This text is about starting to learn how to design large digital systems. A large part of that activity is knowing how to use and apply electronic design automation tools including hardware description languages (HDLs) for design specification, simulation, and synthesis.

This chapter describes the essential VHDL needed for this text. VHDL is used in this text for designing, simulating, and implementing digital systems at the gate and register transfer level (RTL) of specification, and hence we concentrate on the synthesis subset of the language. VHDL, because it is a concurrent programming language with support for describing circuit delays and structure, can also be used to create abstract high-level models of digital systems. These simulation models, and those aspects of the language, can be extremely complex and are beyond the scope of this text.

VHDL is chosen because it is an IEEE standard and can be used to write abstract system-level models to gate level specifications for programming a simple field programmable device (FPD). Verilog would also be a good choice, but most low cost tools support VHDL and not Verilog. As to other specialized or vendor specific languages that just target FPDs, they are not really any simpler to learn, and are not sufficient for complex tasks. VHDL will be a language you can use throughout your engineering career whether you are programming a 16V8 PLD or modeling a flight control system.

The following style guidelines should be sufficient to study the problems posed in this text. These are a minimal synthesis set and will work with almost all VHDL synthesis tools (with minor modifications). See your tools for additional supported constructs. This chapter provides general background. Additional language details and examples are provided in subsequent chapters.

---

### IEEE Library Declarations

Every VHDL file should begin with library declarations. Generally these will be the IEEE libraries that are becoming a standard for VHDL synthesis. It may also be necessary to include vendor specific libraries to access various vendor specific features.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

The library declaration makes the library IEEE visible and the use declarations cause the packages std_logic_1164, std_logic_arith, and std_logic_unsigned to be used when referenced. Packages contain useful predefined language elements such as standard types and over-loaded arithmetic operators such as arithmetic addition ("+"") and subtraction ("-"") that can be used with logic valued types such as std_logic_vector.

### Types

VHDL defines data representation types, and the scalar and array types include objects such as integer, real, and bit. For various reasons none of these are quite right for describing digital circuit values. Most synthesis tools, and simulation models, use the type std_logic from the IEEE library. The type std_logic is a resolved type that can model buses by allowing the representation of multiple tri-state drivers driving a signal and is derived from a base type std_ulogic.
Essential VHDL

TYPE std_ulogic IS ('U', -- Uninitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- Weak Unknown
'L', -- Weak 0
'K', -- Weak 1
'-' -- Don’t care
);

Figure 7.2
VHDL Std_logic Values

For modeling of digital systems the type std_logic with its range of signal values other than just '0' and '1' can better represent the physical behavior of digital systems. For synthesis, where the function of a circuit is defined, as opposed to modeling the behavior of an existing system, only four of the nine possible values are needed. These include the fundamental boolean values '1' and '0', 'Z' to represent the high-impedance state, and '-' to indicate a don’t care value. The '-' is the equivalent of a don’t care in a K-map. The other values such as a forcing unknown ('X') would never be used in synthesis, though it might appear in simulation due to a design flaw from a driver conflict on a signal.

In this text we will use the types std_logic and std_logic_vector to the near exclusion of all other types. Std_logic is a scaler (a one bit signal) and std_logic_vector is an array (or composite or aggregate) of one bit values. Std_logic_vectors are used to represent buses. Our std_logic_vectors will always have a descending range. That is (N downto 0). The least significant bit (LSB) 2^0 will be X(0) and the most significant value 2^n (MSB) will be X(N). Constant std_logic values are represented by '0', '1', '-', and 'Z', while std_logic_vectors are a string value of 1's, 0's, '-', and 'Z' surrounded by double quotes, e.g. "00110010".

