common method of designing ASICs and FPGAs involves writing an RTL level design and then using a software tool to “synthesize” your design. The synthesis process involves generating an equivalent gate level HDL description. Using various methods, the resulting description can be optimized to suit the particular target hardware. FPGAs, for example, typically have a very well-defined, large grain architecture. Mapping a gate level design to an FPGA would normally be difficult. However, you can write an RTL level model and synthesize it specifically for a particular FPGA. That same RTL description can be used in the future and synthesized for a particular ASIC technology.

Standards

Both major HDLs, Verilog and VHDL, are public standards. Verilog was initially a privately developed language, which was released to the public in 1990 and is now maintained by the Open Verilog International (OVI), a consortium of companies that sustain and improve the language. It has been adopted by the Institute of Electrical and Electronic Engineers (IEEE) as the standard IEEE-STD-1364. VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, and is supported by the U.S. Department of Defense. It was adopted by the IEEE as the standard IEEE-STD-1076. Because these languages are public standards, it ensures that a design written in the language can be accepted by every software tool that supports the language. It also means that you are not tied in to purchasing the tools of any one company, but can choose from tools offered by many vendors.

Documentation

HDLs, being text-based, programming-type languages, lend themselves easily to documentation. This is not to say that the code is self-

documenting. However, the code is relatively easy to read, and a code module shows much about the functionality, the architecture, and the timing of the hardware. In addition, as with any programming language, statements can be organized in ways that give more information about the design, and comments can be included which explain the various sections of code. It is up to the designer and the project leader to determine how to document the code, but the nature of HDLs certainly encourages this type of documentation.

Large, Complex Designs

Large, complex designs require all of the above features. Because of this, HDLs are much better suited to large, complex designs than schematic capture or any other methods of design currently available.

Reference:

1. Active-HDL Reference, <http://www.aldec.com/ActiveHDL/>
2. J. Bhaskar, A VHDL Primer. Englewood Cliffs, NJ: Prentice Hall, 1995.
3. J. Bhaskar, A VHDL Synthesis Primer, Star Galaxy Publishers, 1996.
4. K. C. Chang, Digital Systems Design with VHDL And Synthesis: An Integrated Approach, Wiley- IEEE Computer Society, 1999.
5. B. Cohen, VHDL Coding Styles and Methodologies. Boston, MA: Kluwer Academic, 1995.
6. IEEE Standard VHDL Language Reference Manual: ANSI/IEEE Std 1076–1993. New York : IEEE, June 1994.
7. Randall L. Geiger, “VLSI Design Techniques for Analog and Digital Circuits”, McGraw Hill Int. Co., 1990.
8. VHDL Programming by Douglas Perry, McGraw Hill Int. fourth edition.
9. Verilog Digital System Design by Zainalabedin Navabi, McGraw Hill Professional.