Figure 7.3

Most synthesis tools will support synthesis of integers when the range is constrained by a subtype declaration. In other words, if we declare that an integer signal can only hold values in the range 0 and 15 the synthesis tool can map this onto a 4-bit binary vector (which can represent the decimal values 0 through 15). Using unconstrained integers will result in a mapping onto a 32-bit vector which will create an excessively large solution which will be difficult to simulate, difficult to synthesize, difficult to map to an actual device, and typically is not what is wanted. While using integer subtypes makes for more readable VHDL, it may require using conversion functions to convert between the integer subtype and the types std_logic and std_logic_vector.
It is also possible to use non-numeric types in synthesis. These enumerated types allow specifying integer values as string literals. They are very useful for describing states and distinct operations. Descriptive names such as START, STOP, RUN are easier to understand than binary or integer encodings such as “001”, “010”, and “100”. Typically the synthesis tool will encode the enumerated types for conversion to a state machine depending on its settings. For example a
Entity Declarations

Every digital circuit has input and output ports. VHDL uses an ENTITY declaration to describe the circuit’s ports. You can think of the ENTITY declaration as a block in a block diagram. The ENTITY declares the names of the inputs and outputs, specifies their direction, and defines their type and width.

```vhdl
entity ENTITY_NAME is
  port( SIGNAL_NAMES: DIRECTION_MODE TYPE );
end ENTITY_NAME;
```

Figure 7.6  VHDL Basic Entity Declaration

Ports, the inputs and outputs of a circuit, have a mode that specifies their direction. Modes used in this text are IN, OUT, INOUT, and BUFFER. Ports of IN can only be read in the entity’s architecture. Trying to write, or assign a value to a port of mode IN will result in an error. In other words, ports of mode IN can only appear on the right hand side (RHS) of an assignment statement. Ports of mode OUT can only be written, attempting to read the value will result in an error. Ports of mode OUT can only appear on the left-hand side (LHS) of an assignment statement.

A port of INOUT can be both read and written by the entity and is usually used for true bi-directional ports. INOUT ports can be driven by other modules and these external values can be read by the entity. BUFFER ports can be written and read, but only the value written by the entity can be read. In other words a BUFFER is like an OUT tied to an IN. Ports of mode BUFFER can be used to avoid using an internal signal to store a value, allowing a port to be both a port and the output of a register or combinational logic.

```vhdl
SIGNAL_NAME: MODE TYPE;
```

Figure 7.7  VHDL Basic Port Declaration

The basic port declaration has a SIGNAL_NAME that is the identifier or name of the port. The only thing to remember here is not to use a VHDL keyword (like "entity" or "IN" or "OUT") or start the identifier with a number. Popular signal_names might be AIN, Data_out, Clock, and reset. VHDL literals are not case sensitive. In other words AIN, ain, Ain, and aln all refer to the same identifier. The MODE is either IN, OUT, INOUT, or BUFFER, and the type will typically be a std_logic scaler or a std_logic_vector.

Architecture Declarations

While the entity declaration is used to describe the ports of the digital system, the architecture is used to describe the behavior and internal structure of the system.

```vhdl
architecture ARCH_NAME of ENTITY_NAME is
  SIGNAL_DECLARATIONS;
begin
  THE_HARD_STUFF;
end ARCH_NAME;
```

Figure 7.9  Basic Architecture

Every architecture has a name (ARCH_NAME) and is associated with an entity (ENTITY_NAME). There is a declarative region between the architecture declaration and the BEGIN in which signals (circuit nodes) internal to the component are declared. While ports on entities are visible to other entities, internal signals are not. After the BEGIN is where the “hard stuff” is specified. Here is

Figure 7.8  Example Entity Declaration
where the statements that describe the behavior and structure of the digital system are declared. The sample architecture shows some basic signal declarations.

```
architecture synthesis of design is
  signal X, Y, Z: std_logic;
  signal tmp: std_logic_vector(7 downto 0);
begin
  --The Hard Stuff Missing
end synthesis;
```

Figures 7.10 Sample Architecture

Describing Circuit Behavior

At this point the "boiler-plate" needed for the basic use of VHDL for synthesis has been described. "Boiler-plate" is the standard stuff for all designs. (As in steel plate to build a boiler.) We now need to study how to define the behavior and structure of digital systems in a subset of VHDL. It is important to remember that VHDL is a large language, used for many different purposes and only a tiny subset of it is presented in this text. VHDL that might be acceptable to a full VHDL simulator may not synthesize, or may not produce the hardware implementation you expect. Unlike programming and compiling a program, different forms of the same specification will frequently result in quite different hardware implementations. Synthesis tools require more piloting than compilers to arrive at the desired circuit implementation, size, and performance.

Concurrent Statements

Unlike programming languages, some statements in a VHDL architecture execute concurrently, not in sequential order typical of most programming languages. This is necessary if we are to be able to describe digital hardware. In a digital circuit consisting of multiple components, all the components are active simultaneously, updating their outputs whenever an input changes. The hardware components execute concurrently. In many ways "hardware design" is the design of concurrent system. While once the realm of the hardware designer, concurrent systems design is becoming more a part of software systems, ever more blurring the line between "hardware" and "software".

```
architecture synthesis of example begin
  --These three statements execute simultaneously
  --whenever X, Y, A, or B change
  A <= X AND Y;
  B <= X OR Y;
  C <= A AND B;
end synthesis;
```

Figures 7.11 Concurrent Statements

Concurrent signal assignment statements are the simplest VHDL statements and are typically used to specify combinational logic. Operators available for our use include AND, OR, NOT, NAND, NOR, XOR, XNOR, + (binary addition), - (binary subtraction), and & (concatenation). Basic concurrent signal assignments can be used to specify basic boolean functions, leaving it up to the synthesis tools to minimize and optimize the equation.

```
A <= X + Y;
B <= (M AND N) OR (O AND P);
C <= X & Y;
```

Figures 7.12 Legal Concurrent Assignment Statements

VHDL is a strongly typed language. The types on the RHS and the LHS must be the same, and if vectors (arrays), must be of the same length. This makes sense and helps to avoid design errors. One should not be able to tie a 4-bit bus to an 8-bit bus unless one explicitly resolves the dangling 4 bits.
signal X: std_logic_vector(3 downto 0);
signal Y: std_logic_vector(7 downto 0);
begin
  Y <= "0000" & X; -- Extending a 4 bit bus with zeros.
end process;

Figure 7.13
Concatenating Vectors

Most of the operators like AND, OR, and arithmetic addition “+” should be familiar to you. One that may not have an obvious function is the concatenation operator “&”. Concatenation allows signals to be combined into larger composites (bigger arrays or buses). The opposite, slicing apart a bus, is implied by the language.

An important issue in writing boolean equations in VHDL is the precedence of operators. This determines how the language associates operators. If two operators, such as AND and OR, have the same precedence, the association must be made clear by the use of parenthesis. NOT has a higher precedence than AND and OR so that it associates with the nearest literal without parenthesis. Of course using parenthesis even when not required won’t hurt, and will make the equation more readable and less prone to errors.

A <= X AND Y OR W AND Z; -- AND and OR have the same precedence.
-- Parenthesis must be used to indicate
-- the order of evaluation or an error
-- will be reported.
A <= (X AND Y) OR (W AND Z); -- Typically what is wanted
B <= NOT M AND N; -- This is the same as ...
B <= (NOT M) AND N; -- as the NOT has higher precedence than AND
-- hence NOT M is evaluated before the AND term.

Figure 7.15
VHDL Boolean Operator Precedence

At this point with simple signal assignment statements we can write boolean equations in VHDL, and even arithmetic equations using the overloaded “+” and “-” operators, but we are still forced to derive the boolean equations manually. The advantage of VHDL comes from using complex VHDL concurrent signal assignment statements and letting the synthesis tool derive the implied boolean (logical) equations and their optimized implementation.

Concurrent Signal Assignment Statements

We will use two forms of concurrent signal assignment statements. The conditional signal assignment statement allows the expression of arbitrary if-then-elsif-then-else-end conditionals in concurrent signal assignment statements. This statement is the equivalent of an if statement within a VHDL process statement. (The process statement is discussed later.)
The general form of the conditional signal assignment consists of expressions to be assigned to the signal and arbitrary conditions which determine when each expression is assigned to the signal. This statement looks and acts like an arbitrary if-elsif-else statement, and it has a shortcoming with respect to hardware. An if-elsif-else statement implies a priority. If two conditions are true, the first in the list is executed. This results in more hardware to enforce the order of execution priority, unless the conditions are mutually exclusive in which case a CASE statement can be used.

Another problem is that if the statement does not have a trailing else the hardware generated will have a latch generated. The reason is quite simple. Without an else the signal being assigned may not be assigned a value every time the if-elsif is executed. This implies that the signal must hold it previous value and hence a latch is needed in the digital circuit to faithfully replicate that behavior. Many synthesis tools enforce the else in conditional signal assignment statements, some do not. Either way, if you use a conditional signal assignment statement make sure to include the final else to ensure that combinational logic is generated.

While the conditional signal assignment is like an if-elsif-else statement the selected signal assignment statement is equivalent to a case statement.

with X select A <= B when "00", C when "01", D when others;

process begin  case X is   when "00" => A <= B;   when "01" => A <= C;   when others => A <= D;  end case;end process;
Conditionals in VHDL test the relationships between objects such as signals, ports, and constants. The usual operators are available including equality (=), inequality (=/=), less than (<), less than equal (<=), greater than (>), and greater than or equal (>=). Comparisons can be between signals or between signals and literals (constants). The IEEE libraries allow the comparison of std_logic_vectors with integer literals and signals.

```
signal X: std_logic_vector(7 downto 0);
signal Y: std_logic_vector(7 downto 0);
signal A: std_logic;
signal B: std_logic;
X = "00001000" --Is X equal to 8
X /= 255 --Is X not equal to 255 decimal
X > Y --Is X greater than Y
(X < 16) OR (X > 32) --Is X less than 16 or greater than 32
A = X(0) --Compare two bits
(A AND B) = '1' --Compare a boolean equation to a constant
```

At this point we can write an entity, create an architecture, and even describe combinational circuits in VHDL. But this does not allow us to describe more complex behaviors associated with sequential circuits, nor to take full advantage of the synthesis system’s ability to generate the digital circuit implementation from a complex VHDL description using familiar programming constructs.

Within the body of an architecture the general form of concurrent statement is the process statement. The process statement is the general method for the expression of concurrency in VHDL. The process statement contains statements that execute in sequential order and in zero time. This allows specifying complex behaviors of components above the boolean equation level. Any concurrent signal assignment statement (as just described) can be expressed as process statement. Conditional and selected signal assignments statements are actually a shorthand for a corresponding process with an if-elsif-else or case statement within a process declaration.

The process statement has the general form of a process declaration, a declarative region for variables local to the process, and then a statement declarative region. Of all the VHDL constructs we will use in this text, the process statement will be the most simplified. It will be used for describing all clocked circuits, and complex combinational circuits. It is also how concurrency is expressed in VHDL, with multiple processes within a single architecture expressing the concurrency of digital hardware.

```
process
  --Variables are declared here in the process declarative region
begin
  --Sequential statements are here in the statement declarative region
end process;
```

Figure 7.21  Conditional Expressions

Figure 7.22  Process Declaration
architecture count of synthesis is
signal A: std_logic_vector(3 downto 0);
begin
    process
        begin
            wait until clk='1';
            if reset = '0' then
                A <= "0000";
            else
                A <= A + 1;
            end if;
        end process;
    end synthesis;

Figure 7.23
Simple Counter Process

The above process reads exactly as one expects and the synthesis tool will create the expected circuit. This process waits until the clk goes from '0' to '1' then executes the sequential statements. Notice that the process has only one wait statement and it is the first statement. Do not try to use more wait statements or use them elsewhere in the process with an RTL level synthesis tool. (Behavioral synthesis tools can support multiple wait statements.) The VHDL (or circuit) tests the value of reset, and if it is '0', A is cleared, else A is incremented by 1. The following circuit shows one possible implementation.

Figure 7.24
Synthesized Counter

An important point is to notice what has happened to the signal A. It has become a register. This makes sense in terms of the semantics of VHDL and those of a clocked circuit. Signals hold their values, and so do registers. So while a signal is a node in the circuit, in this case the synthesis system places a register driving the node A. Hence the signal A and the register that holds the value of A are one in the same. The synthesis tool inferred a register from the VHDL syntax and semantics. Not all signals become the output of registers. If A was simply assigned a value of a combinational equation, and there was no “wait until clk='1'” statement, it would be that function and appear to be the output of the corresponding gates. The synthesis tool inferred a register from the VHDL syntax and semantics.
use a different style for a clocked process. The asynchronous reset puts the reset condition first in the if-elsif statement giving it priority over the clock, just as in a asynchronous clear flip-flop.

```vhdl
process(clk,reset)
begin
    if reset='0' then  -- Initial conditions for registers
        elsif (clk'event and clk='1') then  -- Behavior of the circuit when not in reset.
            end if;
    end process;
```

At this point you might notice that synthesis tools depend on "style" to infer much of the function of the circuit. The condition `(clk'event and clk='1')` is a sample of full VHDL. The clause `clk'event` is true when clk has just changed value ('0' to '1' or '1' to '0') and `clk='1'` tests for clk being one. Together these two statements test if clk has just changed from '0' to '1'.

For the synthesis tool `(clk'event and clk='1')` is effectively just style keywords that directs it to implement the design using positive edge-triggerd flip-flops. The form `(clk'event and clk='1')` does simulate correctly (before synthesis in a VHDL simulator) and so is used for synthesis. As with the `wait until clk='1';` the `(clk'event and clk='1')` can only appear once as the elsif condition. No “else” part is allowed. Negative edge-triggered flip-flops can be indicated with `wait until clk='0';` or `(clk'event and clk='0').`

Style is very important to VHDL synthesis. Arbitrary VHDL may produce an error or worse synthesize to an unexpected and unwanted circuit. On the other hand, particular style guidelines may be needed by a particular tool to take advantage of a particular tool’s capability or technology. On the other hand if you are familiar with programming then feel free to experiment. For example a slight modification of the VHDL for a asynchronous reset allows it to be used for a synchronous reset simply by making the `(clk'event and clk='1')` the if conditional for an if-then-end statement.

The use of an explicit wait statement implies an synchronous reset. This is frequently not a problem as many small FPDs don’t support asynchronous resets (their flip-flops simply don’t have that capability). To specify an asynchronous reset and make use of the global asynchronous resets available in FPGAs one must
VHDL has the full range of sequential statements found in any programming language including IF, CASE, and various forms of LOOP. We will not use the LOOP constructs much (though they can be synthesized if the loop values are constants). Loops in VHDL are typically used for describing combinational logic and implied structure. VHDL also has GENERATE statements for the conditional generation of structural descriptions. Advanced synthesis tools can convert general for loops into sequential machines by inferring and creating the state machine to implement the loop counter and control logic.

```
if conditional then
  --sequential statements
elsif conditional then
  --sequential statements
else
  --sequential statements
end if;
```

What happens with an “improper if” that does not cover all cases is that when A is not equal to "00" or "01" no value is assigned to X. Since the value of X is not defined for all values of A it must hold its value for values of A other than "00" and "01". Hence while this process does not have a clock it still implies a register and the synthesis tool builds a latch controlled by the value of A (and not the clock). It will latch X on A = "00" or A = "01". While expecting a combinational equation, one ends up with an unclelocked transparent latch controlled by combinational logic, and very unpredictable behavior.

```
if (clk'event and clk='1') then
  if reset = '1' then
    A <= "0000";
  else
    A <= A + 1;
  end if;
end if;
end process;
```

Figure 7.28

**Alternative Synchronous Reset**

```
begin
  if reset = '0' then
    X <= "0000";
  else
    if A = "00" then
      X<= X+1;
    else
      X<= X;
    end if;
  end if;
end process;
```

Figure 7.30

**If Conditional in a Process**

Use if statements freely in clocked process. You can also nest them as necessary. You get what you expected. A register to hold X that is only incremented by one when A = "00". But things go awry if the if conditional does not cover all cases and the circuit is not clocked.

```
process(A,X)
begin
  if A = "00" then
    X <= X + 1;
  elsif A = "01" then
    X <= X + 2;
  end if;
end process;
```

Figure 7.31

**Improper IF**
process (A, X)
begin
if A = "00" then
    X <= X + 1;
elsif A = "01" then
    X <= X + 2;
else
    X <= X + 4;
end if;
end process;

Figure 7.32
Proper IF Covers Every Case

The above unclocked "proper if" process will generate a purely combinational circuit. Notice the "else" condition. X is assigned a value for any possible execution of this process. The following process achieves the same result, in a much more round about way, by creating a variable, assigning a value to it outside of the if, changing the value if the if statement executes, and then finally assigning it to the target signal X.

process
    variable tmp: std_logic_vector(1 downto 0);
begin
    tmp := X + 4; --tmp gets unconditionally assigned
    if A = "00" then
        tmp := X + 1;
    elsif A = "01" then
        tmp := X + 2;
    end if;
    X <= tmp;
end process;

Figure 7.33
Proper IF with Variables

In general, IF statements should be used with care when specifying combinational logic. First they add logic for enforcing the implied priority of an IF statement in addition to the possibility of unintended latches. Instead use a CASE statement when possible. It will enforce that every input is covered ensuring a combinational circuit in unclocked processes.

case X is
when "00" => Z <= "001";
when "01" => Z <= "111";
when "10" => Z <= "011";
when others => Z <= "101";
end case;

Figure 7.34
Case Statement

The above case statement simply implements a truth table of input variables X(1) and X(0) and outputs Z(2), Z(1), Z(0). The use of the others condition is required for std_logic_vectors. While the case will typically explicitly cover all the combinations of '1's and '0's, std_logic defines another seven values ('Z', 'X', 'U', 'L', 'H', 'W'). These other choices are covered by the others.
if sel='1' then
A <= X;
else
A <= Y;
end if;

if SIG = "11" then
D <= X1;
end if;

case Q is
when "00" => Z <= '1';
when "01" => Z <= '0';
when "10" => Z <= '1';
when "11" => Z <= '1';
when others => null;
end case;

B <= TMP1;
esif N = '0' then  B <= TMP2;else  B <= TMP3;end if;

if M = '1' then
M
N
B
TMP2
TMP3
SEL
X
1
0
Y
A

if SIG = "11" then
D <= X1;
end if;

if sel='1' then
A <= X;
else
A <= Y;
end if;

process(d0, d1, d2, d3, d4, d5, d6, d7)
begin

tmp := "000";
ins := d0 & d1 & d2 & d3 & d4 & d5 & d6 & d7;
for I in 0 to 7 loop
tmp := tmp + ins(I);
end loop;
cnt <= tmp;
end process;

Another example is a simple parity checker of 2-input XOR gates. Here the loop is again unrolled to generate the final value of the variable result and the signal parity_out.
process(input)
  variable result : bit;
begin
  result := '0';
  for k in 0 to 3 loop
    result := result xor input(k);
  end loop;
  parity_out <= result;
end process;

Figure 7.37 For Loop Parity Checker

94 Digital Engineering with VHDL

In this course our VHDL designs are generally small enough that the entire design is readily described by a single entity/architecture pair with at most several processes. But as in schematic capture or programming there comes a point when breaking the design into separate modules makes the problem easier to solve and to understand. It also supports repeated reuse of a module. Divide and conquer.

VHDL supports the structural design of digital systems with several language constructs. First there are packages that can contain functions and procedures. The package is intended to hold types, functions, and procedures common to a design. But these simply allow organizing the language, and do not directly allow
expressing the structure of the digital system under design.

The basic structural statement is the component instantiation statement that names the instance, declares which entity is being instantiated, and associates signals and ports with the ports on the entity.

Assuming an adder and a register entity are available they can combined by creating a new entity/architecture pair that specifies how they are to be connected or wired together to build an incrementer.

A bit of bother is the component declaration (VHDL-93 does not require this as it supports direct instantiation) that is required in the architecture (or a supporting package) that specifies the ports on the instantiated entity (notice that the component declaration contains the same information as the entity declaration.)
At this point you have been introduced informally to the syntax of VHDL. This section attempts to informally present the behavior, or semantics, implied by VHDL. Together these informal introductions should allow you to write simple VHDL descriptions of digital circuits that will produce the intended synthesized circuit behavior.

The entity declaration as mentioned before simply names and describes the mode (in, out, inout, buffer) and type (std_logic, std_logic_vector) of the ports of the circuit. It does not impact the behavior of the circuit directly. The architecture is where the behavior is described. The biggest obstacle to writing synthesizable VHDL is that in VHDL same behavior can be described in multiple ways. Complicating the matter even more is that different, functionally equivalent, VHDL descriptions when given to a synthesis tool can produce different implementations that can differ in size and performance.

Digital systems are concurrent systems. In other words a system’s components are all simultaneously active. VHDL is a concurrent programming language and concurrency is expressed by the use of multiple processes. In other words, processes correspond to the simultaneously active components in a digital circuit. These processes can have arbitrarily complex behaviors. Single threaded behaviors are described by a series of sequential steps. Hence while processes all execute concurrently the statements within a process execute in order, but in zero time.

When writing VHDL the concurrent operations of the digital system are described using processes, concurrent signal assignments, and component instantiation statements within an architecture body. Processes allow the description of complex behaviors using sequential statements. Concurrent signal assignments are a shorthand for simple behaviors. And component instantiation statements allow describing systems in terms of the interconnection of entities (which consist themselves of processes, concurrent signal assignment statements, and component instantiation statements).

Any digital system that can be described in VHDL can be described as a collection of processes. Structural VHDL is simply a way of organizing the design such that processes reside in separate architectures. But the design could be collapsed into a single architecture containing all the processes and signal to interconnect them. All concurrent signal assignment statements have an equivalent process form.

Processes, concurrent signal assignments, and component instantiation statements communicate via signals within an architecture. Signals are the wires in VHDL that connect together the digital components described by VHDL processes, concurrent signal assignments, and component instantiation. Ports are also signals, and allow entities to communicate when they are attached to signals when instantiated within another architecture.
When describing systems for synthesis, circuit delays are determined by the target technology, and hence when writing VHDL for synthesis signal assignment statements never include a delay assignment. But all digital circuit elements have a delay, even if it is infinitesimally short. This 0ns delay (delta delay) has little impact on writing VHDL for synthesis but must be included so that all circuits will be physically realizable. Physical circuits always have have finite delays. Zero delay circuits, and circuit designs that depend on zero delay components could never be built. A finite non-zero delay corresponds to a unit delay.

The only time this non-zero signal assignment of RHS values to LHS values is noticeable is in processes where statements are evaluated sequentially. In this case multiple assignments always read the old value of a signal, never the new value. This has little consequence in combinational processes as the values will take on the new value in two delta delays and the circuit will behave as if the assignment to the signal occurred in zero time. In other words the combinational logic generated will be correct.

For clocked processes the result is quite different. In combination with the non-zero delay and the inference of flip-flops it becomes clear the Y is working with the old value of X. (The previously stored value of X). One way to understand this is that X does not take on a new value until a delta delay after the clock 0 to 1 transition and is to late to be assigned to Y. This behavior is identical to the behavior of flip-flops. Not matter the technology the output of the flip-flop does not change until after the clock edge and the output of a flip-flop is delayed by a full clock cycle when driving the input of another flip-flop.

The semantics of signals in VHDL is described by the simulation semantics of the language. Signals only take on their new values when the process has suspended during simulation. A process suspends when it reaches a wait statement. Either the explicit wait such as “wait until clk='1'” or an implicit wait defined by a sensitivity list “process(A,B,CLK)”. The sensitivity list implies a wait statement as the last statement in a process as “wait on A,B,CLK”. This notion of a process waiting on signals works for both simulation and as a good model of how digital circuits operate. The concurrent components of a digital system wait for an change on an input, and then update their outputs.

VHDL also support variables within processes. These do take on new values immediately. Typically variables are used to generate intermediate or temporary values within process. Variables should be used carefully as unless unconditionally assigned will generate a register to hold the value between invocations of the process. This can cause the variable to behave like a signal, defeating its use, and creating unintended registers.
Common Mistakes

1) Getting fancy. Keep your VHDL simple. If you are lost draw a block diagram and then write the VHDL for each block as a process in an entity/architecture pair connecting the processes with signals.

2) Being too simple. Remember digital systems are concurrent systems. Use multiple processes and concurrent signal assignment statements to break up the behavior of your circuit. A typical use of multiple processes and current statements is to separate sequential blocks from combinational blocks.

3) Forgetting your boolean algebra - A NAND B NAND C is not equal to NOT (A AND B AND C).

4) Precedence - VHDL does not assign precedence to the AND and OR operators. Hence given the equation A AND B OR C AND D you will get a compiler error. Parenthesis must be supplied. You will likely want (A AND B) OR (C AND D).

5) Duplicate assignments - In this course we can only have one concurrent signal assignment to a signal or port. If two drivers assign a common node in a circuit an error will occur both in the VHDL simulation and in the actual circuit. This is a very common error. Outputs cannot be connected directly together. This does not work in VHDL or in digital logic at the RTL level. You must use a multiplexer or a tri-state bus controlled by a conditional to enforce mutual exclusion on the shared signal. A signal can be assigned repeatedly in a single process, but not in multiple processes.
6) Unintended latches and flip-flops - A common problem is the creation (or insertion) of latches and flip-flops. Combinational latches typically arise when signals are not assigned values for all possible execution paths of a process or concurrent signal assignment statement. To avoid a latch and make sure that every signal is assigned a value under all execution paths, use a case statement or selected signal assignment statement, use an else part for if statements, or assign the signal a value unconditionally before any conditional assignment. Unintended flip-flops can also arise in clocked processes. All signals and variables appearing on the LHS of an assignment statement in a clocked process will be assigned to a flip-flop unless the signal or variable is unconditionally assigned a value.

7) Unintended hardware - In the following example equivalent behaviors result in quite different implementations. Both cause A to be assigned either the value $X+Y$ or $X+Z$ but the first uses two adders while the second uses one adder.

Don’t assume the synthesis tool will make obvious optimizations. If you want a particular implementation you must be explicit. Either in behavioral VHDL or by using a structural implementation.

```vhdl
if SEL = '1' then
  A <= X + Y;
else
  A <= X + Z;
end if;
```

```vhdl
TMP <= Y when SEL = '1' else Z;
A <= X + TMP;
```
### Basic Rules

- Always include the library and use declarations.
- Use the std_logic and std_logic_vector types. No need to get fancy with entity declarations.
- Adhere to the style guidelines.
- Use the architecture declarative region to declare signals.
- Use case statements instead of if statements.
- Simple combinational circuits should use concurrent statements.
- Signals in unclocked processes must be assigned a value for every possible execution sequence of the process to avoid unintended latches.
- Use case statements for combinational logic.
- If statements when used for combinational logic must include an else part to ensure all signals are assigned a value under all possible execution sequences of the if conditional.
- Use a process for clock circuits.
- Use “Wait until clk = ‘1’;” for a synchronous reset.
- “Wait until clk = ‘1’;” must be the first statement in a process and can only appear once.
- Use (clk’event and clk=’1’) for asynchronous resets.
- “(clk’event and clk=’1’)” must either be the only conditional of an if-then-end statement, or must be the elsif conditional of a if-then-elsif-then-end statement.

- The use of variables does not ensure the generation of combinational logic. The same rules as for signals must be observed.

- Understand the capabilities of your target technology and don’t ask through your VHDL synthesis system do more than it can. Remember you are describing hardware that consists of basic gates, registers, adders, and counters.